Imperial College London

Department of Computing

Automatic test case reduction of randomly generated OpenCL kernels

Author:
Moritz Pflanzer

Supervisor:
Alastair Donaldson
Andrei Lascu

Second marker:
Tony Field

Submitted in part fulfilment of the requirements for the degree of
Master of Science in Advanced Computing

3rd September 2015

Abstract

While software developers can review their programs to fix bugs, they might not be able to patch a compiler bug due to which their perfectly valid programs misbehave. Moreover, even a formal verification of source code provides no guarantees if the code is miscompiled. For these reasons great effort is spent to increase the reliability of compilers. One recent approach is compiler fuzzing in combination with random differential testing to detect miscompilations. Because the correct behaviour of randomly generated programs is not known they are compiled with multiple compilers or with different optimisation levels and the behaviour of the resulting binaries is compared to detect miscompilations.

Randomly generated programs are often large, sometimes by necessity, because this increases the chances that they cause a bug. On the other hand, for large programs it is more difficult to locate and extract the root cause of the triggered bug. Therefore automatic test case reduction was recently proposed in the context of testing C compilers. The key problem with the idea to automatically remove parts from test cases is to avoid undefined behaviour because it leads to useless reduction results if it interferes with the actual misbehaviour or is present in the final test cases.

In the scope of this project the existing implementation of the C-Reduce framework has been lifted to the domain of OpenCL kernels. In doing so the principal challenge has been to find appropriate analysis tools for the design of interestingness tests to prevent undefined behaviour, because the previously used tools are not compatible with OpenCL. It turned out that a reliable detection of undefined behaviour is only possible if the dynamic analysis of Oclgrind is used as oracle. To make this practical, the most significant achievement has been the design of a new plugin to reliably detect the usage of undefined values which have been identified as one of the largest sources of undefined behaviour which is overlooked by static tools.

Additionally, the contributions of this project include bug reports and patches for general issues with Oclgrind, a patch to fix a case of wrong code generation in the Clang compiler which has been discovered during the project and improvements of the transformations in C-Reduce. Further, parts of the C-Reduce algorithm for Windows have been reimplemented to improve the performance of the automatic reductions.

Finally, the robustness and performance of automatic reductions in the domain of many-core compilers have been evaluated. The reduced test cases are free from undefined behaviour and with less than one kilobyte small enough to be reported without the need for manual postprocessing. Moreover, the average reduction time of 100 minutes per test case is short enough to integrate the process into the everyday development.

First I would like to thank my supervisor, Alastair Donaldson, for the enthusiasm with which he supported this project. His strong interest in the subject and his guidance towards the important aspects of the project have been of great value. Moreover, he provided excellent feedback during the writing of the report which greatly helped to improve its overall quality.

Also I am grateful to Andrei Lascu who provided help for the usage of CLsmith and to the maintainer of Oclgrind, James Price, whose fast reactions on the bug reports were essential for the progress of this project. Further, I would like to thank the developers of C-Reduce for their willingness to accept my contributions to the C-Reduce framework and the Clang developers for the review of my submitted patch.

Finally, a very special thanks is due to Simon who told me Imperial will be a good match for me and kept me motivated after “bad” exams, to Sabine who finally convinced me to come here, reminded me to take some leisure time and supported me throughout the last year, and, of course, to my parents who took care of the financial aspects of my studies.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Compiler bugs are a great challenge in the area of software development because they can introduce wrong behaviour in previously verified code. This is especially worrisome for safety-critical systems such as health monitors or encryption software. Additionally, compiler bugs are hard to debug and it can take a long time before the compiler has been identified as the failing instance. To mitigate the risk of compiler bugs, projects have been started to develop *verified compilers* for which the optimisations are proven to be correct. However, their implementation of the C standard is limited and the number of available optimisations is small compared to GCC or Clang.

To increase the reliability of the non-verified compilers, the technique of *compiler fuzzing* has recently received significant attention. Specifically, the use of *random differential testing*, whereby the randomly[1] generated programs are compiled with multiple compilers, or a single compiler at different optimisation levels, to circumvent the oracle problem. Since for the automatically generated programs the correct behaviour is unknown it cannot be used as reference. Instead, any program that behaves differently than the majority of the compiled programs is considered as miscompilation. This approach makes it possible to evaluate the compiler on an unlimited number of programs to find potential bugs without significant manual effort. Previous work of Yang et al. [Yan+11] explains the principles behind this technique in more detail and provides a fuzzer implementation for the C language.

Most compiler bugs are caused by unsound optimisations which fail to handle edge cases correctly or depend on wrong assumptions. Since OpenCL compilers have to apply aggressive optimisations to translate the fairly general

[1] Throughout the report the word “random” is used in its informal sense and implies the technical meaning of “pseudo-random”.

OpenCL C language into highly efficient and device-specific code they are in particular at risk of invalid code generation. For this reason Lidbury et al. [Lid+15b] have lifted the compiler fuzzing strategy to the domain of many-core compilers and in particular added functionality to generate OpenCL kernels to the existing fuzzer for C programs.

One of the issues with compiler fuzzing is that the generated kernels have to be quite large – on average a few hundred kilobytes – to achieve a high success rate in triggering bugs. Locating and extracting bugs from test cases by hand is a time consuming task and hence cannot be afforded by the majority of developers. The alternative is the approach of *automatic test case reduction*, whereby a large bug triggering program is automatically converted it into a smaller program – a few hundred bytes are acceptable – which can be directly presented to developers and is just large enough to trigger the bug. So called *delta debugging* tools simply try to remove parts from source files without making the bug disappear until no further changes are possible. Regehr et al. [Reg+12] have implemented C-Reduce, a more sophisticated test case reducer for C programs which features specialised source-to-source transformations in addition to general delta removal.

However, just shrinking the size of test cases through automatic reducers will not yield the desired result of presentable bug reports. If no extra care is taken the test cases themselves can become syntactically invalid or undefined behaviour can be introduced during the reduction which makes them useless. Examples of undefined behaviour in C-like languages include the usage of uninitialised variables or dereferencing a null pointer. A crucial part of the reduction process are therefore *interestingness tests* which are executed after each alteration of a test case. The tests have to check whether the program can still be compiled and executed, is free from undefined behaviour and finally still triggers the “desired” misbehaviour. Additionally, with an average number of 30 000 invocations of the interestingness test per reduction they have to be fast to obtains acceptable runtimes. While static analysis tools fulfil the requirement of having a short execution time, they cannot be as precise as dynamic analysers.

The principal aim of this project has been to lift the C-Reduce framework for reduction of C programs to the context of OpenCL, to enable automatic test case reduction for parallel OpenCL kernels. Combined with the test case generation method of Lidbury et al. [Lid+15b], this has the potential to provide a fully automatic method for discovering small OpenCL kernels that trigger compiler bugs. While the C-Reduce framework in general is not specific to a particular language and thus could be used without larger modifications, the main challenge has been to construct suitable interestingness tests. The previously used tools to detect indications for undefined behaviour in test cases are not compatible with the OpenCL C language and had to be substituted.

For this reason Oclgrind has been included in the interestingness tests as a dynamic analysis tool. But since its provided detection of undefined values caused several problems, a more precise plugin has been developed in the scope of this project. The new plugin increased the accuracy and robustness of the interestingness significantly and finally allowed to apply the automatic reduction to OpenCL kernels. Though the plugin has been developed with its usage in the interestingness tests in mind, it improves the quality of Oclgrind in general and will be useful in various other situations too.

In addition to the new plugin the contributions of this project include the discovery of bugs in Oclgrind as well as a patch for a bug in the Clang compiler to correct a wrong code generation for vector expressions. Furthermore, some transformations of the C-Reduce framework have been improved and performance of the framework itself on Windows has been increased.

Finally, an evaluation of the robustness and performance of automatic reductions of OpenCL kernels has been conducted. It confirmed the hypothesis that automatic reductions can be successfully applied to OpenCL kernels resulting in small test cases and being fast enough to be integrated into the everyday development process. Moreover, as result of the automatic reductions several bugs in OpenCL compilers have been detected and were confirmed by the developers after they have been reported.

The rest of this report is organised as follows: Chapter 2 presents related work in the field of compiler fuzzing and automatic test case reduction as well as concepts of the detection of undefined values during a dynamic analysis. In Chapter 3 discovered problems and proposed solutions are explained. This includes small patches to the C-Reduce project, the Clang compiler and Oclgrind but also the redesign of the parallel interestingness tests of C-Reduce on Windows and a new plugin of Oclgrind to detect uninitialised values. The results of the automatic test case reduction of OpenCL kernels are shown in Chapter 4. In the course of this project the performance in terms of the size of the reduced test cases as well as the runtime of the automatic reductions were the primary targets of the experiments. Chapter 5 evaluates the results and compares them to the existing tools. Finally, Chapter 6 summarises the project and provides a short outline of open questions and possible continuations.

2
Related work

This chapter summarises the techniques on which this project is based on. First an introduction in the field of *compiler validation* with focus on automated compiler testing is given. Further, a detailed description of two implementations of tools for a *detection of undefined values*, e.g. uninitialised variables, is presented.

2.1 Compiler validation

According to Regehr et al. [Reg+12] the validity of an optimising compiler can either be ensured by verifying that the compiler works correctly or by testing the compiler and its output. A formal proof of validity provides a stronger guarantee than testing but is often not practical or not complete. Standards like the C++0x language comprise large, complex and also imprecise specifications which makes it hard to imagine a verified compiler over the complete set of features. The verification therefore concentrates on small subsets of the language and only specific parts of the compiler such as the middle-end. Testing on the other hand is able to provide an end-to-end evidence for the correct functionality of the tested inputs. It thus complements the aims of a verified compiler. Moreover, it is much easier to create valid inputs which can be used for testing than to proof the correctness of an optimising compiler. There are mainly two different types of errors a compiler can be tested against. First, *crash errors* which occur whenever the compiler aborts during the compilation or does not produce an executable program. Second, *(silent) wrong-code errors* due to which the produced output computes a wrong result (without any warning during the compilation).

2.1.1 Random Differential Testing

The general idea of testing a compiler includes the generation of *valid* source code of a program for which the output after the execution is already known or can be gathered from the source code itself. The code can then be compiled with the compiler and if it does not lead to a crash error the compiled program can be executed. The result of this execution can be compared to the correct result and if they differ the compiler has produced a wrong-code error.

In contrast, as Regehr et al. point out, *Differential testing* does not need an oracle for the correct output. Instead the same source code is compiled with different compilers. The results of the runs of all produced programs are compared against each other. If they are not all the same at least one compiler must have produced a wrong-code error. The wrong compilers can be determined according to a majority vote. The largest group of compilers for which the executable programs produced the same result are considered as correct and all others as wrong. In their paper Regehr et al. [Reg+12] state that in their experiments they have not come across a case in which the vote was misleading or in which two compilers produced the same wrong code. Therefore differential testing is a reasonable approach to find wrong-code errors without the need of an external oracle. The only requirement on the programs is that they have to be deterministic.

For the purpose of *Random differential testing* Regehr et al. have developed the random program generator *Csmith*. Testing with random programs (*fuzzing*) is a black-box approach in which the output of an execution of the generated programs is not known. But since differential testing is applied this is also not necessary as long all implementations have deterministic behaviour. The standard of the C99 language lists 191 kinds of *undefined behaviour* and 52 kinds of *unspecified behaviour*. Both have to be avoided as they would allow for non-deterministic behaviour of the programs. A third group of *implementation defined behaviour* is not avoided by Csmith. The authors claim this would limit the expressiveness of the generated programs significantly resulting in programs with only a small set of features. Furthermore, as there are only a few different implementations among the compilers it is no problem to test only within a group of the same implementation details.

In Csmith a large effort has been spent to prevent non-deterministic situations. The grammar-based code generation is combined with a static analysis of the code but nevertheless some properties have to be enforced through hard-coded dynamic checks, e.g. checks for null pointers before dereferencing. The first step consists of creating declarations of random struct types which can be used in the program. Subsequently a top-down approach is used and the program is generated originating in a single main function. In order to add a new part to the program Csmith first uses the grammar to determine the kind of expression that is need and then consults a probability table to choose from one of the possible options. After each change the validity of the new program is tested and only if all safety checks pass the new part is finally added. Most of the changes only affect the *local safety* and can be checked efficiently or can be avoided at all. Changes that involve function calls and loops also affect the *global safety*. Since a complete dataflow analysis after each change would come with high performance costs Csmith creates locally safe code in a greedy manner and only validates the global safety at certain defined points. If then an invalid situation is determined the program is rolled back in a controlled way until a safe point is reached. For instance, in the case of generating loops the global safety is only evaluated when the closing bracket of the loop body is added. An invalid situation is solved by removing lines from this point up the loop until no further violation exists.

2.1.2 Validation via Equivalence Modulo Inputs

Instead of generating new programs as inputs for differential compiler testing Le et al. [LAS14] present the strategy of *Equivalence Modulo Inputs* (EMI) which focuses on the generation of variants from existing programs. Two programs are equivalent with respect to their inputs[2] if they have the same side effects during their execution, e.g. produce the same output. Although two such programs are only semantically invariant under the given inputs the optimisations and transformations of the compiler have to be valid for all possible inputs. Despite being semantically equivalent the programs are likely to vary in their data- and control-flow which will trigger different compiler optimisations. The idea of creating EMI programs helps the authors to (completely) avoid the test case validity problem since as long as the original program has been free from undefined behaviour their modifications will not affect the validity. The authors mention a few more advantages of their approach. By using existing real world programs as basis the chances increase to detect compiler errors that "more directly impact software vendors and users" [LAS14]. Furthermore, this method of differential testing does not require a second instance to verify the correctness of a compiled program. The output of the original variant can be captured by simply executing it. All the generated variants have to produce the same outputs since they are equivalent to the original version by construction. If one of the variants produces different output the compiler must by erroneous. And lastly, the complexity of their EMI generator is much lower than the complexity of test case producers like Csmith which makes it more easily extendible to other languages.

[2] Throughout this section *equivalent* always means *equivalent w.r.t. the inputs* if not stated otherwise.

In practise the search space for variants can be gigantic especially if the base program is large. The authors solve the problem of how to efficiently produce variants by following a *profile and mutate* strategy. As a first step they run a static code coverage analysis on the original program to find regions of *dynamically dead code* under the given inputs. Since these regions are not executed they can be freely changed as long as the program as a whole remains syntactically correct. In their paper the authors only present a basic pruning approach. A probabilistic algorithm decides which regions of unexecuted code should be removed. All modification are performed in the abstract syntax tree (AST) which guarantees that the program stays valid.

2.1.3 Many-Core Compiler Fuzzing

While the authors in the previous sections mainly focused on testing of optimising C- and C++-compilers Lidbury et al. [Lid+15b] adapt and extend the random differential testing and EMI techniques to many-core OpenCL compilers. One feature of OpenCL is the online compilation of kernels during the execution of the host program. This makes it more difficult to circumvent bugs than with the general offline compilation of C-like programs and thus compiler reliability becomes even more important.

The ability to apply random differential testing was one of the key aspects of the prior work on fuzzing C-compilers since it allows to avoid the oracle problem. The authors compared the output among various compilers against each other or used just one compiler with different optimisation options. When targeting OpenCL the compilers are specific for each device and there is only the option to either enable or disable optimisations. This restricts the applicability of random differential testing of OpenCL compilers to some extend but nevertheless Lidbury et al. found a significant number of bugs by using just the comparison between the two optimisation stages.

Also the EMI approach faces two problems when lifted to OpenCL kernels. First, there are currently no code coverage tools available for OpenCL. But this information is needed to find regions of dynamically dead code which are used to create the variants. Second, even if such a tool existed their is still the problem that most of the OpenCL kernels do not contain regions of dynamically dead code. To bypass these difficulties the authors propose the concept of injecting *dead-by-construction* code into existing kernels. Instead of finding regions of dynamically dead code a new block is generated for which the contained statements are dynamically unreachable. The latter property can easily be enforced during the initialisation of the inputs in the host program. This generated block is then inserted into an existing kernel and variants are created by applying the pruning strategies suggested by Le et al. [LAS14] to the interior of the block. In addition to deleting a leaf node or a branch node from the abstract syntax tree of the block a third transformation is introduced. It allows to move the children of a branch node to the parent node before the branch node is finally removed.

In terms of the test case validity for OpenCL programs basically the them rules as for the C programs that the other authors analysed apply. OpenCL is an extension to the C99 standard and thus inherits most of its undefined behaviour. Additionally, in OpenCL the risks of data races and barrier divergence are introduced due to the concurrent execution of kernels. Barrier divergence describes the situation in which two threads reach different barrier and thus fail to synchronise. It can occur if the threads dynamically execute different control flow paths. To shield against these kinds of undefined behaviour Lidbury et al. [Lid+15b] use only dedicated data structures and explicit memory barriers for the communication between the threads. The access to these data structures follows strict rules which are enforced during the construction and which make it impossible for data races to occur. Moreover, local expressions must not depend on the local or global thread id and global variables are initialised uniformly.

All the mentioned techniques are implemented in the tool *CLsmith* which extends the existing Csmith random program generator. It comprises six modes for the program generation. The simplest BASIC mode does not use any OpenCL related features despite from a global memory resource to collect the final results of the computations. Since global variables are not permitted in OpenCL the Csmith algorithm had to be adapted. Instead the kernel function creates a struct which is then passed as pointer into all functions to simulate global variables. As a side effect this primes the CLsmith tool to find a lot of struct related bugs in the kernel compilers. In the VECTOR mode setting special instructions and data types for data parallel vector arithmetic are used in the kernel. Since OpenCL does not allow arbitrary conversions between vector types the generation has to be performed context sensitive. The threads in these modes do not communicate at all among each other but this is added through the next two modes. The BARRIER mode maintains at each point during the execution a permutation of the kernel ids and assigns each kernel a unique number from the range of the ids. It is used as an offset into a globally shared resource through which the kernels are able to exchange data. At barrier points inside the kernel the parallel executions synchronise and the ids are reassigned. This procedure guarantees race free execution and deterministic results. Another form of interaction between the kernels is targeted in the ATOMIC SECTION mode. By using an atomic increment operation on a shared variable only one thread is allowed to enter a special conditional section. Inside this section special care has to be taken that only variables within the current scope are changed and the section is not left in an uncontrolled way to ensure that the kernel has well-defined behaviour. Furthermore, the ATOMIC REDUCTION mode uses atomic operations to compute a shared result. After the atomic operation the threads synchronise a first time to allow the result to be added to the final output and a second time that ensures that all threads have a consistent version of this result.

2.1.4 Test case Reduction

The authors of Csmith [Reg+12] empirically determined the highest bug finding rate in test cases with a size of 81 kB. This conflicts with the requirement of bug reports to be small and optimally also easy to understand.[3] Even for small files reducing a test case while preserving the wrong behaviour is a time consuming task and for large test cases it becomes infeasible.

Delta debugging

In their paper Yang et al. [Yan+11] mention *Delta debugging* as an alternative to manual test case reduction. Nevertheless the task of test case reduction is hard since there exists no method to predict whether a variant will reproduce the error. Therefore most of the time naive approaches simply make use of greedy elimination strategies. By removing contiguous parts from the current test case they create variants and evaluate whether these new files still create the determined misbehaviour. If no successful variants can be produced the chunk size is decreased and the search is started again.

The authors found two main problems with these tools. Due to the greedy search these tools often get stuck in local minima which means that the reduced files are still too large to be submitted as a bug report. However, the major problem is that the tools fail to address the *test case validity* issue. They potentially will produce variants which contain undefined behaviour which is not detected by the compiler. Once such an invalid variant has been accepted it is no longer possible to distinguish between the misbehaviour which is caused by the actual bug and effects that are introduced by the undefined behaviour.

Generalised delta debugging

To prevent these problems Yang et al. propose the concept of *Generalised delta debugging*. This extends the general delta debugging approach in three points. The first two extensions aim at the prevention of local minima which would lead to smaller reduced files.

First, in addition to the removal of contiguous substrings transformations based on the semantic of the test case are added to allow changes at multiple positions spread across the file in one step. Second, the greedy search is partially lifted to allow intermediate results to increase in size. As a third

[3] The guide for reporting bugs in LLVM for instance directly asks “to narrow down the bug so that the person who fixes it will be able to find the problem more easily” (`http://llvm.org/docs/HowToSubmitABug.html`, visited on 27/05/2015).

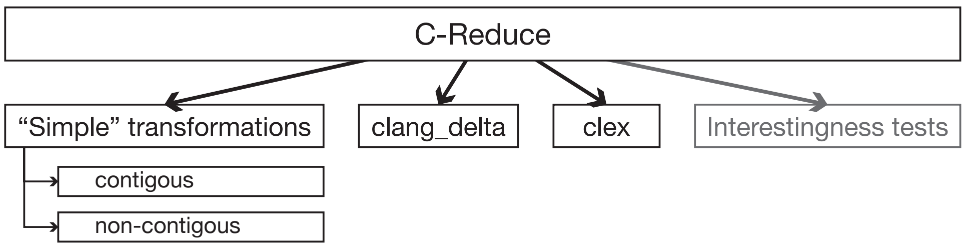


Figure 2.1 – Visualisation of C-Reduce's modular components. Only the top level C-Reduce module is exposed to the user. It is connected to the different modules that C-Reduce comprises and invokes them as necessary. The "Simple" transformations are implemented in Perl and can be categories whether they perform contiguous changes in the test case or not. The clang_delta and the clex module are helper tools to perform more complex changes based on the AST or a lexical analysis of the test case. The interestingness tests are rather an external dependency as a real module. But they are also required by the C-Reduce implementation.

point the test case validity is addressed by using external tools in addition to the tested compiler.

C-Reduce

In particular the authors developed the tool *C-Reduce* which is an implementation of their proposed generalised delta debugging approach. It consists of three different modules which comprise various kinds of transformations. The modules are automatically managed by a single main interface and are thus transparent to the user. Figure 2.1 visualises the different modules.

The most basic transformations just perform contiguous changes such as changing the value of integers or removing parts of arithmetic expressions or text segments within surrounding brackets. Transformations of the next level make non-contiguous changes like removing just pairs of brackets without deleting the contents. Further, traditional delta debugging routines are applied which remove parts on a per line basis and change the format and indenting style of the file.

The second category comprises a set of semantic aware source-to-source transformations based on the abstract syntax tree of the program. They cover a wide range of language specific changes from the removal of unused or statically dead expressions and functions over the alteration of types up to complex code refactorings. Additionally they are carefully designed such that they do not introduce undefined or unspecified behaviour. All these transformations are bundled in the *clang_delta* helper tool which is written in C++ to make use of the Clang AST parser.

Finally, in addition to the context-free AST parser a custom lexer for the C language (*clex*) is used as the basis for the third kind of source transformations. The generated tokens are either changed or deleted independent of each other or alternatively modified according to specific patterns. These types of transformations make explicit use of the flat representation of the tokens as opposed to the parsing tree and furthermore the lexer is more efficient due to its smaller capabilities.

Transformation cycle

The reduction process of C-Reduce consists of three nested loops as shown in Figure 2.2. The outermost loop (red area) determines when the reduction ultimately finishes by comparing the size of the best test case with the size of the best test case from the previous iteration. As long as the test case gets smaller the reduction is continued.

The middle loop (blue area) iterates over all available transformations and creates a clean transformation state in each cycle. During the initialisation all necessary resources including counters and the input and output files are prepared. In the innermost loop (green area) the current transformation is repeatedly applied to the respective current source code until the transformation space is exhausted or an error occurred during the transformation.

After each alteration of the source file a check is performed whether the transformed code is still *interesting*, e.g. is free from undefined behaviour and exhibits the wrong behaviour. On success the modified code is used as basis for further transformations whereas otherwise the produced variant is discarded and the state is advanced, e.g. the next token is considered for deletion.

In either case the transformation loop is continued because successfully applying a transformation has the same effect as advancing the state explicitly (Listing 2.1). In the first case one item is removed from the transformation space. Therefore the next item that was going to be transformed becomes the new current item and applying the same instance of the transformation again will produce a new result. In contrast, if the source code is not changed applying the same transformation would not be of any use. Hence the transformation state is explicitly advanced also making the next item the new current one. Thus in both cases the progress is guaranteed and at some point the exit criterion has to be met.

Parallelizing Delta Debugging

The key feature in this design is the existence of exactly *one* best, i.e. interesting, test case at any time. This greatly simplifies the delta debugging process since every step consists of applying a transformation to the currently best result, checking the transformed source for its interestingness and either using it as new best result or discarding it. There is no need to determine which test case would be the best to apply the transformation to.

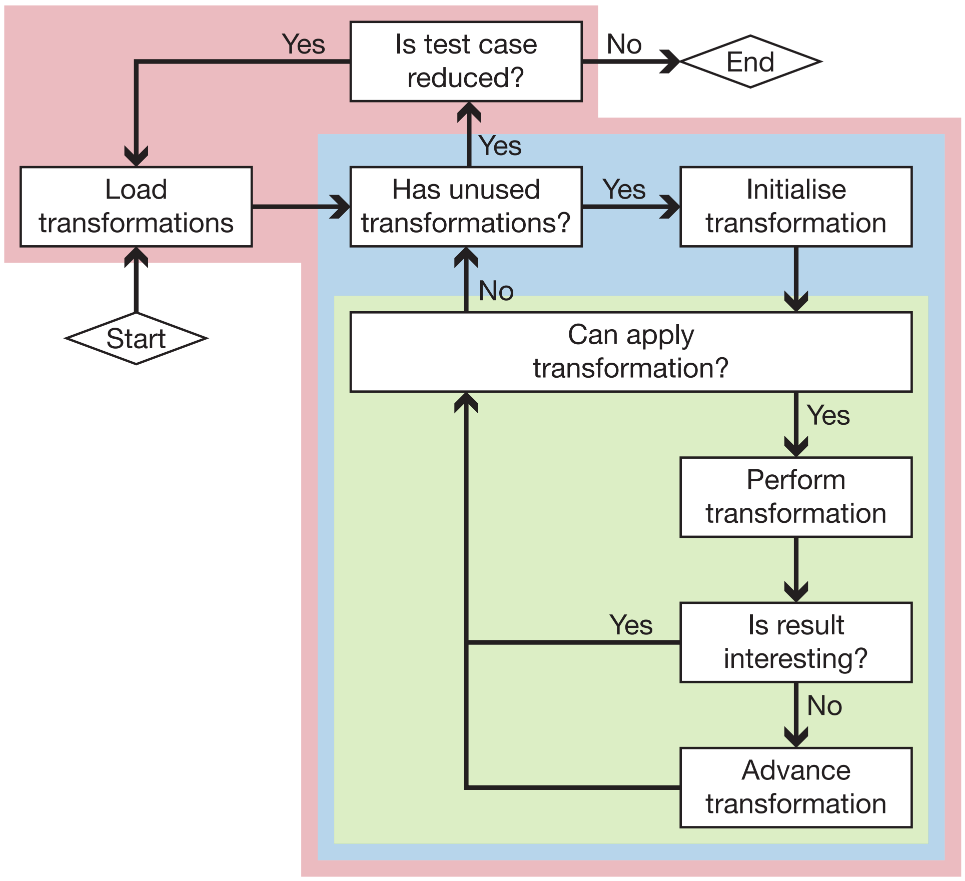


Figure 2.2 – Schematic representation of the transformation cycle in C-Reduce. At the beginning all available transformations are loaded. Then an unused transformation is selected and initialised. It is repeatedly applied as long as it produces interesting results (green area). After an invalid result the transformation state has to be advanced to skip the uninteresting change. Once the transformation space of this transformation is exhausted the next unused transformation is chosen (blue area). Finally, if all transformations have been applied the size of the current best test case is compared to the size of the best test case from the previous round of transformations or the initial test case. If the test case has been reduced another round of transformations is performed (red area) otherwise the reduction ends.

```
struct S {
    int f;  // member 1
    char g; // member 2
    int h;  // member 3
};
```

(a) Initial test case

```
struct S {
            // removed
    char g; // member 1
    int h;  // member 2
};
```

(b) `remove-unused-member 1`

```
struct S {
            // removed
    char g; // member 1; reduction failed
    int h;  // member 2
};
```

(c) `remove-unused-member 1`

```
struct S {
            // removed
    char g; // member 1
            // removed
};
```

(d) `remove-unused-member 2`

Listing 2.1 – Progress in transformations of C-Reduce. The transformation in this examples tries to remove unused members from the given struct (a). It is assumed that the member named "g" cannot be removed. The first time the transformation is applied (b) the first member is successfully removed and the counter is not increased since member g has automatically become the new first member. The second time (c) the transformation fails and "g" remains the first member. Therefore the counter has to be increased to remove the member after "g" the next time the transformation is applied (d).

Unfortunately, this invariant causes the reduction process to be inherently sequential. As soon as two transformations are executed in parallel and both produce interesting results the invariant would be violated and it would not be clear how to restore it.

The nearest solution is to relax the invariant and to introduce a merging step as described by Regehr [Reg12]. The described solution involves to start multiple instances of the delta reduction loop in parallel to increase the performance. The invariant would still hold locally for each of the instances but nevertheless the different reduction paths have to be interleaved at some points to combine the positive effects of all globally performed transformations. Hence, in each step instead of applying another transformation it could be chosen to merge changes from a different instance into the locally transformed file. One possible execution is illustrated in Figure 2.3.

The main problem is to define an appropriate merge operation. The traditional merge tools are far to conservative and warn eagerly about conflicts [Reg12]. While this is important if the exact behaviour of the merged files has to be preserved it is not suited for the purpose of delta debugging. There it is not important to obtain the same behaviour after the merge but only that the result is still interesting independent of the actual computation. Further, it remains unclear how often a merge would be necessary to get the maximal benefit from the parallel reductions without wasting time on uninteresting merges.

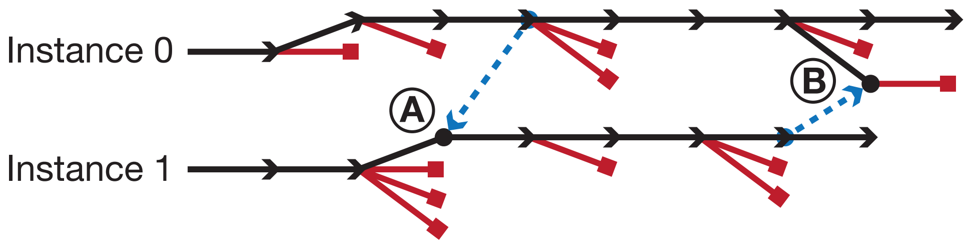


Figure 2.3 – Merging parallel reductions. Both instances run independent of each other most of the time. The individual transformations are represented through the edges and are executed sequentially from left to right and bottom to top. Arrows symbolise successful transformations and squares uninteresting ones. At point Ⓐ the second instance merges its locally best test case with a test case from the first instance instead of applying another transformation. The merge operation produces a new valid test case. On the other hand, the merge operation in point Ⓑ does not result in an interesting test case and has to be discarded.

Since these problems seem to be hard to solve for arbitrary test cases Regehr proposes an alternative solution in his article [Reg12]. The idea is to keep the invariant of one best file at any time but to execute multiple successive transformations on this file in parallel. Each transformation is followed by an interestingness test and when it finishes the result is stored and the transformation which would also be the next in a sequential execution is started (Figure 2.4).

The list of results is then searched in the order in which the transformations have been *started* – which is not necessarily the order in which they finish. Unsuccessful transformations are simply discarded and the search is continued until either a transformation produced an interesting test case or the next transformation in the list is still running. In the latter case the evaluation of the list of results has to be paused until the result is available but in the mean time it is still possible to start new transformations and interestingness tests. More interesting is the case when a transformation has been successful. Then all *later* transformation have to be aborted and their results have to be discarded since the best file has changed and retroactively they have been run on the wrong test case. The reduction is then continued with the next transformation after the one which produced the new best test case.

The improvement over a serial reduction is based on the assumption that in general most of the transformations will fail to produce interesting results. The failing transformation can be executed in parallel without inferring with each other. Only in the situation when a transformation has been successful all later transformation have to be restarted with the new test case as basis. Moreover, as long as the parallel transformations do not affect each others performance this parallel approach cannot be slower than the serial reduction. In the worst case, i.e. all transformations are successful, the transformations

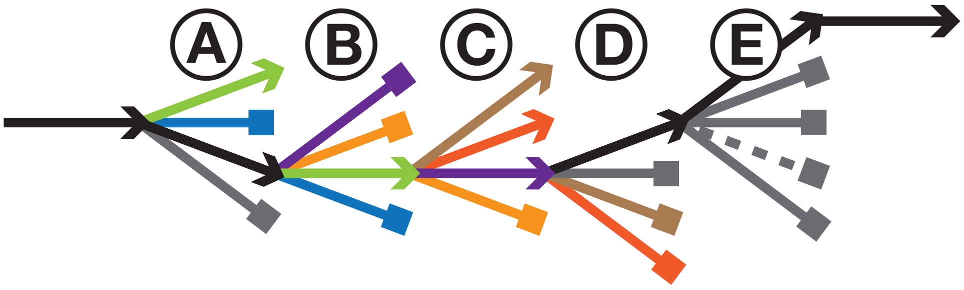


Figure 2.4 – Parallel transformations in C-Reduce. At every stage four transformations are executed in parallel represented through the edges. Arrows are successful transformation and squares indicate uninteresting results. The transformations are started in left to right and bottom to top order. At stage Ⓐ the second transformation already succeeds. Therefore the other two transformations which run in parallel but have been started later must be repeated in stage Ⓑ. There they still produce the same results and again two transformations have to be carried over to stage Ⓒ. The orange one still fails but the purple transformation succeeds now in Ⓒ. This is possible because the test case has changed between the first run in Ⓑ and the repetition. In stage Ⓓ all repeated transformations fail and only the last transformation is interesting. Thus no transformations have to be repeated in the next stage. Stage Ⓔ comprises five transformations since one finished early (dashed line) and the next transformation has been started. Nevertheless, there were only four transformation running in parallel at any point in time.

would be restarted after every transformation. But since it is only ever waited for the first transformation in the result list running more transformations in the background does not affect the runtime which would be equal to the one of a serial reduction.

Finally, the new parallel approach does not change the behaviour of the reduction since the transformations are executed in a sequentially consistent order. The parallel transformations are started in the order in which they would have been executed during the sequential reduction and the list containing the test results is searched according to the *first-in-first-out* (FIFO) principle. On detection of a successful transformation state is reset to the transformation directly after the successful one such that no transformations are skipped during the reduction.

Interestingness tests

The validity, i.e. interestingness, of the transformations is checked by first compiling the variations with different compilers and compiler options to allow for a fast fail if the program is syntactically invalid or if the compilers warned about misbehaviour. Afterwards the static analysis tools *Clang Static Analyzer*[4] and *Frama-C*[5] are used. Variants which exhibit dynamically

[4] `http://clang-analyzer.llvm.org`, visited on 27/05/2015.

invalid behaviour are discarded through the analysis with the semantic-based interpreter *KCC*[6] and the instrumentation framework *Valgrind*.[7] The authors point out that without the dynamic checks still 29 % of the programs comprised undefined behaviour which would have made bug reports based on these files useless. Besides the attribution of the importance of test case validity the C-Reduce tool is able to produce reduced test cases which are free from undefined behaviour and small enough to be directly included into a bug report.

2.1.5 Taming compiler fuzzers

In the field of automated bug detection random program generators or compiler fuzzers have proven themselves of great value. They triggered thousands of previously unknown bugs and thus helped to improve the quality of software. As useful as it seems at a first glance the enormous amount of failure generating test cases can become a problem for the user of the fuzzer. The test cases often contain a large number of duplicates and they might trigger bugs of little importance for the time being.

To mitigate these problems Chen et al. [Che+13] address in their paper the *Fuzzer taming problem*. The problem describes the situation that given a collection of test cases it has to be sorted by relevance and diversity such that tests triggering distinct bugs appear first. This would help the user to concentrate on only a few test cases out of the many generated while covering most of the triggered bugs and thus increasing greatly the effectiveness of random program generators.

Previous ranking algorithms relied on user provided metadata about the different test cases as basis for the ranking. But, due to the large size of automatically generated programs and their complexity even just collecting information can become a time-consuming and hence costly task. Chen et al. instead couple the fuzzer taming problem with the application of test case reduction to automatically retrieve information about the test cases. The authors conclude that reducing a test case as far as possible is essentially the same as extracting its characteristic features since nothing else will be left over. Moreover, the reduction can be seen as a variant of *fault-localisation* because (ideally) nothing else than the fault triggering instructions will remain in the test case.

An ideal ranking would require to compute the distance between the various faults that are triggered through test cases. In practise this is not possible as the faults can not be observed directly [Che+13]. Instead the distances have to be estimated based on the similarity of the visible side

[5] `http://frama-c.com`, visited on 27/05/2015.
[6] `https://github.com/kframework/c-semantics`, visited on 27/05/2015.
[7] `http://valgrind.org`, visited on 27/05/2015.

effects. These are for instance the compiler output during the compilation, potential crash messages, the output of the compiled program and lastly the test case itself. In their paper the authors mainly differentiate between the Levenshtein distance which is applied to any textual output and the test cases themselves, and the Euclidean distance. The later is used to compare domain-specific feature vectors which are derived from the test cases and again compiler outputs. The authors describe the reason why they use multiple distance functions over various inputs is that a priori it is not known in which way a fault will manifest. Therefore they try to cover as much static as well as dynamic information as possible to reduce the risk of missing the characteristic of a bug.

The authors evaluated two different methods to perform the ranking based on the defined distance functions. The first method is based on *X-means clustering*. Although creating a ranking is not directly a clustering problem it can be seen as such. After all clusters have been determined theoretically all distinct bugs in the set of test cases can be retrieved by picking one representative member out of each cluster and ranking them by the quality of the cluster. Nonetheless, the second method based on the *Furthest point first* (FPF) method performed better and was less complex. Despite the fact that it also tries to approximate clusters it directly generates an ordering in which the most distant points are selected first. For the purpose of test case ranking the clustering information has simply been discarded.

The authors present results which show that with their ranking scheme less test cases have to be considered to find all distinct bugs compared to a random order. Further, they noticed that the distances based on the test cases themselves are an important factor but also remark that their approach to vectorise the test cases into domain-specific feature vectors failed to improve the accuracy of the ranking. Furthermore, for compiler crash bugs the distances based on the compiler output helped to improve the quality of the ranking and for wrong-code errors the function coverage as reported by the compiler was of great use.

In addition to the goal of ordering the test cases produced by a compiler fuzzer Chen et al. analysed the potential to exclude previously marked test cases or to give them a low rank respectively. This could be useful if some bugs have been identified in earlier runs but have not been fixed. Then clearly these bugs should not be reported again as they are already known. To achieve this secondary goal the authors seeded the FPF algorithms with examples of the already known bugs to implicitly lower the ranking of duplicates in the real set of test cases. They claim that the obtained results are “reasonably good” but also suggest that classification instead of clustering might be a better approach.

2.2 Undefined value detection

A common error in programs involves the usage of undefined values such as uninitialised variables. Furthermore, the origin of the error cannot be observed directly but only the consequences once the observed behaviour does not match the expected one. This makes these errors particullary hard to track down. To support developers, various tools have been proposed and implemented over time; all of them more or less precise and sophisticated. This section introduces the concepts of the two most prominent tools.

2.2.1 Valgrind's Memcheck

Valgrind[8] is a dynamic binary instrumentation (DBI) framework which can be used to analyse the behaviour of programs during their runtime without the need to recompile them manually. Valgrind follows the *disassemble-and-resynthesise* (D&R) paradigm[9] [NS07b] and translates the original instructions of the loaded binary into its custom "UCode" intermediate representation (IR). This RISC-like IR can be accessed from the plugins based on Valgrind to implement the desired analyses.

The check for undefined values is part of the *Memcheck* plugin of Valgrind [SN05]. It is able to detect "invalid" usages of undefined values such as uninitialised variables at bit-level precision. Thus also partially initialised bytes can be handled accurately and a low false positive rate can be achieved. Examples for partially defined values are struct members which have been declared as bit fields or the more general bit arrays.

In their paper Nethercote et al. [NS07b] identify nine requirements parted into three categories which have to be met in order to fully support shadow values. The three categories are described as the current state of a program, instructions which read or write memory and instructions which allocate or deallocate memory.

Shadow values

In the context of evaluating undefined values the state of a program is defined as the content temporarily held in registers and the values stored in memory. To support a detection with bit-level precision every bit of data has to be *shadowed* with an additional *definedness bit.* Valgrind's Memcheck plugin calls these shadow values *V bits* (validity bits). If such a V bit is set to zero the corresponding data bit is considered as defined whereas a value of one for the V bit specifies an undefined data value.

[8] See `http://valgrind.org`, visited on 15/08/2015.

[9] The alternative is the *copy-and-annotate* (C&A) paradigm which uses the original instructions and maps a description to each one.

The data registers are shadowed through a simple one-to-one mapping to shadow registers. Both data registers and shadow registers are treated as *first class entities* [NS07b] and can be modified through the instructions provided by the intermediate representation. This makes it easy to keep the data and their shadow values synchronised.

Compared to the data held in registers a much larger amount of shadow memory has to be maintained. Thus the Memcheck plugin uses a more sophisticated algorithm to increase the efficiency of the implementation. The details of the organisation of the shadow values are described by Nethercote et al. [NS07a]. Not all data structures which are described in the paper are exclusively used to store undefined values. However, the purpose of this section is to provide a basic idea about the mechanism behind the part of the plugin which deals with undefined values rather than to explain every detail of the design. Thus – to keep this section focused – some information is omitted or simplified as long as the changes do not affect the fundamental principles. Places that have been changed are indicated through footnotes which provide some more information but for details about the concrete implementation it is referred to the paper.

In its basic implementation a two-level mapping scheme is used instead of the one-to-one mapping (Figure 2.5). The first level table (PM) divides the 32 bit address space[10] into 65 536 blocks. Every entry in the table points to a secondary table (SM) containing 65 536 entries to shadow 64 kB of memory. Space in the second level table is allocated on demand (*copy-on-write*) and deallocated together with the data values. Thus not all entries in the first level table are set.[11] The design of the shadow tables is thus similar to the layout of real page tables.

Every time a data access requires an update of the corresponding shadow values the address of the actual data is used to index the shadow tables (Figure 2.5). The upper half of the address (16 bit) is used as key in the primary map whereas the lower half (16 bit) selects a single byte in the secondary map. The precise handling of individual bits is left to the functions which requested the shadow values.

To reduce the memory footprint of the shadow values the Memcheck plugin "compresses" them where possible. Therefore a slightly different table layout has been implemented. The primary map is not changed but the secondary map encodes three different states by using only two bits per entry instead of the previous eight bits per byte of shadow memory (Figure 2.6).[12] The states "DEFINED" and "UNDEFINED" indicate that all bits of the

[10] The Memcheck plugin has been designed with a 32 bit address space in mind. An extension for 64 bit address spaces exists but is excluded from this report as it is not essential to demonstrate the general principle.

[11] In the actual implementation those "unset" pointer point to a special *distinguished* SM which defines all entries as "valid".

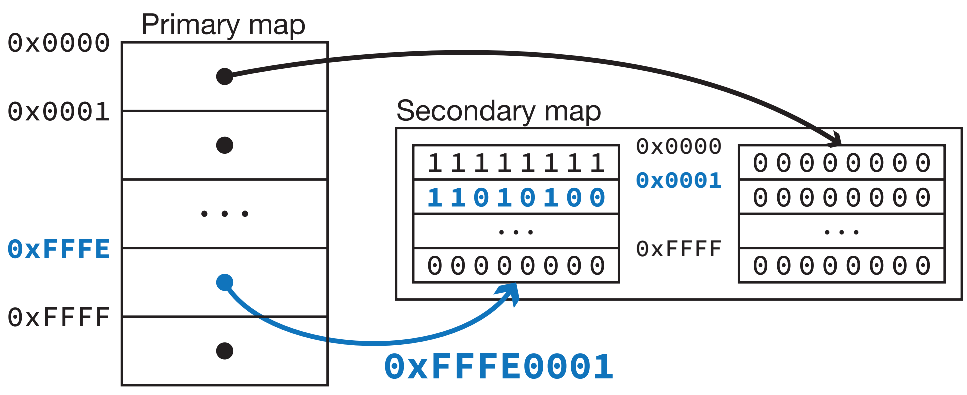


Figure 2.5 – Design of the two-level tables layout of shadow memory. The first level table (PM) consists of 2^{16} entries which can point to a block in the second level table. Each block shadows 64 kB of memory. Thus the complete 32 bit address space can be shadowed. To access the shadow values the address of the actual data request is used. The upper half indexes the primary map whereas the lower half of the address selects the corresponding byte of shadow values from the secondary map. Access to individual bits is not managed through the tables.

corresponding byte have the same status. In this case no further lookup is required. In contrast, the third state "PARTDEFINED" denotes that the V bits have to be loaded from an additional sparse table.

Although lookups of partially defined shadow bytes are less efficient in this design they are rare and do not contribute much to the overall performance. Moreover, the changes make most of the other lookup operations more efficient and the cache locality is improved which results not only in a smaller memory footprint for this design but also in shorter runtimes [NS07a].

The initial state of the shadow registers and the shadow memory is straightforward: In general everything is considered as undefined at program startup registers the corresponding shadows are thus set to all one. Th exceptions are literals, read only and mapped memory which are directly defined as valid.

Shadow operations

According to Nethercote et al. [NS07b] the shadowing of the state of a program, i.e. the values stored in registers and memory, has to be backed by shadow operations for memory accesses and memory management. Only if

[12] A fourth state is used as indicator for the *addressability* of a memory location. Therefore the encoding with two bits is optimal. Previously an additional ninth bit was used to store this information.

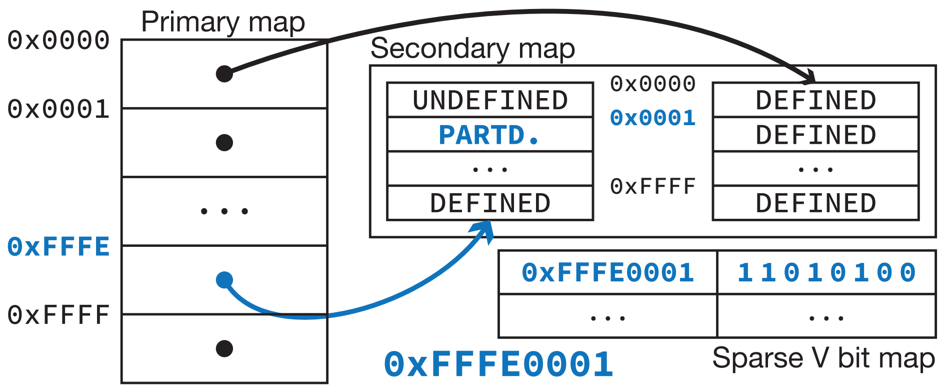


Figure 2.6 – Design of the compressed two-level tables layout of shadow memory. The first level table (PM) is unchanged when compared to the earlier design. The second level table has been changed and stores one of the states "DEFINED", "UNDEFINED" and "PARTDEFINED" for each byte of shadow memory. Accesses to partially defined shadow bytes require an additional lookup in the new sparse V bit table to load the corresponding V bits.

all three categories are covered a reliable[13] detection of undefined values is possible.

While the effects of a less precise handling of shadow registers and memory could be estimated the consequences of only a partial coverage of all operations are hard to confine. For instance, if shadow values were only accurate up to the scale of bytes it would be clear that every operation involving bit operations could lead to wrong detection results. If the shadow values were computed with a conservative approach the risk could even be limited to reporting false positives instead of missing false negatives. In contrast, any instruction, system call or external function has the potential to perform memory operations. As long as the operation is not covered with a corresponding operation on the shadow memory both states will diverge and the result would be false positive warnings as well as false negatives.

The Memcheck plugin is able to achieve a total coverage not only of the program under test itself but also of dynamically linked libraries [SN05]. This is made possible by Valgrind's dynamic instrumentation of machine executable instructions which also includes the dynamic linker. Further, the RISC-like UCode IR of Valgrind makes the shadowing of the complete instruction set easier compared to the more complex x86 instruction set.

For each instruction a trade-off between accuracy of the shadow value and the performance of the shadow operation has to be found. For most

[13] "Reliable" has to be set in contrast to "perfect". A perfect definedness check would be equivalent to solving the Halting problem [NS07b]. Therefore the goal has to be to build a reliable method which emits as less false positives and false negatives as possible.

of the operations the Memcheck plugin puts emphasis on the accuracy and sacrifices efficiency. For example, arithmetic operations shadow the effects of carry chains, though a faster approach would be to invalidate the complete results as soon as one operand is (partially) undefined.

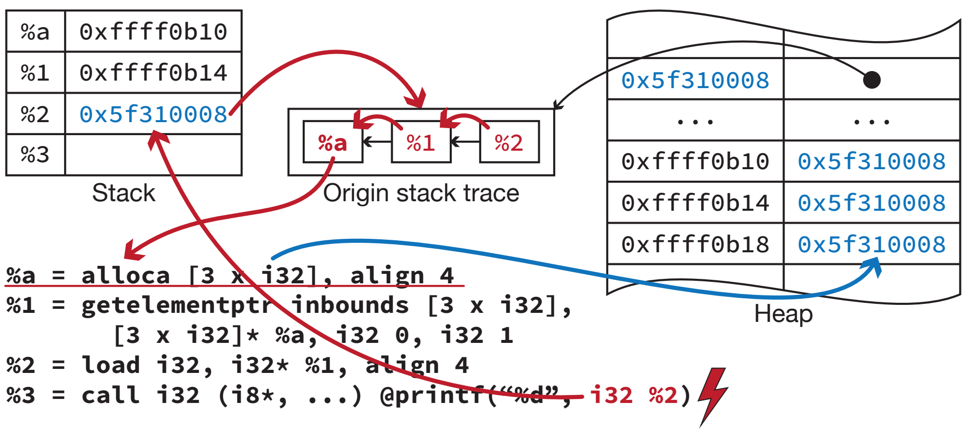
Another design decision is when to emit warning messages about invalid usages of undefined values. The Memcheck plugin *lazily* propagates undefined values through the shadow operations and warnings are only produced at a few critical check points [SN05]. An operation is considered as critical if it alters the observable behaviour of the program. Seward et al. define four distinct groups of such operations. The first two groups include operations that change the control flow like branch instructions and conditional moves. The third groups comprises memory operations where the address operand might be undefined. Finally, system calls form the fourth group. After a warning has been generated the undefined values are explicitly set as defined to prevent chains of warnings resulting from the same problem.

The Memcheck plugin also defines an *eager* reporting strategy where all operands are checked and as soon as either is (partially) undefined a warning is generated. While this makes the origin tracking for undefined values easier it is slower thane the lazy propagation and can lead to more false positives. The latter is due to the fact that many programs legally copy undefined values around without accessing them directly. For instance the compiler can add undefined padding values to structs to enforce alignment rules. Therefore the lazy strategy is favoured most of the time over the eager approach [SN05].

Origin tracking

So far the Memcheck plugin is able to warn about the usage of undefined values. However, since undefined values are propagated through functions the place were the warning is emitted is likely not the root of the undefined value. As propagation chains can be long, undefined values do not contain useful information themselves [Bon+07] and also the stack trace does not provide any helpful information about the origin in half of the cases [Lib+05] it can be hard to track down the root cause of an undefined value.

The Memcheck plugin already keeps origin information for both heap and stack allocations as part of its functionality to signalise incorrect memory accesses. This information can as well be used as root locations for undefined values. To reduce the additional overhead of storing origin information along the propagation chain externally like the V bits Bond et al. [Bon+07] apply a form of *piggy backing*. Since undefined values per definition do not contain valuable information they can be overwritten with information about the origin of the value without affecting the *semantics* of the program.[14] Therefore the plugin instruments all allocations with methods to initialise the otherwise



```
%a = alloca [3 x i32], align 4
%1 = getelementptr inbounds [3 x i32],
       [3 x i32]* %a, i32 0, i32 1
%2 = load i32, i32* %1, align 4
%3 = call i32 (i8*, ...) @printf("%d", i32 %2)
```

Figure 2.7 – Tracking origins via piggy backing. The allocation is instrumented such that the memory is initialised with the origin key specific for this allocation. The key is the address of a data structure which is used by Valgrind to keep track of the allocation. Later the uninitialised array %a is operand in the `load` instruction and the origin key automatically gets propagated into the temporary variable %2. Assuming the `printf` function triggers a system call the undefined value would be detected and the origin could now be tracked back to the initial allocation. The figure is intended to visualise the underlying principles and thus some things are simplified compared to the actual implementation.

uninitialised memory with repeated copies of the origin key (Figure 2.7). The actual definedness state, i.e. the V bits, is of course not affected by this pseudo-initialisations.

In addition to the saved memory space the origin information is propagated for free via the original instructions. But these benefits are not completely without disadvantages. The first issue is that the origin key is the 32 bit pointing to the data structure which keeps track of the origin trace. Therefore it cannot be "piggy backed" in values smaller than four bytes. For those values the origin information is simply not tracked. Second, the origin information is also lost if the actually undefined values are modified through instructions. For instance, if two values are added and at least one of them is undefined the result will not yield any useful information about the origin. The same holds for unaligned loads from allocated memory where only parts of the origin key might be loaded. Lastly, retrieving origin information for conditional branches requires a special treatment since the condition of the branch has been narrowed down to a one bit value. The instructions have to be searched backwards until an origin for a value is found from which the condition has been derived.

[14] It has to be noted that the *behaviour* of the program might well change. Often undefined values consist to a large portion of zeros which are now replaced with meaningful values.

Performance impact

Analysing tools has a high price in terms the runtime. When the Memcheck tool is selected the average slowdown factor is 22. About half of the loss in performance can be attributed to the actual detection of shadow values [NS07a]. The additional overhead of the origin tracking varies widely between different applications but is still negligible compared to the rest of Valgrind. For some programs activating the origin tracking even improved the performance [Bon+07].

In terms of memory efficiency a naive implementation would at least double the required amount of memory since all values have to be shadowed. Some additional space might be necessary to store intermediate results of the shadow operations. The actual implementation of the Memcheck plugin reduces the memory overhead on average to 125 % of the normal memory consumption of the analysed program [NS07a]. For instance the introduced compression of shadow values makes a large portion of the improvements. Because of the piggy backing strategy the origin tracking does not increase the memory consumption.

Besides the extra instructions which have to be executed and a poorer cache locality multi-threaded programs suffer from a serialisation during the analysis. This is necessary to handle loads and stores of the actual data and of the shadow data as atomic operations. Otherwise the memory operations could interleave and the synchronisation of actual data and corresponding shadow values could not be guaranteed [NS07b].

2.2.2 Clang's MemorySanitizer

Like Valgrind's Memcheck plugin the *MemorySanitizer*[15] for the Clang compiler is a dynamic tool to detect invalid usages of undefined values during the runtime of the program. The main difference is that the MemorySanitizer uses static compile time instrumentation (Listing 2.2) instead of the D&R paradigm of the Memcheck plugin. Once the program is compiled with MemorySanitizer support the resulting executable contains all necessary instructions to detect undefined values and does not depend on external tools. On the other hand, it is strictly required to recompile the program under test manually.

Shadow values

The MemorySanitizer uses the same approach of shadowing the data memory with bit-level accuracy like Valgrind's Memcheck plugin but applies a simpler one-to-one mapping between application data and shadow bits [SS15]. This

[15] See `http://clang.llvm.org/docs/MemorySanitizer.html` and `https://code.google.com/p/memory-sanitizer/`, visited on 17/08/2015.

```
%r = alloca i32, align 4
%5 = ptrtoint i32* %r to i64
%6 = and i64 %5, -70368744177665
%7 = inttoptr i64 %6 to i8**
%8 = bitcast i8** %7 to i8*
call void @llvm.memset.p0i8.i64(i8* %8, i8 -1, i64 4, i32 4, i1 false)
```

Listing 2.2 – Code instrumentation through Clang's MemorySanitizer. The first line represents the actual allocation of a 32 bit local variable. All other instructions belong to the code instrumentation. First the address of the shadow value is computed by masking the original address with a fixed constant. Afterwards the four bytes of shadow memory are set to "undefined". Passing "-1" as intialiser value to `memset` has the effect of setting every byte to `0xFF` since the "-1" is converted to an unsigned value.

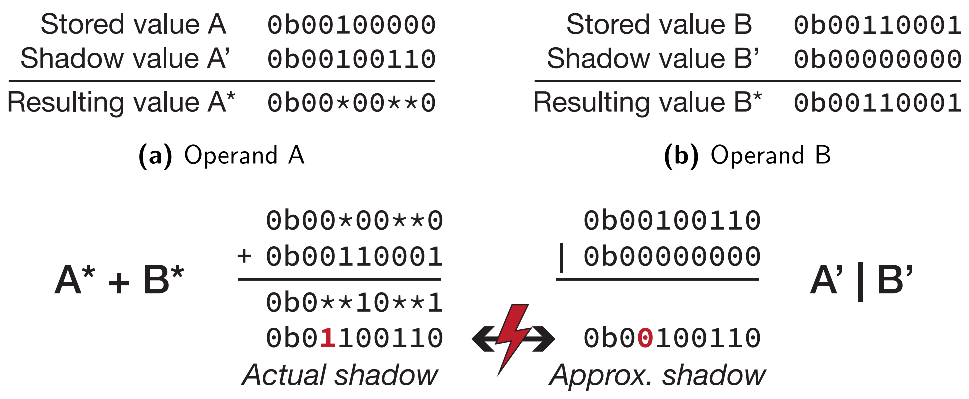
comes at the cost of a higher memory consumption compared to the compacter mapping scheme of Valgrind's Memcheck plugin but simplifies the address computation at the same time. In fact the shadow address is derived by flipping one bit in the original address such that all shadow addresses are projected into a commonly unused address space depending on the actual platform. For instance, on a Linux 64 bit architecture the mask is chosen such that all shadow values are stored beyond the initial 32 TB of the address space.

In contrast to tools operating on a binary representation MemorySanitizer does not have to shadow registers explicitly. The instructions that are injected into the LLVM IR by the MemorySanitizer to propagate shadow values represent the temporary values themselves and are automatically assigned with a unique identifier.

Shadow operations

Most of the shadow propagation operations of the MemorySanitizer are less precise than the corresponding functions in the Memcheck plugin of Valgrind. Moreover, some of the operations even allow false negatives to occur if this has the potential to greatly improve the performance. For instance an addition is approximated through a simple bitwise OR instruction of both operands. This only guarantees that (a) the result is defined if both operands are defined and (b) that the result cannot be valid if either operand is (partially) undefined. However, the actual bits that are undefined after the addition might not be correctly shadowed since the carry propagation is not modelled (Figure 2.8).

Function arguments that are passed by value require that a copy of the shadow data is made. Therefore a separate array of shadow memory is maintained and the callee is told about the address from which the shadow


Stored value A 0b00100000
Shadow value A' 0b00100110
Resulting value A* 0b00*00**0
Stored value B 0b00110001
Shadow value B' 0b00000000
Resulting value B* 0b00110001
A* + B*
0b00*00**0
+ 0b00110001
0b0**10**1
0b01100110
Actual shadow
0b00100110
| 0b00000000
0b00100110
Approx. shadow
A' | B'

(a) Operand A

(b) Operand B

(c) Shadow computation

Figure 2.8 – Imprecise shadow propagation for arithmetic expressions in Clang's MemorySanitizer. This examples demonstrates that the shadow propagation used in MemorySantizer might produce a different shadow value than a precise computation. The shadow value of operand A is at three positions marked as "undefined". Since these bits can have any value – either zero or one – these positions are masked with asterisks in the resulting number. Operand B on the other hand is fully defined and thus does not contain asterisks. When an addition is performed with these two operands ((c), left) certainly all places where either of the operands is masked have to be masked in the result. But further, if a one is added to a masked value it remains unclear if the next position is affected by a carry value or not. Therefore the next position must be masked too. This effect is neglected if the bitwise OR is used to propagate the shadow ((c), right).

values can be retrieved. The same is true for the return value of a function. Another problem is the handling of variadic functions since the caller side and the callee side are handled differently in the LLVM IR. This is again solved through a special memory location for those function arguments but also relies on platform dependent behaviour [SS15].

Further, the MemorySanitizer can hardly achieve a full coverage of all functions if external libraries are involved. This would require that all libraries have been instrumented with the MemorySanitizer instructions. To mitigate the consequences the MemorySanitizer provides wrapper functions for some widely used and hard to compile libraries like “libc” which define the side effects of each function and change the shadow values accordingly in addition to invoking the original function.

Origin tracking

The origin tracking is not implemented with the same piggy backing idea which has been proposed by Bond et al. [Bon+07]. While it helps to save storage capacity and requires nearly no instructions to propagate origin keys the disadvantages are that origin information is lost if the undefined values are altered or unaligned loads are performed and that origin information is only available to values of at least 32 bit.

To avoid these limitations which are coupled to storing the origin information in the undefined values themselves MemorySanitizer maintains a separate memory region to store origins. Every new allocation is assigned a unique 32 bit origin ID which is stored together with the current stack trace. The maximum resolution is one origin ID per four bytes of allocated space.

While this helps to find the location in the program where the undefined value has been allocated it might not always be sufficient to see where the value got undefined. Thus MemorySanitizer provides also an advanced tracking mode where every memory store between allocation and use of the undefined value is stored.

Performance impact

Due to its nature as static compile time instrumentation tool the MemorySanitizer does not suffer from long startup penalties and high performance overheads during the execution [SS15]. Both affect tools like Valgrind which have to dynamically instrument and recompile the binary each time it is executed. Additionally, since the code instrumentation is implemented as *LLVM Pass* it is optimised by later passes during the compilation which improves the efficiency of the injected instructions by approximately 13 % [SS13].

The main advantage over Valgrind’s Memcheck plugin is that multi-threaded programs do not need to be serialised when instrumented with MemorySanitizer [SS15]. Furthermore, no explicit locking is required to

```
T1 store' D
              T2 store' U
              T2 store  X
T1 store  5

T1 load  5
T1 load' U
```

(a) False positive

```
T1 store' D
              T2 store' D
              T2 store  X
T1 store  5

T1 load  5
T1 load' D
```

(b) Store def. shadow

```
T1 store' D
              T2 store' D
              T2 store  5
T1 store  X

T1 load  X
T1 load' D
```

(c) False negative

Listing 2.3 – Handling of atomic memory operations in MemorySanitizer. The names `T1` and `T2` define two concurrent threads. The `store` instruction stores the actual data whereas the `store'` instruction stores the corresponding shadow value. The `load` and `load'` instructions are defined analogous. It is assumed that all operations access the same (shadow) location. The value `X` represents any undefined value, `U` and `D` are symbols for undefined and defined shadow values. The execution in (a) shows a possible interleaving of instructions that leads to a false positive report. To prevent this MemorySanitizer always stores *defined* shadow values for atomic operations (b). As example (c) shows this does not prevent false negatives tough.

handle shadow memory operations since no shared data structure is used to store the shadow values. As long as the original program has been race free – which is required by the C memory model [C11, § 5.1.2.4.25] – the instrumented code is valid as well.

Atomic memory operations on the other hand can only be implemented in a precise way if the original and the shadow operations are protected through a lock. Otherwise it cannot be guaranteed that an operation on the original data is followed by the shadow operation before any other operation accesses the memory location again. To save the overhead of additional locks MemorySanitizer only implements an approximate solution. It prevents the generation of false positives by loading the shadow value after the original value and storing always a *defined* value to the shadow memory *before* the original store is executed [SS15]. This way the shadow of an atomically accessed location can only change from (partially) undefined to defined but not the other way round. The downside of this approach is that it leads to false negatives as a defined shadow value can be assigned to potentially undefined values. The different scenarios are illustrated in Figure 2.3.

All in all, the memory consumption of a program compiled with the MemorySanitizer roughly doubles since every bit of memory is shadowed with a bit representing its definedness. When additionally the origin tracking is activated the memory consumption can increase up to three times compared to the uninstrumented program. In the worst case a 32 bit origin ID is assigned to every four bytes of memory which also results in a one-to-one mapping [SS15]. The runtime of the instrumented program increases about three times without origin tracking and on average five times if origin tracking is activated [SS13]. Nevertheless, compared to the expected slowdown of running the programs under control of Valgrind this is a negligible performance setback.

3 Contributions and design

This chapter provides an overview over all contributions that have been made in the scope of the project. The first section describes an extension of the Csmith tool. This has not been used directly during this project but the change also affects its variant CLsmith which generates the random test cases for the automatic reductions. In Section 3.2 a bug in the Clang compiler is described and the developed solution is explained. The compiler is used as static analyser in the interestingness tests and generates the LLVM IR for the Oclgrind simulator which is utilised as dynamic analysis tool. Besides several bugs which have been identified and reported or patched the diagnostics system of the simulator has been made customisable and an entirely new plugin to detect undefined values during the execution of kernels has been implemented (Section 3.3 and 3.4). The C-Reduce framework (Section 3.5) performs the actual automatic reductions. To increase the compatibility with Windows systems a new CMake build system has been developed and the performance of the framework on Windows has also been improved. Moreover, OpenCL support has been added and some of the transformations have been improved. Finally, a modular test system has been developed in Python to support the workflow from the generation of the random kernels up to the automatic reduction. The details of the concepts and the design are explained in Section 3.6.

3.1 Csmith – timer function for the ARM architecture

The CLsmith tool – and thus also parts of the Csmith tool – has been used as utility to generate the OpenCL test cases for the automatic reduction. During the experiments on the Chromebook 2 (see Section A.2.6) it has been noticed that the function to generate timestamps as seed for the random number generator of Csmith was not compatible with the ARM architecture. To

```
%5 = shufflevector <2 x i32> %3, <2 x i32> undef, <2 x i64> <i64 0, i64 undef>
LLVM ERROR: Broken function found, compilation aborted!
```

(a) Oclgrind crash report

```
Assertion failed: (V[i]->getType() == T->getElementType() && "Initializer for vector
 ↪  element doesn’t match vector element type!"), function ConstantVector, file
 ↪ llvm/lib/IR/Constants.cpp, line 1031.
Stack dump:
1.  <eof> parser at end of file
2.  Bug_12_1.cl:3:6: LLVM IR generation of declaration ’test1’
3.  Bug_12_1.cl:3:6: Generating code for declaration ’test1’
clang-3.7: error: unable to execute command: Illegal instruction: 4
clang-3.7: error: clang frontend command failed due to signal (use -v to see
 ↪ invocation)
```

(b) LLVM assertion violation

Listing 3.2 – Crash outputs triggered by an invalid `shufflevector` operand. The `shufflevector` instruction does accept `undef` input values but the shuffle mask (third operand) has to be of 32 bit integral type. The assertion is violated since the shuffle mask is wrongly created with elements of different types.

prevent multiple runs from having the same seed the timer value is required to have a high accuracy and hence the *Time Stamp Counter* register is read with the `rdtsc` assembler instruction. As the instruction is not supported on ARM architectures it had to be replaced (Listing 3.1).

The alternative has been taken from the codebase of the *gperftools*.[16] On modern ARM architectures the user mode performance counter can be used as replacement. Therefore it has to be configured as readable and activated to actually count the CPU cycles. As a fallback for older systems or if one of the conditions is not satisfied a timestamp is obtained from the `gettimeofday` function.

3.2 Clang – invalid shufflevector operands

This bug[17] triggered a crash of Oclgrind while it was used for testing as an OpenCL implementation in its own right. However, the emitted warning “Invalid shufflevector operands!” and the additional output shown in Listing 3.2a indicated that it is actually a bug in the code generation of the Clang compiler.

The test case in Listing 3.3 is the result of reducing the original program first with C-Reduce and applying some final tweaks manually afterwards. The wrong code generation can be reproduced when Clang 3.6 is invoked

[16] See `https://code.google.com/p/gperftools/`, visited on 10/08/2015.
[17] See `https://llvm.org/bugs/show_bug.cgi?id=23800` and `http://reviews.llvm.org/D10838`, visited on 03/08/2015.

```
diff --git a/src/platform.cpp b/src/platform.cpp
index be456f7..aecaa3d 100644
--- a/src/platform.cpp
+++ b/src/platform.cpp
@@ -70,6 +70,28 @@ static unsigned __int64 read_time(void) {
         }
         return (h << 32) + l ;
 }
+#elif __ARM_ARCH_ISA_ARM == 1
+// From: https://gperftools.googlecode.com/git-history/100
 ↪ c38c1a225446c1bbeeaac117902d0fbebfefe/src/base/cycleclock.h
+static unsigned long long read_time(void) {
+#if __ARM_ARCH >= 6  // V6 is the earliest arch that has a standard cyclecount
+    unsigned int pmccntr;
+    unsigned int pmuseren;
+    unsigned int pmcntenset;
+    // Read the user mode perf monitor counter access permissions.
+    asm("mrc p15, 0, %0, c9, c14, 0" : "=r" (pmuseren));
+    if (pmuseren & 1) {  // Allows reading perfmon counters for user mode code.
+      asm("mrc p15, 0, %0, c9, c12, 1" : "=r" (pmcntenset));
+      if (pmcntenset & 0x80000000ul) {  // Is it counting?
+        asm("mrc p15, 0, %0, c9, c13, 0" : "=r" (pmccntr));
+        // The counter is set up to count every 64th cycle
+        return static_cast<unsigned long long>(pmccntr) * 64;  // Should optimize
 ↪ to << 6
+      }
+    }
+#endif
+    struct timeval tv;
+    gettimeofday(&tv, 0);
+    return static_cast<unsigned long long>(tv.tv_sec) * 1000000 + tv.tv_usec;
+}
 #else
 static long long read_time(void) {
         long long l;
```

Listing 3.1 – Alternative to `rdtsc` on ARM. The `rdtsc` instruction which Csmith uses to generate a seed for the random number generator based on a timestamp is not supported on ARM architectures. Instead the user mode performance counter is accessed or `gettimeofday` is invoked.

```
typedef unsigned int uint2 __attribute((ext_vector_type(2)));

void test1(void) {
    (uint2)(((uint2)0).s0, 0);
}
```

(a) Reduced test case

```
store <2 x i32> zeroinitializer, <2 x i32>* %2
%3 = load <2 x i32>* %2
%4 = extractelement <2 x i32> %3, i64 0
%5 = shufflevector <2 x i32> %3, <2 x i32> undef, <2 x i64> <i64 0, i64 undef>
%6 = insertelement <2 x i32> %5, i32 0, i32 1
```

(b) Generated LLVM IR instructions

```
store <2 x i32> zeroinitializer, <2 x i32>* %2
%3 = load <2 x i32>* %2
%4 = extractelement <2 x i32> %3, i64 0
%5 = insertelement <2 x i32> undef, i32 %4, i32 0
%6 = insertelement <2 x i32> %5, i32 0, i32 1
```

(c) Alternative (unoptimised) LLVM IR instructions

Listing 3.3 – Reduced test case which triggers the invalid shuffle operands bug. The `typedef` in (a) is only necessary since Clang does not know about the OpenCL vector types by default. The LLVM IR instructions in (b) are generated by Clang 3.6 and comprise the invalid shuffle mask. The alternative to the generation of the `shufflevector` instruction is demonstrated in (c).

without optimisations on a 64 bit system. If Clang has been compiled with enabled assertions an assertion failure is raised during the compilation of the test case and Clang crashes due to an “Illegal instruction” (Listing 3.2b).

Before the actual bug can be described it has to be explained where the `shufflevector` instructions that finally “reports” the bug originates. The trigger is the element access inside the vector initialiser which causes an optimised control flow in the code generation of Clang. Or, to be more precise, the specific pattern when a vector is initialised with elements from another vector of the same size allows to reuse and shuffle the inner vector instead of constructing a new outer vector, extracting the desired elements from the inner vector and inserting them into the new outer vector.

Both scenarios have in common that initially the inner vector is created and loaded into a temporary variable. Moreover, in both situations the initialisation of the second element in the outer vector is the same. The only possibility is to insert the constant which has been specified in the initialiser expression into the outer vector. The difference between the versions is *how* the outer vector is created.

In the unoptimised case the desired element from the inner vector is first extracted and subsequently inserted into a *new* outer vector. In con-

```
llvm::ExtractElementInst *EI = cast<llvm::ExtractElementInst>(Init);              1
                                                                                  2
if (EI->getVectorOperandType()->getNumElements() == ResElts) {                    3
  llvm::ConstantInt *C = cast<llvm::ConstantInt>(EI->getIndexOperand());          4
  Value *LHS = nullptr, *RHS = nullptr;                                           5
  if (CurIdx == 0) {                                                              6
    // insert into undef -> shuffle (src, undef)                                  7
    Args.push_back(C);                                                            8
    Args.resize(ResElts, llvm::UndefValue::get(CGF.Int32Ty));                     9
```

Listing 3.4 – Part of the Clang code which introduces the invalid shuffle operands. On 64 bit systems the index operand of the `extractelement` instruction is of 64 bit integral type. The type is preserved when it is converted to `llvm::ConstantInt` and subsequently added to the argument vector but all arguments have to be of 32 bit type.

trast, in the optimised version the outer vector *results* from applying the `shufflevector` instruction to the inner vector. The shuffle mask is chosen carefully such that the desired element from the inner vector is placed correctly to match its position in the outer vector. All other elements in the inner vector are hidden by the `undef` values in the mask and thus do not appear in the outer vector. The `extractelement` instruction which is generated in the optimised case is actually unused in the LLVM IR and is only a side effect of generating the `shufflevector` instruction. It would simply be removed if the compiler optimisations were enabled.

Nevertheless, exactly this `extractelement` instruction is the true cause of the bug. Before the `shufflevector` instruction can be generated the index of the selected element from the inner vector has to be determined. For this purpose the expression `((uint2)0).s0` is converted into an `extractelement` instruction which provides access to the index (Listing 3.4, lines 1 and 4). According to the *LLVM Language Reference Manual* [LLVM15, '`extractelement`' Instruction] the index operand is allowed to be of any integer type and the size of the operand seems to be platform depended. On 32 bit architectures it is a four byte integer and on 64 bit an eight byte integer respectively.

The value of the index operand is then directly added to the vector mask (line 8) which preserves the original size of the type. All other values of the mask are filled with the special value `undef` of 32 bit integral type (line 9) to ignore all elements from the inner vector except the selected one. The four byte integral type can be hard-coded for the mask as its element type must be `i32` [LLVM15, '`shufflevector`' Instruction].

The bug does not manifest on 32 bit architectures as the index operand is created of four byte integral type anyway. In contrast, on 64 bit architectures the mismatch between eight byte index operand and four byte undefined values triggers either an assertion violation – if LLVM has been compiled with assertions enabled – or silently produces wrong code with a mask comprising elements of type `i64` since the overall type for the mask is derived from the first element – in this test case the index operand.

Since no one of the Clang developers responded to the submitted bug report the bug has been fixed in the scope of this project and the patch has been submitted. The proposed solution (Listing 3.5) asserts that the index operand can be converted safely into a four byte integral type and performs the conversion before the shuffle mask is created. The patch has been merged into the development branch[18] of Clang 3.8 such that the bug does not appear in fresh builds of Clang. Further, the patch has been backported into the release branch of Clang 3.7.[19] Thus the current stable release[20] is no longer affected by the bug.

3.3 Oclgrind

Over most of the time of this project there has been a good collaboration with James Price the maintainer of Oclgrind. He managed to fix most of the reported issues quickly such that the focus of the development for Oclgrind has been on the new plugin to detect uninitialised values which is described separately in the next section. This section describes two smaller contributions, namely the implementation of a reliably index out-of-bounds detection and the extension of the diagnostics system to allow to select which messages should be emitted.

3.3.1 More precise index out-of-bounds check

During the project a situation was identified in which Oclgrind did not report an out-of-bounds array access. After the issue had been reported a new check for the static array size was added. While this solved the concrete problem, later on it has been noticed that the check could lead to false positives warnings under certain circumstances. Finally, the check for out-of-bounds accesses has been refined in the scope of this project. The pull request[21] has already been accepted and the changes have been merged into the main project. The following sections describe the details of the two involved bugs.

[18] The patch has been pushed as SVN commit `rL243851` and Git commit `2a3c904` respectively.
[19] The revision number in the release branch for version 3.7 is `r243851`. For the Git repository this is commit `94f152a` respectively.
[20] Clang 3.7 has been released on the 01/09/2015.
[21] See `https://github.com/jrprice/Oclgrind/pull/75`, visited on 01/09/2015.

```
--- cfe/trunk/lib/CodeGen/CGExprScalar.cpp
+++ cfe/trunk/lib/CodeGen/CGExprScalar.cpp
@@ -1162,6 +1162,16 @@
   return llvm::ConstantInt::get(I32Ty, Off+MV);
 }

+static llvm::Constant *getAsInt32(llvm::ConstantInt *C, llvm::Type *I32Ty) {
+  if (C->getBitWidth() != 32) {
+      assert(llvm::ConstantInt::isValueValidForType(I32Ty,
+                                                   C->getZExtValue()) &&
+             "Index operand too large for shufflevector mask!");
+      return llvm::ConstantInt::get(I32Ty, C->getZExtValue());
+  }
+  return C;
+}
+
 Value *ScalarExprEmitter::VisitInitListExpr(InitListExpr *E) {
   bool Ignore = TestAndClearIgnoreResultAssign();
   (void)Ignore;
@@ -1212,7 +1222,8 @@
           Value *LHS = nullptr, *RHS = nullptr;
           if (CurIdx == 0) {
             // insert into undef -> shuffle (src, undef)
-            Args.push_back(C);
+            // shufflemask must use an i32
+            Args.push_back(getAsInt32(C, CGF.Int32Ty));
             Args.resize(ResElts, llvm::UndefValue::get(CGF.Int32Ty));

             LHS = EI->getVectorOperand();
```

Listing 3.5 – Proposed patch that solves the invalid code generation. An assertion has been added to make sure that the index operand can safely be converted to a 32 bit integral type. Then the four byte representation is used to as argument to generate the vector mask.

Invalid read masked by padding values

The test case in Listing 3.6 models a situation in which Oclgrind fails to generate a warning for undefined behaviour. While the first access to the array member of the struct is within the valid bounds accessing the second position of the array exceeds the static size of the array. Therefore Oclgrind should emit an "invalid read" warning as it does for the third access to the array.

Here Oclgrind was tricked by padding values that are append to the end of the struct. The member "`f`" of the struct is of type `ulong` and hence enforces 8 byte alignment for the struct. The only other member occupies four bytes such that four bytes padding values are appended to the end of the struct [C99, § 6.7.2.1.15]. Before this issue has been reported Oclgrind only checked whether the load from a given address with a given size was inside of some allocated space. Although the padding values are not initialised they are nevertheless allocated. The second access to the array hits exactly the additional space that has been allocated for the padding values and therefore Oclgrind does not detect an invalid read. In contrast, the third access to the array exceeds also the size of the struct and hence tries to load from an address that has not been allocated. In this case the correct warning message is generated.

A solution to the problem had been added with commit `b7e7a89`. Still Oclgrind could not differentiate between actual data values and padding values but a check for the static size of array members has been implemented and appropriate warning messages are emitted (Listing 3.6c).

Refined out-of-bounds check

While the newly added check solved the initial problem the test case in Listing 3.7 demonstrates a special use case of the `getelementptr` instruction which now results in a false positive warning about an allegedly invalid array access.

Through the *Scalar Replacement of Aggregates* transformation the first struct "`s`" is replaced by a scalar typed array. Further, the first member of the struct has been optimised out as it is not used in the kernel. Hence the array covers only the remaining 39 bytes (7 bytes padding plus 32 bytes for the second member) of the original struct and the first member is never initialised. The entire array is set to zero as the second member of the struct would have been.

The copy assignment following the initialisation in the test case is translated into an explicit store of a "1" at the address of the first member of the struct "`t`" – this store has to be done explicitly since the first member of struct `s` does not exist in the optimised code. The remaining part of the new struct is initialised by copying the values of the replacement array. However,

```
// -g 1 -l 1

struct S0 {
    ulong f;
    uint a[1];
};

__kernel void entry(__global ulong *result) {
    struct S0 c = {1, {2}};
    ulong d;

    d = c.a[0];
    d = c.a[1];
    d = c.a[2];

    result[0] = d;
}
```

(a) Test case

```
Invalid read of size 4 at private memory address 0x3000000000010
    Kernel: entry
    Entity: Global(0,0,0) Local(0,0,0) Group(0,0,0)
      %13 = load i32* %12, align 4, !dbg !26
    At line 14 of input.cl:
      d = c.a[2];
```

(b) Emitted warning message

```
Index (1) exceeds static array size (1)
    Kernel: entry
    Entity: Global(0,0,0) Local(0,0,0) Group(0,0,0)
      %8 = getelementptr inbounds [1 x i32]* %7, i32 0, i64 1, !dbg !25
    At line 13 of input.cl:
      d = c.a[1];
```

(c) New warning message

Listing 3.6 – Invalid array access is masked by padding values. (a) The member "`f`" of the struct enforces 8 byte alignment. The array "a" only occupies four bytes and hence the struct is padded with additional four bytes. The first access to the array is valid and thus generates no warning. Accessing the second position of the array does exceed the static bounds of the array but wrongly generates no warning because the access falls into the padding area of the struct. Finally, accessing the third position of the array emits the correct warning message (b) as the access also exceeds the space of the struct. To solve the issue a check for the static size of the array has been added and an appropriate warning (c) is generated.

```
struct S0 {
    uchar f[1];
    ulong g[4];
};

__kernel void entry(__global ulong *result) {
    struct S0 s = {{1}};
    struct S0 t = s;

    volatile int i = 0;
    *result = t.g[i];
}
```

(a) Valid test case

```
%struct.S0 = type { [1 x i8], [4 x i64] }

define void @entry(i64* nocapture %result) #0 {
  %s.sroa.5 = alloca [39 x i8], align 1
  %t = alloca %struct.S0, align 8
  %i = alloca i64, align 8
  %1 = getelementptr inbounds [39 x i8], [39 x i8]* %s.sroa.5, i64 0, i64 0
  call void @llvm.memset.p0i8.i64(i8* %1, i8 0, i64 39, i32 1, i1 false)
  %2 = getelementptr inbounds %struct.S0, %struct.S0* %t, i64 0, i32 0, i64 0
  store i8 1, i8* %2, align 8
  %3 = getelementptr inbounds %struct.S0, %struct.S0* %t, i64 0, i32 0, i64 1
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %3, i8* %1, i64 39, i32 1, i1 false)
  store volatile i64 0, i64* %i, align 8
  %5 = load volatile i64, i64* %i, align 8
  %6 = getelementptr inbounds %struct.S0, %struct.S0* %t, i64 0, i32 1, i64 %5
  %7 = load i64, i64* %6, align 8, !tbaa !8
  store i64 %7, i64* %result, align 8, !tbaa !8
  ret void
}
```

(b) Generated LLVM IR instructions

Listing 3.7 – Test case producing false positive out-of-bounds warning. The test case in (a) is a valid OpenCL kernel, though Oclgrind emits an index out-of-bounds warning. The warning is caused by the instruction highlighted in red (b). The instruction calculates the pointer to the second member of the struct by moving the pointer to the first member one past its allocated size. The behaviour is well defined by both the *LLVM Language Reference Manual* [LLVM15] and the C99 standard [C99].

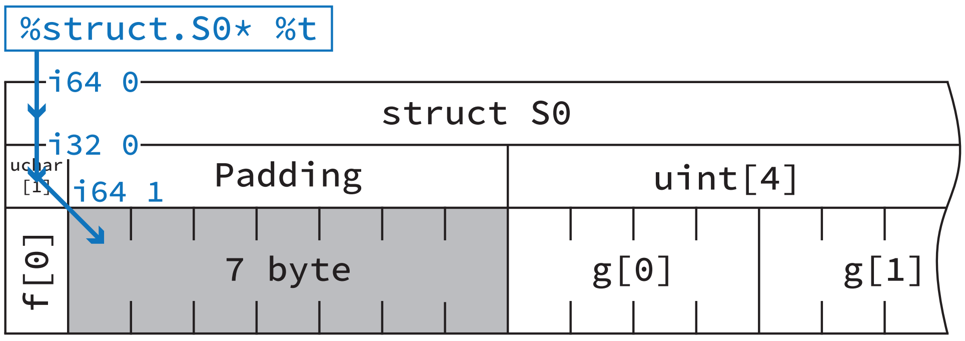


Figure 3.1 – Accessing padding values via the `getelementptr` instruction. Padding values cannot be accessed directly since they are no explicit member of the struct. Instead the first array member is accessed but indexed one past its last element. The constructed address points to the first byte of padding values.

the array cannot be copied to the address pointing to the second member of the struct because the first seven elements of the array belong to the padding values between the two struct members. Moreover, there is no direct way to address padding values and thus the highlighted `getelementptr` instruction indexes the first struct member but accesses it one past its last element (Figure 3.1).

The *LLVM Language Reference Manual* [LLVM15, ‘`getelementptr`’ Instruction] in general does not make any guarantees for the pointer computed with the `getelementptr` instruction. It can even “be outside the object pointed to by the base pointer”. If the `inbounds` keyword is specified the result is indeed undefined if the computed “address is outside the actual underlying allocated object and not the address one-past-the-end”. Yet this does not mean that the static size of an array restricts the address computation since the underlying object is in this case the struct and not the array.

Also the C99 standard does not forbid to construct an address in this way. It explicitly states that even if the resulting address points “one past the last element of the array object, the evaluation shall not produce an overflow” [C99, § 6.5.6.8]. The only restriction is that such an address must not be dereferenced to access the value pointed to. But this is not the case for the generated LLVM IR instructions where the address is used as destination of a `memcpy` instruction.

The consequences from the test case itself and the generated LLVM IR instructions being valid, i.e. being free from undefined behaviour, is that the `getelementptr` instruction cannot be used ultimately to check the validity of array accesses and is also discouraged by the *LLVM Language Reference Manual* [LLVM15, ‘`getelelemtptr`’ Instruction]. While it might work for most of the cases the `getelementptr` instruction operates on a low-level pointer arithmetic concepts which does not always coincides with the concept of static array bounds. Nevertheless, it can be guaranteed that no invalid access will be missed as long as the generated LLVM IR is valid.

Yet, to improve on the false positive warnings a more precise solution has been implemented in the scope of this project. The key to allow arbitrary address computations but to warn on real out-of-bounds conditions at the same time is to perform the validity check only in those situations in which a computed address is used as operand to a load or store instruction. This becomes obvious if the subscript operator is replaced by its definition. According to the C99 standard "`E1[E2]` is identical to `(*((E1)+(E2)))`" [C99, § 6.5.2.1.2]. The addition of the pointer expression `E1` and the offset `E2` can result in undefined behaviour if the resulting pointer is not within the allocated object and also not one past the last element. But the restrictions for the dereference operation are stronger since they require the pointer to be inside the bounds of the allocated object. Since in the LLVM IR it is possible to trace the pointer argument of the load or store instruction back to its origin it is possible to evaluate retroactively whether an already computed pointer violates static array bounds.

The changes to find the origin of the pointer operand of a load or store instruction are shown in Listing 3.8. The body of the `checkAccess` function consists of the validity check which James Price added previously and is shown in Listing C.1.

3.3.2 Customisable diagnostic messages

Depending on the behaviour which is considered "interesting" it can be useful to ignore specific warning or error messages. Especially, if some warnings are known to be false positives for the given test case it is important to disable them to preform the automatic reduction.

Filtering diagnostic messages

Instead of extending Oclgrind it would have been possible to just ignore certain messages in the interestingness tests. But this approach has two major disadvantages. First, all messages would have to be logged regardless whether they are meaningful or would be ignored anyway. This increases the runtime of Oclgrind as well as the time to parse the log file for important information. Second, Oclgrind has a hard limit for the number of messages that get reported. If this limit is exceeded Oclgrind keeps running but no further output is generated. This bears the risk that important messages are hidden by (probably known) false positives if these flood the limit.

After these considerations it seemed best to make the existing diagnosis system of Oclgrind customisable similar to the mechanism compilers offer. The flag `-Wall` enables all diagnostic output and individual message types

```
void MemCheck::instructionExecuted(const WorkItem *workItem,
                                   const llvm::Instruction *instruction,
                                   const TypedValue& result)
{
    // Check static array bounds if load or store is executed
    const llvm::Value *PtrOp = nullptr;

    if(const llvm::LoadInst *LI = llvm::dyn_cast<llvm::LoadInst>(instruction))
    {
        PtrOp = LI->getPointerOperand();
    }
    else if(const llvm::StoreInst *SI = llvm::dyn_cast<llvm::StoreInst>(instruction))
    {
        PtrOp = SI->getPointerOperand();
    }
    else
    {
        return;
    }

    if(auto GEPI = llvm::dyn_cast<llvm::GetElementPtrInst>(PtrOp))
    {
        checkArrayAccess(workItem, GEPI);
    }
}
```

Listing 3.8 – Changes to validate array accesses for load and store instructions. For load and store instruction the pointer operand is extracted from the instruction. If the pointer has been constructed through a `getelementptr` instruction it must be evaluated if the construction of the pointer violated static array bounds.

```
enum MessageType
{
  DEBUG,
  INFO,
  WARNING,
  ERROR,
};
```

(a) Original categories

```
enum MessageType
{
  // Base types
  OCLGRIND_DEBUG = 0,
  OCLGRIND_INFO = 1,
  OCLGRIND_WARNING = 2,
  OCLGRIND_ERROR = 3,
  // Special warning types
  OCLGRIND_WARNING_UNINITIALIZED = OCLGRIND_WARNING + 4,
  // Special error types
  OCLGRIND_ERROR_DIVERGENCE = OCLGRIND_ERROR + 4,
  OCLGRIND_ERROR_INVALID_ACCESS = OCLGRIND_ERROR + 8,
  OCLGRIND_ERROR_DATA_RACE = OCLGRIND_ERROR + 12,
  OCLGRIND_ERROR_UNALIGNED = OCLGRIND_ERROR + 16,
  OCLGRIND_ERROR_ARRAY_BOUNDS = OCLGRIND_ERROR + 20,
  OCLGRIND_ERROR_FATAL = OCLGRIND_ERROR + 24,
};
```

(b) Extended categories

Listing 3.9 – Message categories in Oclgrind. Originally Oclgrind featured only a rough categorisation into different kinds of messages (a). This has been extended by more specific types (b) to allow filtering of individual messages.

or at least a single kind of diagnosis can be enabled (`-W...`) or disabled (`-Wno-...`) separately.

Oclgrind featured already a rough categorisation into different kinds of messages based on their estimated severity (Listing 3.9a). This has been split up into a more fine grained system for individual types of messages (Listing 3.9b) which can than be filtered according to the rules given by the user or the default settings. The codes to represent each of the individual messages are chosen carefully such that the kind of a given message can be derived from the two least significant bits. This makes it easier to filter for complete groups or to determine the severity of a given message without having to store it explicitly for every message.

There are two ways to configure which diagnostic messages should be emitted. The fist one is to call `oclgrind` or `oclgrind-kernel` with the respective `-W...` parameters (Table 3.1). The parameters are parsed in a sequential order from left to right and hence later options overwrite earlier ones. The second method is to export an environment variable named `OCLGRIND_DIAGNOSTIC_OPTIONS`. The options have to be specified in the same way as if they had been passed through the command line and also have to be separated with a whitespace character. Again later options dominate earlier ones.

Aborting on errors

As mentioned above, Oclgrind already featured a mechanism to suppress further diagnostic messages once the specified limit has been reached. While this helps to reduce the generated output during the execution of a kernel

Flag	Description
`-Wall`	All diagnostic options
`-Wgeneric`	All error and warning messages
`-Wdebug`	All debug messages
`-Winfo`	All info messages
`-Wuninitialized`	Messages about reads of uninitialised values
`-Wdivergence`	Messages about work-group divergence (barrier, async-copy)
`-Winvalid-access`	Messages about invalid memory accesses
`-Wdata-race`	Messages about data races (read-write, write-write) between work items
`-Wunaligned`	Messages about unaligned memory accesses
`-Warray-bounds`	Messages about array accesses outside the static bounds

Table 3.1 – Diagnostic options for Oclgrind. Except for the "`-Wall`" option all flags have a second form "`-Wno-SPECIFIER`" to selectively disable specific messages. The options are parsed from left to right in the order in which they have been specified such that later options override earlier ones.

it was not possible to stop the execution entirely after an error has been detected. Yet the later one being essential to reduce the runtime of an interestingness test which rejects a kernel on any reported problem.

This feature has been enabled through the definition of the new command line parameter `--stop-errors`. It takes the number of errors after which the execution should be aborted as argument. The default is not to abort the execution on any number of errors to remain compatible to the old behaviour. The number overall number of errors is then increased with every newly detected one and once the limit has been reached Oclgrind print a warning message and terminates with a non-zero exit status.

3.4 Uninitialised value plugin for Oclgrind

Although the plugin is also part of Oclgrind this section has been separated from the other contributions. Since this new plugin is one of the central aspects of the project the design decisions and the implementation details are explained in more detail.

In the beginning of this project Oclgrind featured already a lightweight plugin to detect undefined behaviour due to uninitialised values. The plugin monitored all interactions with the memory system and warned as soon as undefined values have been involved in memory operations. This caused false positives in cases where uninitialised values were just copied around but

```
// -g 1 -l 1

struct S0 {
    ulong f;
    uint g;
};

__kernel void entry(__global ulong *result)
{
    struct S0 s;
    s.f = 1;
    s.g = 2;

    struct S0 t;
    t = s;
}
```

(a) Test case

```
Uninitialized value read from private memory address 0x200000000000c
    Kernel: entry
    Entity: Global(0,0,0) Local(0,0,0) Group(0,0,0)
      call void @llvm.memcpy.p0i8.p0i8.i64(i8* %4, i8* %5, i64 16, i32 8, i1 false)
    At line 17 of input.cl:
      t = s;
```

(b) Emitted warning message

Listing 3.10 – Test case triggering a false positive warning due to copy of padding values. The member "`f`" causes the struct to be 8 byte aligned and hence four bytes of padding values are appended after the member "`g`" which occupies only four bytes itself. In the beginning of the kernel (a) both members of the struct are initialised and yet Oclgrind emits a warning (b) that uninitialised values have been read. The problem are the padding values which are not initialised but have been copied by the generated `memcpy` instruction.

not actually used. For instance, the C99 standard defines padding values to have indeterminate values and specifies that they do not have to be copied together with the struct [C99, § 6.2.6.2.6]. In return this *does not* mean that they *must not* be copied on a struct assignment.

The test case in Listing 3.10 creates a valid instance of a struct and initialises both members. The only undefined values are the four bytes of padding which are appended to the end of the struct to fulfil alignment requirements. The copy assignment of the struct is translated into a `memcpy` instruction which also copies the padding values. At this point the plugin detects the uninitialised padding values and wrongly emits a warning for an uninitialised read. While using the uninitialised padding values directly would be undefined behaviour just copying them around is not.

The issue with this concrete test case has been fixed by James Price in commit `0ba22b1` by explicitly handling the padding values in structs allocated on the stack. However, just a slight modification in which the copy assignment spans across different address spaces brings the issue back. It has been confirmed that the design of the plugin is to simple to handle these cases properly. However, since structs play an essential role in the randomly generated kernels the false positive reports were too frequent to simply ignore them. Thus this new plugin had to be developed. Following, instead of talking about "the plugin" it will be referenced by its working title "*ShadowKeeper*".

3.4.1 Concepts

With Valgrind's Memcheck plugin and Clang's MemorySanitizer (see Section 2.2) there are already two widely used and appreciated implementations of tools to check for undefined values. The ShadowKeeper plugin for Oclgrind is based on a combination of the strategies of these tools but still had to be adapted to the existing infrastructure of Oclgrind and were adapted to match the needs of this project.

Conceptually the ShadowKeeper plugin sits somewhere in the middle between the other two implementations. MemorySanitizer on the one hand is entirely trimmed for efficiency and takes even loss in accuracy for a better performance. Valgrind's Memcheck on the other hand tries to be as precise as possible and optimises only for efficiency as long as it does not have a major impact on the accuracy. The goal for ShadowKeeper is to make the detection of undefined values as complete as possible but not necessarily at a high level of precision. This allows to approximate the propagation of undefined values to improve the performance though the approximations always have to be conservative such they do not decrease the "undefinedness" of any value.

Moreover, the plugin follows the spirit of Oclgrind which is designed with maintainability and modularity in mind although this introduces some overhead in the performance. The object-oriented design as well as the existing plugin system provide an excellent staring point to integrate the new ShadowKeeper plugin into the existing framework. Figure 3.2 provides a rough overview over the different components.

The main class of the plugin acts as a controller which intercepts the action callbacks before and during the kernel execution and creates a context for all shadow data. The design mirrors most the features of the overall architecture of Oclgrind since basically every value has to be doubled with a corresponding shadow value to store its validity. The "ShadowContext" is the collection of all shadow values and the complete shadow memory.

Direct members of the shadow context are the data structures which keep track of the global shadow values and memory which have to be accessible from everywhere in the plugin. Further, the shadow context comprises a thread local workspace that encapsulates all non-global data structures. These includes mappings from real work groups and real work items to

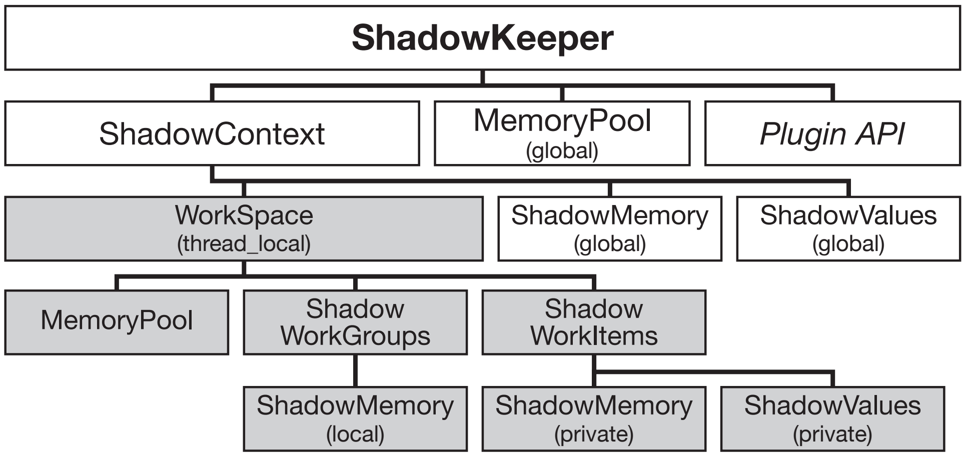


Figure 3.2 – Hierarchical design of the ShadowKeeper plugin. The main class acts as a controller that intercepts the API calls of the plugin system and triggers state changes in the "ShadowContext". The context itself bundles all shadow information. The non-global data structures are encapsulated in a `thread_local` workspace (grey) to prevent data races. Each work group maintains its local shadow memory. Work items store in addition to their private memory also shadow values for the intermediate results during the kernel execution.

shadow work groups and shadow work items respectively as well as the shadow data structures. Making this workspace thread local is part of the (almost) lock free implementation of ShadowKeeper which is explained in the last part of this section.

Finally, every shadow work group maintains the state of the group local shadow memory and each shadow work item stores the validity of intermediate results during the kernel execution and the private shadow memory.

3.4.2 Shadow data

The ShadowKeeper plugin uses the same differentiation between clean and poisoned shadow data as Valgrind's Memcheck and MemorySanitizer. Clean shadows are represented through zero bits and poisoned values through one bits respectively. While in Valgrind's Memcheck constant data values always have a clean shadow the LLVM IR has the special value `undef` for "defined" undefined data. MemorySanitizer provides a flag which changes whether these special values are considered poisoned or not but the ShadowKeeper plugin offers no choice and treats them solely as poisoned. This is also the default for MemorySanitizer and no situation has been discovered where this would have caused problems.

To handle the `undef` constants correctly the function which maps an actual data value to its shadow data includes two explicit tests (Listing 3.11). The function first tries to convert the provided data value into an `undef` value

```
TypedValue ShadowFrame::getValue(const llvm::Value *V) const
{
    /* Parts removed for the sake of presentation */
    else if (llvm::isa<llvm::UndefValue>(V)) {
        return ShadowContext::getPoisonedValue(V);
    }
    /* Parts removed for the sake of presentation */
    else if(const llvm::ConstantVector *VC = llvm::dyn_cast<llvm::ConstantVector>(V))
    {
        TypedValue vecShadow = ShadowContext::getCleanValue(V);
        TypedValue elemShadow;

        for(unsigned i = 0; i < vecShadow.num; ++i)
        {
            elemShadow = getValue(VC->getAggregateElement(i));
            size_t offset = i*vecShadow.size;
            memcpy(vecShadow.data + offset, elemShadow.data, vecShadow.size);
        }

        return vecShadow;
    }
    /* Parts removed for the sake of presentation */
}
```

Listing 3.11 – Special treatment of `undef` data values. Since the LLVM IR has the special `undef` representation of partially initialised data structures not all constants can be mapped to a clean shadow. The plain `undef` value and constants vectors – which can contain `undef` values – are handled separately. The complete function is displayed in Listing C.2.

and return a poisoned shadow if the conversion is successful. Furthermore, constants vectors can potentially contain `undef` values at single positions. Hence, if the provided data value is a constant vector the lookup function is applied recursively to each component of the vector. This way a precise shadow representation is obtained.

The abstract concept of "shadow data" is divided into two categories. First there is the usual shadow *memory* which mirrors the address space of the original kernel. This category is also present in both Valgrind's Memcheck plugin and MemorySanitizer. The second category are actual shadow *values* which represent the validity of global and private variables in the kernel. These values are comparable to the shadows of register values which Valgrind's Memcheck plugin has to maintain. MemorySanitizer on the other hand does explicitly need to store these intermediate values since they are indirectly represented through the statically added shadow operations.

Shadow memory

Like MemorySanitizer, the ShadowKeeper plugin creates a simple one-to-one mapping with bit-level accuracy between the address spaces of the kernel and the shadow address spaces. The only exception is the constant address space which is currently not mapped. The reasonable behind this decision is the fact that for OpenCL kernels all constants have to initialised statically and hence cannot contain undefined values. Therefore it is faster to generate a new clean shadow for each access to constant memory instead of looking it up and it saves space. Each of the other three address spaces (global, local, private) is represented through a separate map which uses the original address of the memory access as key. Using separate maps has the advantage that the lookup of a particular value is likely to be faster and it prevents clashes between the addresses of the different address spaces.

To reduce the memory overhead the implementation uses sparse maps. Nevertheless, memory buffer objects are inserted every time an allocation is performed as the use of a *copy-on-write* mechanism similar to the one in Valgrind's Memcheck plugin turned out to be inefficient in combination with the sparse map. The copy-on-write strategy works well if the address space is represented through a continuous chunk of memory since on a write the values can simply be written to the position which is indexed by the address without the need to actually allocate a buffer object. The problem with a sparse map containing multiple buffer objects is that copy-on-write can lead to a higher fragmentation of the buffer objects as it would be necessary since the store might not access the entire object (Figure 3.3c). The fragmentation can in turn lead to problems with copying entire buffer objects which have been split (Figure 3.3d). Moreover, every write would have to check whether the currently allocated buffer at the target address is large enough to contain the new value which is going to be stored. In favour of a better runtime performance the slightly worse memory efficiency without copy-on-write had to be accepted.

In contrast, deallocations are handled lazily and are only performed on demand instead of together with the deallocation of the data memory. The main reason is that tracking deallocations precisely would introduce a huge amount of extra work since there is no explicitly instruction for deallocations. An object is simply considered as deallocated if its lifetime ends for instance if the scope of the allocation is left. The implemented strategy is to check for every allocation whether there is still shadow memory allocated for the given address and to deallocate it if necessary before the new allocation is performed (Listing C.4). This method is fail-safe since Oclgrind itself ensures the consistency of the data memory and the same addresses are used for the shadow memory.

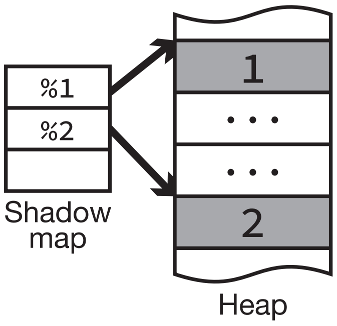
In general all allocated memory is assumed to be undefined and the shadow memory in consequently filled with poisoned values. This assumption might not hold for memory buffers which have been created from initialised memory of the host program and for buffer which are mapped into the host address space. Since ShadowKeeper cannot determine the validity of the data which the host program writes to the memory the best guess – in favour of the

```
struct S0
{
    ulong f;
    ulong g;
};

__kernel void entry()
{
    struct S0 s;

    s.f = 1;
    s.g = 2;

    struct S0 t = s;
}
```

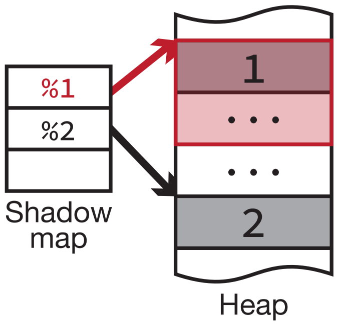
(a) Kernel

```
define void @entry() #0 {
  %s = alloca %struct.S0, align 8
  %t = alloca %struct.S0, align 8
  %1 = getelementptr inbounds %struct.S0,
         %struct.S0* %s, i32 0, i32 0
  store i64 1, i64* %1, align 8
  %2 = getelementptr inbounds %struct.S0,
         %struct.S0* %s, i32 0, i32 1
  store i64 2, i64* %2, align 8
  %3 = bitcast %struct.S0* %t to i8*
  %4 = bitcast %struct.S0* %s to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %3, i8* %4,
    i64 16, i32 8, i1 false)
  ret void
}
```

(b) Generated LLVM IR



(c) High fragmentation



(d) Invalid `memcpy` operation

Figure 3.3 – Problems with copy-on-write in combination with sparse maps. The kernel in (a) creates a struct, initialises both members separately and copies the initialised struct over to a new variable. The corresponding LLVM IR instructions (b) are an allocation of 16 bytes, two stores with each 8 bytes and a `memcpy` instruction with a length of 16 bytes. Since every store has only a size of 8 bytes two buffer objects would be created with a copy-on-write strategy (c). This leads to problems with the `memcpy` operation (d) because the buffer objects need not to be contiguous in memory.

host program – is that it might be defined. Therefore the shadow memory for buffers that are not exclusively mapped for writing and the regions that are written by the host program are filled with clean shadow values.

Shadow values

The intermediate results during the kernel execution are also shadowed with bit-level precision and finally stored in a sparse map. Instead of the address the signature of the original instruction is used as key. Variables in the global scope, e.g. constants and pointers to the local address space, are stored before the actual execution of the kernel is started, exist throughout the entire execution of the kernel and can be accessed form every point in the plugin.

On the other hand, the management of the private shadow values requires a bit more effort than a single map. Here the lifetime of the temporary variables is important since otherwise shadow values from different scopes[22] could interfere with each other if the signatures of the instructions were identical (Listing 3.12). This is a problem which neither Valgrind's Memcheck nor MemorySanitizer experience. Memcheck only has to shadow a finite set of registers independent of the actual instructions that write to them and the shadow values in MemorySanitizer are implicitly scoped through the memory management of the compiler.

The solution which has been implemented in the ShadowKeeper plugin includes the explicit construction of a new “ShadowFrame” (Figure 3.4) each time a function is called and the return to the old frame when the control flow leaves the function. Before the first instruction of the new function is executed the frame is initialised with the shadow values of all arguments of the function. Additionally each frame has an extra slot to remember the signature of its call instruction. This is necessary to link the shadow of the return value of the function to the previous call instruction.

One problem that occurred during the implementation of the shadow values was that the values are represented through dynamically allocated buffers. As long as the values are finally stored in one of the shadow maps this does not cause problems as all elements of the maps can be deallocated when the execution of the kernel terminates. More complicated is the situation when values are created just to compare a given value with a clean value or if it is even unknown at the point of creation if the value will be stored or discarded. To prevent memory leaks all values are therefore obtained from a `MemoryPool` which keeps track of all allocations and performs the deallocation once it is destroyed. This hides the actual memory management from all other parts of the plugin, though it not always be optimal in terms of

[22] To be precise, only different *function* scopes can cause problems. For different blocks which limit the scope of variables inside the same function the compiler has to mangle the variable names anyway to prevent clashes. Therefore within a function no two instructions can have the same signature if they do not affect the same variable.

```
void fn()
{
    ulong u;
    ulong i = u;
}


__kernel void entry()
{
    ulong v = 0;
    ulong i = v;

    fn();
}
```

(a) Kernel

```
define void @fn() #0 {
  %u = alloca i64, align 8
  %i = alloca i64, align 8
  %1 = load i64, i64* %u, align 8
  store i64 %1, i64* %i, align 8
  ret void
}

define void @entry() #0 {
  %v = alloca i64, align 8
  %i = alloca i64, align 8
  store i64 0, i64* %v, align 8
  %1 = load i64, i64* %v, align 8
  store i64 %1, i64* %i, align 8
  call void @fn() #1
  ret void
}
```

(b) Generated LLVM IR

Listing 3.12 – Problems with the scope of private variables. The kernel in (a) first initialises the variable "v" with zero and assigns v afterwards to the variable "i". Then the function "fn" is called and implicitly a new stack frame is generated. Inside the function a *different* variable "i" is set to an undefined value. The variable in the kernel is not affected by this assignment. The translation into the LLVM IR (b) is straightforward. The problem is that both stores to the variables "i" result in identical instructions (red). If there were only a single map for all private shadow values the store in the function "fn" would overwrite the clean shadow of the kernel variable with the poisoned shadow of the function variable.

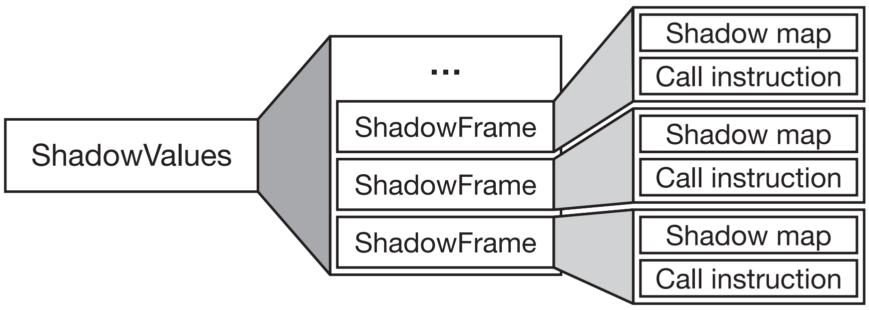


Figure 3.4 – Detailed view on the structure of shadow values. To simulate the different scopes of variables for the shadow values a stack with different frames is maintained. Each function invocation pushes a new frame to the stack and each return from a function pop the frame off. A frame consists of all shadow values in the current scope and an extra slot for the signature of the call instruction of the current function.

the memory which is allocated at any point in time. To reduce the overhead there exists one permanent memory pool for global shadow values and in addition each thread has a thread local memory pool for the shadow values of its current work items. Despite the small overhead in memory consumption this strategy is also used throughout the rest of Oclgrind.

3.4.3 Shadow operations

The ShadowKeeper plugin implements shadow operations in the same way in which it is done for Valgrind's Memcheck and MemorySanitizer. For each instruction on the real data a corresponding function on the shadow data is defined. Most of the function simply propagate shadow values by querying the shadows of the arguments of the real function and combining them in an appropriate way. Only a few functions actually check the definedness of the arguments and emit a warning if necessary.

Oclgrind currently supports 48 LLVM IR instructions and additional 94 built-in functions, though the ShadowKeeper plugin generalises the shadow propagation over a large number of these operations. Instead of computing the result shadow with bit-level precision those generalised functions check the shadow of all operands and if either is (partially) poisoned the entire result is considered as poisoned (Listing 3.13).

First of all this has the great advantage of being faster than an exact computation of the result shadow. Furthermore it cannot make any value less poisoned and does not sacrifice too much of accuracy that it would lead to enormous amounts of false positives.[23] The OpenCL standard does not allow bit fields so the only possibility to intentionally create values which have only single bits undefined is to use bit operations where one operand is undefined. Since the randomly generated kernels do not include this "feature" any undefined bit is actually harmful to the test case such that a warning cannot be a false positive. There is also a small chance that the compiler performs optimisations after which partially initialised values are floating around but this has not been observed during the project.

Warnings about undefined values are only emitted if the values would change the observable behaviour of the program. This is if undefined values are written to memory of a non-private address space and if the control flow depends on undefined values. Further, warnings are generated if undefined values are passed as arguments to external functions since ShadowKeeper cannot check whether the values would exhibit undefined behaviour inside the unknown function. Additionally, though not strictly necessary, undefined values are reported if they are used as index operand in instructions. If the normal propagation had been chosen in these situations the complete

[23] Actually there has been not a single false positive warning since the old plugin has been exchanged with ShadowKeeper.

```
void MemCheckUninitialized::SimpleOr(const WorkItem *workItem,
                                     const llvm::Instruction *I) {
  ShadowValues *shadowValues =
    shadowContext.getShadowWorkItem(workItem)->getValues();

  for(auto OI = I->op_begin(); OI != I->op_end(); ++OI)
  {
    if(!ShadowContext::isCleanValue(shadowContext.getValue(workItem, OI->get())))
    {
      shadowValues->setValue(I, ShadowContext::getPoisonedValue(I));
      return;
    }
  }

  shadowValues->setValue(I, ShadowContext::getCleanValue(I));
}
```

Listing 3.13 – Generalised shadow propagation in ShadowKeeper. Instead of computing the exact shadow value of the result the shadow values for argument of the original instruction are checked. If any of the arguments is (partially) poisoned the entire result value is poisoned.

data structure would have to be marked poisoned on a write access which introduces unnecessary many undefined values. On the other hand, if all values of the index data structure are defined a read access could return a clean value even if the index is undefined. But then the undefined index would remain unnoticed if no warning is generated. The alternative to propagate an undefined value without a warning every time the index is undefined would again introduce unnecessary many poisoned values. Lastly, the addresses of all memory loads and stores a directly checked for their definedness. Without these address checks undefined values would remain unnoticed if for instance a load from a poisoned access luckily hits a valid address and a clean value is loaded. Table 3.2 lists all operations which can generate a warning if undefined values are detected and describes in which situations there warnings are emitted.

The `call` instruction (Listing C.3) requires special treatment since the behaviour of the shadow operations depends on kind of the call. First it is checked if the invoked function is a LLVM intrinsic. Although these functions are externally defined the effects are well known and can thus be handled precisely in terms of the shadow operations. Among the supported intrinsics the `memcpy` operations is special since it may legally copy undefined values without causing undefined behaviour. Nevertheless the definedness of the copied value has to be verified if the destination belongs to a non-private address space.

For this reason a special check has been implemented which is specific to the type of the source value and validates only those parts which must not be undefined to prevent undefined behaviour. Currently the only known

Operation	**Undefined**	**Generated if...**
`async_*_copy`	Index operand	the size or the stride are undefined
	Write	the source value is undefined
`atomic_cmpxchg`	Control flow	the condition is undefined
	Write	the new value is undefined and the exchange is performed
`atomic_*`	Write	the old value or the argument are undefined
`br`	Control flow	the condition is undefined
`call`	Control flow	the call is to an external function and any operand is undefined
`extractelement`	Index operand	the index operand is undefined
`fract`	Write	the floored result is undefined
`frexp`	Write	the exponent is undefined
`insertelement`	Index operand	the index operand is undefined
`lgamma_r`	Write	the sign is undefined
`load`	Address	the address is undefined
`memcpy`	Write	the source is undefined
`memset`	Address	any address is undefined
`modf`	Write	the integral part is undefined
`remquo`	Write	the quotient is undefined
`sincos`	Write	the cosine is undefined
`store`	Write	the stored value is undefined
`switch`	Control flow	any operand is undefined
`wait_group_events`	Control flow	the number of events or any event itself are undefined

Table 3.2 – Operations emitting warnings if undefined values are detected. The two different asynchronous copy functions `async_work_group_copy` and `async_work_group_strided_copy` are combined into one entry in the table. All atomic functions except for `atomic_cmpxchg` are handled in the same way and are not listed separately. The same holds for all load and store instructions. Additionally to the listed warnings every instruction in the list that takes at least on address as argument can emit a "Address" warning if the address is undefined.

situation in which copying undefined values is legal is copying a struct with its the padding values. The check (Listing C.5) is therefore recursively applied to all members of the struct while sparing potential padding areas. If any of the members contains undefined values – other than padding values of potentially nested structs – a warning is emitted.

After the test for intrinsic functions general external function declarations are handled. If the function matches one of the built-in functions known to Oclgrind the specific shadow operation is invoked whereas otherwise the usual check of all arguments is performed. If the function has a non-void return type the call is assigned a clean shadow value. This is not always safe since the function might return an undefined value, though all of its arguments are be defined. But a precise evaluation is not possible and making the return value always poisoned introduces far to many false positives.

Finally, if none of the above criteria is fulfilled and the function is non-variadic[24] a new shadow frame is created to hold the shadow values local to the called function. The frame is initialised with the shadows of the function arguments. Special care has to be taken for pointer values with the `byval` attribute. The shadow memory to which the arguments points has to be duplicated since the original shadow must not be accessed from inside the function. Further, if the function has a non-void return value the call instruction is stored in the dedicated slot in the shadow frame such that the return values can later be assigned to the call instruction. Afterwards the new shadow frame is pushed onto the stack of shadow values.

Each `call` instruction which invokes an internally defined function is accompanied by a `ret` instruction (Listing 3.14). At least the `ret` instruction restores the shadow context of the calling function by popping the top most shadow frame off the stack of shadow values. Moreover, for function with a non-void return type the corresponding `call` instruction is previously looked up in the shadow frame of the called function and the shadow of the return values is assigned to the `call` instruction.

Regarding the generality of the ShadowKeeper plugin one limitation is that currently the image manipulation functions defined in the OpenCL standard are not explicitly supported. They will be treated like any other unknown external function and thus only the arguments are checked for undefined values. Since the kernels which have been generated in the course of this project do not contain image based features the implementation of these function has been postponed to a later version of the plugin.

[24] Variadic functions are not supported at the moment. However, this is not a real limitation of the plugin as also the OpenCL standard prohibits the definition of custom variadic functions and the built-in `printf` function has no side effects that would have to be handled.

```
case llvm::Instruction::Ret:
{
  const llvm::ReturnInst *retInst = ((const llvm::ReturnInst*)instruction);
  const llvm::Value *RetVal = retInst->getReturnValue();

  if(RetVal)
  {
    TypedValue retValShadow =
      shadowContext.getMemoryPool()->clone(shadowContext.getValue(workItem, RetVal));
    const llvm::CallInst *callInst = shadowValues->getCall();
    shadowValues->popFrame();
    shadowValues->setValue(callInst, retValShadow);
  }
  else
  {
    shadowValues->popFrame();
  }

  break;
}
```

Listing 3.14 – Handling of `ret` instructions in the ShadowKepper plugin. If the called function has a non-void return type the shadow of the return value is mapped to the `call` instruction. In the end the current shadow frame is popped from the stack of shadow values to restore the context of the calling function.

3.4.4 Thread safety

Oclgrind in general is able to simulate multiple work items at the same time by running them in different threads. Although the plugins are not required to be thread safe the resulting serialised execution would have rendered the ShadowKeeper plugin useless for large kernels. A major concern in terms of the thread safety are the data structures which are used to store the shadow data.

This is first of all the map which stores all global shadow *values*. Fortunately, the only writing accesses to this map, for instance to initialise constants, are before the actual execution is started. During the execution only reading accesses can happen which do not introduce data races [C++11, § 23.2.2]. Moreover, for the shadow memory of non-local address spaces, although they are accessible from concurrent threads, in general no locking mechanism is needed. Any concurrent access to the same address would correspond to a data race on the actual values since the shadow operations are always synchronised to the real operations. Since data races are forbidden by the OpenCL standard the plugin does not make any guarantees for kernels that exhibit undefined behaviour. The concurrent access to different addresses in the same map is enabled by storing the shadow memory in a buffer separate from the actual map (Listing 3.15) and inserting only a pointer to the buffer into the map. Once again, allocations to the non-private address

```
struct TypedValue
{
    unsigned size;
    unsigned num;
    unsigned char *data;
    /* Parts removed for the sake of the presentation */
};

typedef std::unordered_map<const llvm::Value*, TypedValue> UnorderedTypedValueMap;

UnorderedTypedValueMap m_globalValues;
```

Listing 3.15 – Storage implementation of global shadow memory. Since the map itself must not be changed to prevent data races between concurrent threads only pointers to external buffers are stored in the map. If values are stored into the global memory only the content of the buffers changes but not the address of the buffer and hence not the map. Allocation of buffers is only performed before the kernel is executed and is thus not exposed to multiple threads.

space are performed before the actual execution starts and hence during the runtime the maps are never changed and no data races can occur [C++11, § 23.2.2].

All maps that contain non-global shadow data have been declared as `thread_local` (Listing 3.16) which makes the containers scoped to the executing thread and thus directly prevents any sort of data race without the need for explicit locking. Since non-global shadow data need to be visible to other threads this is introduces no further problems. The usage of `thread_local` however is quite tricky. The Clang compiler does not support `thread_local` variables with non-trivial destructors at all[25] and GCC punishes every use of `thread_local` variables with non-trivial destructors with performance overheads.[26] The C++11 standard has no such restrictions but the implementation of *dynamic* `thread_local` variables is difficult since the initialisation has to be performed during the runtime *before* the first usage and cannot be done statically during the compilation. Mainly to remain compatible to the Clang compiler the use of `thread_local` in the plugin – and throughout Oclgrind in general – is therefore restricted to static variables with constant initialisers. This requires to wrap pointers to the non-global data structures in a trivial struct which can be statically initialised with null pointers (Listing 3.17). The actual initialisation as well as the dynamic destruction is performed manually at the begin and respectively at the end of each work item.

[25] See `http://clang.llvm.org/docs/AttributeReference.html#thread`, visited on 23/08/2015.

[26] See `https://gcc.gnu.org/gcc-4.8/changes.html#cxx`, visited on 23/08/2015.

```
struct WorkSpace
{
    ShadowItemMap *workItems;
    ShadowGroupMap *workGroups;
    MemoryPool *memoryPool;
    unsigned poolUsers;
};

static THREAD_LOCAL WorkSpace m_workSpace;
```

Listing 3.16 – Declaration of `thread_local` workspaces. To prevent data races among the non-global data structures they encapsulated in a `thread_local` workspace. Hence every thread has its own instance of the variables and does not interfere with other threads.

```
THREAD_LOCAL ShadowContext::WorkSpace ShadowContext::m_workSpace =
  {NULL, NULL, NULL, 0};
```

Listing 3.17 – Static dummy initialisation of the `thread_local` workspace. Since dynamic `thread_local` variables are not supported or impractical a trivial workspace is created which can be statically initialised with null pointers (mostly). The real initialisation is performed manually for each work item.

The only exception from the lock free implementation are atomic operations which affect the global data structures. Here it is not responsibility of the kernel to prevent data races but the implementation of the atomic operations must handled concurrent accesses without introducing undefined behaviour. Every atomic shadow operations which writes to the global address space acquires a lock prior to reading the old shadow value and releases it after writing the new shadow value (Listing C.6). Additionally, to reduce the contention multiple locks are provided such that each operations locks only a part of the address space. Atomic operations in different parts do not block each other. For this purpose the part of the address is masked with the number of available mutexes which yields the index into the array of mutexes (Listing 3.18).

3.5 C-Reduce

This section describes the modifications which have been made to the existing C-Reduce implementation in the scope of this project. The contributions include patches fixing bugs in the transformations of clang_delta as well as extensions of the clang_delta tool to handle OpenCL kernels as input. Furthermore, the C-Reduce algorithms on Windows systems has been re-

```
// Multiple mutexes to mitigate risk of unnecessary synchronisation in atomics
#define NUM_ATOMIC_MUTEXES 64 // Must be power of two
static std::mutex atomicShadowMutex[NUM_ATOMIC_MUTEXES];
#define ATOMIC_MUTEX(offset) \
  atomicShadowMutex[(((offset)>>2) & (NUM_ATOMIC_MUTEXES-1))]
```

Listing 3.18 – Implementation of multiple locks to reduce contention. Since accesses to different addresses to not result in data races it is not necessary to make them mutually exclusive. On the other hand providing a mutex for each address is also impractical. As compromise the address space is parted into multiple regions. The addresses inside the same region share a mutex lock but do not affect the addresses in a different region.

```
int foo(ulong *p1);
```

(a) Test case

```
```

(b) Correct transformation

```
;
```

(c) Actual transformation

Listing 3.19 – Example of the wrong handling of function declarations. The transformation fails to remove the trailing semicolon whihc ompletes the function declaration. While this is not a violation of the C99 standard the reduction results is better if the semicolon is remove as well.

designed to increase the performance of the C-Reduce framework on this platform.

3.5.1 Incorrect transformations

In the course of this project a strict compliance with the C99 standard has been necessary to prevent undefined behaviour in OpenCL kernels. This requirement revealed some edge cases in which the transformations produced some non strictly conforming results. Further, some transformation have been optimised to produce smaller test cases. The following sections briefly describe the cause of the problems and outline the solutions which have been worked out during the project.

Transformation: remove-unused-function

The `remove-unused-function` transformation scans the provided source code file for unused functions and removes them subsequently. Originally pure function declarations were not handled correctly (Listing 3.19). In contrast to function definitions declarations are followed by a semicolon which should be removed together with the function declaration. While the failure to remove the semicolon does not introduces undefined behaviour the reduction result is not as small as it could be.

The proposed patch (Listing 3.20) extends the transformation by an additional check for functions with out a body and removes everything up

```
diff --git a/clang_delta/RemoveUnusedFunction.cpp b/clang_delta/Rem[...]Function.cpp
index d264176..bfd83c2 100644
--- a/clang_delta/RemoveUnusedFunction.cpp
+++ b/clang_delta/RemoveUnusedFunction.cpp
@@ -423,8 +423,16 @@ SourceLocation RemoveUnusedFunction::getFunctionLocEnd(
                  SourceLocation LocEnd,
                  const FunctionDecl *FD)
 {
-  if (!FD->getDescribedFunctionTemplate())
-    return LocEnd;
+  if (!FD->getDescribedFunctionTemplate()) {
+    // Remove trailing ; if function has no body
+    if (!FD->hasBody()) {
+      return RewriteHelper->getLocationUntil(LocEnd, ';');
+    }
+    else {
+      return LocEnd;
+    }
+  }
+
   SourceLocation FDLoc = FD->getLocation();
   const char * const FDBuf = SrcManager->getCharacterData(FDLoc);
   const char * LocEndBuf = SrcManager->getCharacterData(LocEnd);
```

Listing 3.20 – Patch that adds handling of pure function declarations. Without the patch the trailing semicolon for function declarations (without a function body) was not removed.

to and including the next semicolon in the source code. As pure functions declarations must be terminated by a semicolon to conform to the C standard this cannot remove too much of the test case. The pull request has been accepted and the patch has been merged into the main project.[27]

Transformation: empty-struct-to-int

The `empty-struct-to-int` transformation removes struct declarations and replaces all usages of the struct type with type `int`. Despite its name the transformation does not only remove structs without any member but also structs with at most one unreferenced member. This is important as the behaviour for structs without any named member is undefined [C99, § 6.7.2.1.7].[28] Thus test cases with empty structs would be rejected by the interestingness test and the struct declaration could only be removed by the simple delta-reduction steps.

Completely empty structs cannot have an initialiser list when a variable of the struct type is defined. In this case the transformation is easy as it is possible to just replace every occurrence of the struct type with an integral

[27] See `https://github.com/csmith-project/creduce/pull/57`, visited on 29/06/2015.
[28] Empty structs are a GNU C extension of the C99 standard. See `https://gcc.gnu.org/onlinedocs/gcc/Empty-Structures.html`, visited on 24/08/2015.

```
struct S0 {
   int a[5];
};

struct S0 s = {{1,2,3,4,5}};
struct S0 as[2] = {{1}, {6,7}};
```

(a) Test case

```
int s = 0;
int as[2] = {0, 0};
```

(b) Correct transform.

```
int s = {{1,2,3,4,5}};
int as[2] = {{1}, {6,7}};
```

(c) Actual transform.

Listing 3.21 – Example of the empty-struct-to-int transformation. Originally the transformation did not modify the initialiser list if a struct type was converted into an integral type. But providing initialisers for non-existent objects is undefined behaviour according to the C99 standard.

type. However, structs with at least one named member can have an initialiser list even if the member itself is unreferenced. Moreover, initialiser lists can be nested for members and variables of array or struct type (Listing 3.21a). The existing implementation of C-Reduce did not consider these cases and left the initialiser list unmodified (Listing 3.21c). But again such a transformation result would have to be rejected in the interestingness test since according to the C99 standard "[n]o initializer shall attempt to provide a value for an object not contained within the entity being initialized" [C99, § 6.7.8.2].

The solution is similar to the principle of the remove-unused-struct-field transformation. This patch adds a new tree-like data structure which keeps track of all usages of the struct that is going to be rewritten. During the parsing of the AST the tree is filled such that in the end every occurrence of the struct type under change is represented as a leaf node (Figure 3.5). The inner nodes are structs that have itself a member of the struct type under change. This information is essential as also the initialiser lists of these "surrounding" structs have to be altered. Once all occurrences have been determined the specific initialiser lists are simply replaced with a zero value.

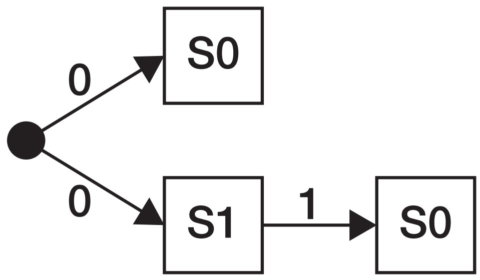
A pull request has been created[29] to include the changes into the C-Reduce mainstream project. It has been accepted and the commit with the hash `49782e7` merges the changes.

During the development of the patch for the struct initialiser lists it has been noticed that the transformation behaves differently if C++ input is assumed during the parsing. Then, in contrast to the documentation, only completely empty structs were replaced with an integral type. Structs with one unreferenced member were not considered like they are in C input mode. While there are special cases in the C++ language like inheritance of structs (Listing 3.22) where the transformation cannot be applied a last unreferenced member should not prevent the transformation.

[29] See `https://github.com/csmith-project/creduce/pull/55`, visited on 30/06/2015.

```
struct S0 {
    int a[3];
};

struct S1 {
    int b;
    struct S0 s0;
}

struct S0 s0 = {1,2,3};
struct S1 s1 = {0, {1,2,3}}
```

(a) Source


$$\begin{tikzcd} & S0 \\ \bullet \arrow[ur, "0"] \arrow[r, "0"'] & S1 \arrow[r, "1"] & S0 \end{tikzcd}$$

(b) Type usage chain

Figure 3.5 – Example of the collected type usage information. The struct S0 in (a) has only one unreferenced member and can thus be replaced with an integral type. Prior to the actual transformation the complete usage information of the type `struct S0` has to be collected during the AST parsing (b). For the first occurrence the complete initialiser list has to be replaced ("0"). For the second occurrence only the initialiser list of the second member ("1") has to be replaced.

```
struct S0 {
    int a;
};

struct S1 : S0 {
    int b;
};
```

(a) Struct inheritance

```
struct S1 : int {
    int b;
};
```

(b) Invalid transformation

Listing 3.22 – Struct inheritance in C++ can prevent transformations on structs. Although the struct S0 in (a) does only have a single unreferenced member it cannot be replaced with an integral type. Otherwise as shown in (b) the struct S1 would inherit from the type int which is invalid.

```
diff --git a/clang_delta/EmptyStructToInt.cpp b/clang_delta/EmptyStructToInt.cpp
index fbc2efd..5631740 100644
--- a/clang_delta/EmptyStructToInt.cpp
+++ b/clang_delta/EmptyStructToInt.cpp
@@ -343,17 +437,121 @@ bool EmptyStructToInt::isValidRecordDecl(const RecordDecl *RD)

   const DeclContext *Ctx = dyn_cast<DeclContext>(CXXDef);
   TransAssert(Ctx && "Invalid DeclContext!");
+  int count = 0;
   for (DeclContext::decl_iterator I = Ctx->decls_begin(),
        E = Ctx->decls_end(); I != E; ++I) {
-    if (!(*I)->isImplicit())
-      return false;
+    if (!(*I)->isImplicit()) {
+      if ((*I)->isReferenced())
+        return false;
+      ++count;
+    }
   }
+
+  if (count > 1)
+    return false;
+
   return true;
 }
```

Listing 3.23 – Patch that adds non-empty struct support for C++. Instead of marking a record declaration as invalid as soon as any member is found the number of members is counted and only if a referenced member is found or if the record has more than one member it is marked as invalid.

The different behaviour has its root in the `isValidRecordDecl` function of the `EmptyStructToInt.cpp` source file. In this function for record declarations of the C langugage one unreferenced member is allowed explicitly while for the C++ language the record declaration is marked as invalid as soon as any member is found. The patch in Listing 3.23 checks for the concrete number of members and marks the record only as invalid if a referenced member is found or if more than one member is present.

Transformation: remove-unused-field

The `remove-unused-field` transformation scans struct declarations for unreferenced members and removes them one at a time. Although a field might be unreferenced it can still be contained in an initialiser list for the struct. To prevent undefined behaviour due to excess elements in the initialiser list [C99, § 6.7.8.2] the initialisers for the removed field have to be removed from the list as well.

In general the initialiser list can contain a trailing comma after the last element. The exception is that a list without any element[30] must not solely

```
diff --git a/clang_delta/RemoveUnusedStructField.cpp b/clang_delta/Rem[...]Field.cpp
index 01700ad..456744b 100644
--- a/clang_delta/RemoveUnusedStructField.cpp
+++ b/clang_delta/RemoveUnusedStructField.cpp
@@ -291,15 +291,23 @@ void RemoveUnusedStructField::getInitExprs(const Type *Ty,
 void RemoveUnusedStructField::removeOneInitExpr(const Expr *E)
 {
   TransAssert(NumFields && "NumFields cannot be zero!");
-  if (NumFields == 1) {
-    RewriteHelper->replaceExpr(E, "");
-    return;
-  }

   SourceRange ExpRange = E->getSourceRange();
   SourceLocation StartLoc = ExpRange.getBegin();
   SourceLocation EndLoc = ExpRange.getEnd();
-  if (IsFirstField) {
+
+  if (NumFields == 1) {
+    // The last field can optionally have a trailing comma
+    // If this is the only field also the comma has to be removed
+    SourceLocation NewEndLoc =
+      RewriteHelper->getEndLocationUntil(ExpRange, '}');
+    NewEndLoc = NewEndLoc.getLocWithOffset(-1);
+
+    TheRewriter.RemoveText(SourceRange(StartLoc, NewEndLoc));
+
+    return;
+  }
+  else if (IsFirstField) {
     EndLoc = RewriteHelper->getEndLocationUntil(ExpRange, ',');
     TheRewriter.RemoveText(SourceRange(StartLoc, EndLoc));
     return;
```

Listing 3.24 – Patch that fixes the handling of a trailing comma in initialiser lists. If only one initialiser is left in the list everything up to the end of the list is removed. This way it is not necessary to differentiate between lists with a trailing comma and lists without.

consist of a comma. The transformation originally failed to remove the comma in this case and produced semantically incorrect reduction results.

To solve the problem the proposed patch in Listing 3.24 adds an explicit test if only one initialiser is left in the list. If this is the case everything up to the end of the initialiser list is removed instead solely the initialiser. This way it is not necessary to distinguish between lists with a trailing comma and lists without. The patch is included in the pull request which improves the empty-struct-to-int transformation which has been merged into the main C-Reduce repository.

[30] The C99 standard forbids empty initialiser lists. They were added as a GNU C extension. Thus they are not relevant for OpenCL but since C-Reduce is not specific to any standard a solution for the problem has been developed.

```
__kernel void entry()
{
    uint4 i = (uint4)(0);
}
```

(a) Test case

```
__kernel void entry()
{
}
```

(b) Correct transformation

```
}
```

(c) Actual transformation

```
ocl.c:1:6: error: variable has incomplete type 'void'
__kernel void entry()
     ^
ocl.c:1:14: error: expected ';' after top level declarator
__kernel void entry()
             ^
             ;
TranslationUnitDecl 0x10302c2c0 <<invalid sloc>> <invalid sloc>
|- Some unrelated declarations removed
`-VarDecl 0x10302cbd0 <ocl.c:1:1, col:6> col:6 invalid __kernel 'void'
2 errors generated.
```

(d) Generated AST

Listing 3.25 – Transforming OpenCL code with Clang in C mode. The selected transformation is assumed to be `remove-unused-var` and applied to the shown test case (a). After the transformation the variable "`i`" should have been removed from the test case (b). The problem is that Clang does not recognise the OpenCL specific keyword "`__kernel`" and parses it as variable name (d). This "variable" is subsequently unused and therefore most of the test case gets removed (c).

3.5.2 OpenCL support

The C-Reduce framework in general is not tied to any specific programming language. The generic reductions can be applied to OpenCL kernels without the need to be adapted. Also the lexer `clex` designed for languages of the C family works in combination with OpenCL since the latter is an extension of the C99 standard. Tokens from the OpenCL standard which are not part of the C99 language are simply identified as "unknown". Solely the specialised `clang_delta` transformation tool had some problems with the slightly different OpenCL language specifications. Its original version only recognises C and C++ files as input and initialises the used Clang instance correctly. All other file types are simply rejected and the tool aborts.

The first idea was to trick clang_delta by presenting the OpenCL input as plain C input. Since OpenCL is based on the C99 standard is there was quite a chance that the internal Clang instance would be able to parse the OpenCL code even in C mode. However, while clang_delta accepts the file if the extension is altered to `.c` instead of `.cl` the AST generated by Clang is useless. First, it ends after the first unknown token is discovered and worse some unknown tokens are misinterpreted such that the syntax of the program is changed (Listing 3.25d). Consequently the transformation result is useless too (Listing 3.25c).

A straightforward solution to the unrecognised tokens is to preprocess the kernel together with files providing the necessary declarations for all functions and types. Potential sources for such files are the libclc project[31] and the SPIR headers[32] provided by the Khronos Group.

Besides that this has the potential to solve the problem for `clang_delta` without the need to actually modify the tool itself there is also one major disadvantage. The declarations of OpenCL specific types and function in the preprocessed files result in redeclaration errors when the test case is compiled by any other "real" host program, e.g. `cl_launcher`. A workaround would be to make the declarations conditional via `#define` directives but this increases significantly the amount of work that has to be done and care that has to be taken before the automatic reduction steps can be applied.

A better solution seems to be to make use of the OpenCL front-end of the Clang compiler. It can be activated by setting the language options of the compiler to `IK_OpenCL` during the initialisation phase of clang_delta. This enables at least support for the parsing and the compilation of kernels. For instance are the function qualifier `__kernel` and the address space qualifiers `__global`, `__local`, `__private` and `__private` no longer unknown.

Further work has to be done to silence warnings about implicitly declared built-in functions of OpenCL and to make the OpenCL specific types like `ulong` known to the Clang compiler. The libclc project[33] provides header files which contain declarations for all types and functions which are supported by the OpenCL standard. To make these files available to the compiler instance in clang_delta the environment variable `CREDUCE_LIBCLC_INCLUDE_PATH` is evaluated (Listing 3.26). If it contains a valid path it is assumed to point to the include directory of the libclc sources. The actual header file is then included through the command line option `-include` which is provided by the Clang compiler. Additionally, all built-in functions of the compiler have to be disabled as some of them clash with the new OpenCL declarations.

The changes to the `TransformationManager` to activate the OpenCL front-end of the compiler and the include instructions for the libclc header files have been merged into the main C-Reduce project.[34]

3.5.3 Handling of multiple source files

The drawback of including OpenCL header files directly into the instance of the Clang compiler instead of using preprocessed files manifests in the fact that the current version of clang_delta does not work well with files that are not preprocessed and thus not self-contained, e.g. have `#include`

[31] `http://libclc.llvm.org`, visited on 04/06/2015.
[32] `https://raw.githubusercontent.com/KhronosGroup/SPIR-Tools/master/headers/opencl_spir.h`, visited on 04/06/2015.
[33] See `http://libclc.llvm.org`, visited on 24/08/2015.
[34] See `https://github.com/csmith-project/creduce/pull/64`, visited on 01/09/2015.

```
diff --git a/clang_delta/TransformationManager.cpp b/clang_delta/Trans[...]ager.cpp
index e677930..a279daf 100644
--- a/clang_delta/TransformationManager.cpp
+++ b/clang_delta/TransformationManager.cpp
@@ -88,6 +88,32 @@ bool TransformationManager::initializeCompilerInstance(std::
 ↪ string &ErrorMsg)
     // It results an empty AST for the caller.
     Invocation.setLangDefaults(ClangInstance->getLangOpts(), IK_CXX);
   }
+  else if(IK == IK_OpenCL) {
+      //Commandline parameters
+      std::vector<const char*> Args;
+      Args.push_back("-x");
+      Args.push_back("cl");
+      Args.push_back("-Dcl_clang_storage_class_specifiers");
+
+      const char *CLCPath = getenv("CREDUCE_LIBCLC_INCLUDE_PATH");
+
+      ClangInstance->createFileManager();
+
+      if(CLCPath != NULL && ClangInstance->hasFileManager() &&
+         ClangInstance->getFileManager().getDirectory(CLCPath, false) != NULL) {
+          Args.push_back("-I");
+          Args.push_back(CLCPath);
+      }
+
+      Args.push_back("-include");
+      Args.push_back("clc/clc.h");
+      Args.push_back("-fno-builtin");
+
+      CompilerInvocation::CreateFromArgs(Invocation,
+                                         &Args[0], &Args[0] + Args.size(),
+                                         ClangInstance->getDiagnostics());
+      Invocation.setLangDefaults(ClangInstance->getLangOpts(), IK_OpenCL);
+  }
   else {
     ErrorMsg = "Unsupported file type!";
     return false;
```

Listing 3.26 – Including OpenCL related header files in clang_delta. If the source file passed to clang_delta is recognised as OpenCL input the environment variable `CREDUCE_LIBCLC_INCLUDE_PATH` searched to get the path to the libclc include directory. The libclc projects provides files which define types and functions which are specific to OpenCL and thus in general unknown to Clang. The header is then included through the command line option `-include`. Further, the `-fno-builtin` deactivates all built-in functions of Clang which would otherwise clash with the new OpenCL function declarations.

```
# 1 "/home/moritz/msc_project/clsmith/cl_safe_math_macros.h" 1
# 5 "/home/moritz/msc_project/clsmith/CLSmith.h" 2
# 1 "/home/moritz/msc_project/clsmith/safe_math_macros.h" 1
# 6 "/home/moritz/msc_project/clsmith/CLSmith.h" 2
# 19 "/home/moritz/msc_project/clsmith/CLSmith.h"
inline __attribute__((always_inline)) void
transparent_crc_no_string (ulong *crc64_context, ulong val)
{
  *crc64_context += val;
}
```

Listing 3.28 – Origin information added by the preprocessor. The first four lines have been created by the preprocessor and tell the compiler from which files the following function has been included.

statements. Transformations that try to change files other than the primary source file are not handled correctly and result in abortion of clang_delta most of the time. To make the approach of including the header files directly within clang_delta possible either the transformations have to check which file would be changed or clang_delta has to be extended to handle multiple source files correctly.

An extension of clang_delta would require to rewrite large parts of the tool. Moreover, it seems not trivial to block the rewritings to "invalid" files just before they are performed because this could potentially mess up the transformation state.[35] A comprehensive solution to the problem is beyond of the scope of this project and therefore, to prevent at least the crashes a simpler work around has been developed. It adds checks to each of the transformations which skip possible applications of the transformation if they are not within the *main source file* as defined by the Clang compiler (Listing 3.27). This is not perfect as the tests have to be copied into most of the transformations and often a single check is not sufficient such that is has to be repeated at different locations in the transformation. Further, the Clang instance is tricked by origin information for parts of the source code which are for instance added by the preprocessor if include files have been inlined (Listing 3.28). The compiler implementation treats each part as "file" separate from the main file although in reality everything is contained in one file. A simple solution is to remove this information (manually) from preprocessed files before they are reduced.

3.5.4 Native Windows support

The C-Reduce implementation depends mainly on the availability of a Perl interpreter for the simple delta reduction loop and on LLVM and Clang

[35] See `https://github.com/csmith-project/creduce/pull/65`, visited on 24/08/2015.

```
if(!SrcManager->isInMainFile(RD->getLocStart()))
{
    return;
}
```

(a) Simple test

```
// This can be used to process only the main file and those files which are
// (recursively) included from the main file
// It then replaces !SrcManager->isInMainFile(...)
FileID rootFile;
SourceLocation usageLocation = RD->getLocStart();

do
{
    rootFile = SrcManager->getFileID(usageLocation);
    usageLocation = SrcManager->getIncludeLoc(rootFile);
} while(usageLocation.isValid());

if(rootFile != SrcManager->getMainFileID())
    return;
```

(b) Advanced test

Listing 3.27 – Check to prevent changes outside of the main source file. If the location to which the transformation would be applied is outside of the main source file it is excluded from the list of possible instances of the transformation (a). This simple test is not able to differentiate between files that are included from inside the main source file via `#include` directives and files which are included through the command line option `-include`. The test can be extended to the one shown in (b) which has this feature and might be used in the future if the reduction of non-preprocessed files is fully supported.

for the clang_delta reducer. Both of these prerequisites are available for Windows but in practise C-Reduce could only be used on top of environments like *Cygwin*,[36] *MinGW*[37] or *MSYS2*.[38] The main problems have been the lack of a Windows compatible build system, missing support for the `fork` mechanism and minor incompatibilities with Visual C++. The following sections address these issues and present the solutions which have been developed during the project. All changes have been bundled and submitted as a pull request which already has been merged into the main project.[39]

CMake configuration

Originally C-Reduce had to be configured, built and installed via the *GNU build system*.[40] Since Windows has no native support for the GNU tools an (almost) equivalent CMake configuration has been created to simplify the installation of C-Reduce on Windows. Like the existing build system it generates necessary configuration files, prepares the Perl scripts, compiles clang_delta and clex and installs everything into a user-defined location. The current limitation compared to the GNU build system is only the lack of an automated detection whether Flex interprets the variable `yytext` as pointer or as array.

Replacing `fork` with `CreateProcess` on Windows

The C-Reduce implementation uses the Perl function `fork` to run multiple interestingness tests in parallel (see Section 2.1.4). On systems which support the *Fork-Exec* model this is directly translated into the `fork` system call which creates a new process. However, Windows uses a different mechanism to create new processes [Gov10]. Perl still provides the `fork` functionality but emulates the behaviour at interpreter level [Sar95].

The emulation is based on threads in contrast to processes and has therefore some limitations compared to the real Fork-Exec model. Especially the facts that "[t]he outcome of kill on a pseudo-process is unpredictable" and that "using kill[...] on a pseudo-process[...] may typically cause memory leaks" [Sar95] are red flags for the usage of the fork emulation. C-Reduce can create many processes during the reduction and also has to be able to kill them reliably to prevent getting stuck in the reduction loop. With the perlfork emulation a large number of processes has the potential for serious memory leaks and the kill functionality cannot be guaranteed.

[36] See `https://cygwin.com`, visited on 04/08/2015.
[37] See `http://www.mingw.org`, visited on 04/08/2015.
[38] See `http://msys2.github.io`, visited on 04/08/2015.
[39] See `https://github.com/csmith-project/creduce/pull/60`, visited on 04/08/2015.
[40] The *GNU build system* primarily consists of the tools *autoconf* and *automake* to prepare the necessary Makefiles to build the project. See `https://en.wikipedia.org/wiki/GNU_build_system` for further information, visited on 04/08/2015.

```
if ($^O eq "MSWin32") {
    my $cmd = which("cmd.exe");
    my $cmdline = qq{/C "$test" $tmpfn};
    $cmdline .= "␣>␣NUL␣2>&1" unless $VERBOSE;

    my $proc;
    Win32::Process::Create($proc,
                           $cmd,
                           $cmdline,
                           0,
                           Win32::Process::NORMAL_PRIORITY_CLASS() |
                           Win32::Process::CREATE_NEW_PROCESS_GROUP(),
                           ".") || die;
    push @procs, $proc;
    return $proc->GetProcessID();
} else {
```

Listing 3.29 – Process creation for interestingness tests on Windows. The test cannot be launched directly via `CreateProcess` since only binary executables can be started as new processes. Instead a new comand line is launched which then invokes the test. A new process group is created such that all the programs launched by the test itself are terminated once the test gets killed. Like the proper `fork` function the PID of the child process is returned..

Instead of calling `fork` on Windows the native `CreateProcess` function is used to launch the interestingness test as a new process (Listing 3.29). The test cannot be launched directly since only binary executables can be started as new processes. Thus a new command line is launched which then invokes the test. Further, the process is started in a new process group such that all the programs launched by the test itself are terminated once the test gets killed. Like the proper `fork` function the PID of the child process is returned.

As a consequence of the changes to the fork mechanism also the `wait` and `kill` functions had to be handled differently for Windows. The Win32 API does not provide the functionality to wait on *any* child process to finish its execution. So instead when run in a Windows environment a list of active child processes is maintained and the parent always waits for the first process in this list (Listing 3.30). Moreover, the exit code has to be modified to match the one of the actual `wait` function which C-Reduce invokes on non Windows systems.

While the behaviour remains unchanged this adaptation can affect the performance of the test case reduction. After the initial creation of the maximum number of parallel interestingness tests no new test can be started before the first child process in the list has finished. As long as the first test succeeds this has no effect on the performance: all other tests have to be repeated anyway since the transformation belonging to the first test changed

```
if ($^O eq "MSWin32") {
    return -1 if @procs == 0;

    while(1) {
        my $proc = shift @procs;
        $proc->Wait(Win32::Process::INFINITE());

        my $exit_code;
        if ($proc->GetExitCode($exit_code) != Win32::Process::STILL_ACTIVE()) {
            $? = ($exit_code == 0) ? 1 << 8 : 0;
            return $proc->GetProcessID();
        }
        push @procs, $proc;
    }
} else {
```

Listing 3.30 – Waiting for child processes on Windows. The Win32 API defines no method to wait for the termination of *any* child process. Thus on Windows C-Reduce maintains a list of all started interestingness tests and wait for the first one. Further, the exit code is adapted to match the one that the actual `wait` function would return when invoked on non Windows systems..

the source file. Only in the case where the first test fails later but already executed tests would had a positive effect on the performance.

Finally, the `kill` function (Listing 3.31) must use the Windows specific system call `TASKKILL` to ensure that the entire process tree of the interestingness tests gets killed. For this reason a new process group has been created by `CreateProcess` in the `fork` helper function. Invoking the `kill` function with the negative PID to kill the process group instead of the process alone does not work on Windows. After all processes have been killed their temporary folders are removed. This is not specific to the Windows system but had to be executed in a separate loop.

Increasing the portability of the C-Reduce Perl scripts

In addition to the changes of the Fork-Exec mechanism system calls to rename or delete files had to be replaced with Perl functions. While the system calls themselves are not portable across different operating systems the Perl functions abstract for the concrete implementation and are available on "all" systems. For instance, the expression `system "rm␣$tmpfile";` has been replaced with `unlink $tmpfile;`. Furthermore, in some situations require a check on which platform the script is executed. These are references to the special file `/dev/null` which is used to discard the output of a command and the check whether certain executables are available. In the first case the file is reachable under the symbolic name `NUL` and in the second case the extension `.exe` has to be appended to the name of the executable.

```
if($^O eq "MSWin32") {
    while (scalar(@procs) > 0) {
        my $proc = shift @procs;
        # Kill process group
        my $pid = $proc->GetProcessID();
        system "TASKKILL␣/F␣/T␣/PID␣$pid␣>␣NUL␣2>&1";
        $proc->Wait(Win32::Process::INFINITE());
        $num_running--;
    }
    while (scalar(@variants) > 0) {
    my $kidref = shift @variants;
    die unless (scalar(@{$kidref})==5);
    (my $pid, my $newsh, my $tmpdir, my $tmpfn, my $result) = @{$kidref};
    File::Path::remove_tree ($tmpdir, {verbose => 0, safe => 0, error => \my $err});
    }
} else {
```

Listing 3.31 – Killing the process tree of all launched interestingness tests on Windows. The sytem call `TASKKILL` with the option `T` is used to kill the entire process group of the interestingness tests including all programs that have been launched during the test. Afterwards all temporary test directories are removed..

Adaptations to the clex helper tool

The clex tool is used by C-Reduce to remove chunks of the source file based on a lexical analysis. Despite being rather simple it could not be built with Visual C++ as this does not support C99 Variable Length Arrays.[41] The closest alternative would have been to use the `_alloca` function to allocate memory on the stack as the C99 array would have done. However, this function has been deprecated[42] by Microsoft. The replacement function `_malloca` does not guarantee to allocate stack memory and thus requires a call to `_freea`.[43] Without the guarantee to allocate memory on the stack and not having to free the memory there is no real benefit in using one of the Visual C++ specific functions. Therefore the C99 Variable Lengths Arrays have been replaced with a call to the standardised `calloc` function.

The second change to the clex sources is rather a cosmetic one than a definite must. Previously clex returned with a negative exit code if the run was for some reason unsuccessful. On Windows though, Perl's `system` call interprets a negative exit code as failure to start the specified program (Listing 3.32) and emits the warning "Can't spawn "cmd.exe": No such file or directory at [. . .]".[44] Although the warning had no influence on the reduction

[41] See `https://msdn.microsoft.com/en-us/library/zb1574zs.aspx`, visited on 04/08/2015.
[42] See `https://msdn.microsoft.com/en-us/library/wb1s57t5.aspx`, visited on 04/08/2015.
[43] See `https://msdn.microsoft.com/en-us/library/5471dc8s.aspx`, visited on 04/08/2015.

```
status = win32_spawnvp(flag,
           (const char*)(really ? SvPV_nolen(really) : argv[0]),
           (const char* const*)argv);
/* Some code removed for the sake of presentation */
if (status < 0) {
    if (ckWARN(WARN_EXEC))
    Perl_warner(aTHX_ packWARN(WARN_EXEC), "Can't␣spawn␣\"%s\":␣%s",
                argv[0], strerror(errno));
    status = 255 * 256;
}
```

Listing 3.32 – Handling of a negative exit code in Perl on Windows. If the program invoked with the Perl function `system` exits with a negative status code Perl assumes that the program failed to start entirely and emits a warning. The code is an excerpt from the `win32.c` file from the Perl distribution..

itself all negative exit values of the clex tool have been converted into positive equivalents and the Perl scripts have been adapted accordingly to silence the warning.

3.6 Interestingness tests

Structurally the interestingness tests are the same as Yang et al. [Yan+11] use for the reduction of C programs. Lightweight tools are evaluated directly in the beginning whereas slow tools and the actual execution of the test cases are postponed to the end of the test. This strategy saves time if one of the first tools already causes the test to fail. Yet most of the tools used in the tests had to be replaced by alternatives since the original programs do not offer support for OpenCL. After this general overview details about the implementations of the new interestingness tests are presented.

3.6.1 Overview

The interestingness tests used during this project mainly address the issues of undefined values in test cases and data races between concurrent work items or work groups. For this reason each test first performs some hard-coded checks and then invokes the Clang compiler to filter syntactically incorrect kernels out. Further, the warning messages of the compiler and the Clang Static Analyzer are checked for undefined behaviour. Afterwards Oclgrind is used to perform a dynamic analysis and lastly the test cases themselves are executed.

[44] The warning always warns about a failure to spawn `cmd.exe` regardless of the actual program. This indicates that the Perl `system` function first creates a new command line and executes the specified program in this new environment.

All tools are equipped with a fixed time-limit to prevent situations in which the reduction process would get stuck if one of the programs freezes. For the dynamic executions of the kernels the time-limit is moreover important to exit from infinite loops that might have been created through transformations. Since the behaviour of such loops is not defined in the standard anyway it is actually a positive effect that the interestingness tests will fail in these situations. Currently the time-limit is globally set to 300 seconds per tool but might be changed to more fine-grained control on a per tool basis later on.

This section describes the design of the interestingness tests which have been developed during this project, outlines problems which have been discovered and indicates which tools had to be replaced in comparison to the tests used by Yang et al. [Yan+11]. The structure of this section follows thereby the structure of the interestingness tests such that tools described first are also executed first during the tests.

Hard-coded restrictions

Directly in the beginning of the each interestingness test a small number of fix requirements on the test case are checked. The main reason for these tests has been to provide hard-coded checks for problems which could not be detected with the other tools but nevertheless had to be prevented. The situation improved significantly after Oclgrind has been added to the tests to dynamically evaluate the interestingness of a test case and hence most of the hard-coded checks are now obsolete.

Information for cl_launcher The first line of the test case has to be a comment that tells cl_launcher the parameters which were used to generate the kernel as well as the global and local size of the kernel. To prevent that this comment is removed during the reduction a static check matches the first line of the test case with a generalised pattern of the comment.

Definition of `get_linear_global_id` To prevent data races on the final result buffer every work item writes only to the position index by its linear global id. To prevent any change to the definition of the function – which is included in the test case itself after the preprocessing – an explicit had been added to the interestingness test. Only if the definition matched the original one the test case has been considered valid. In principle this check could be removed from the interestingness tests because Oclgrind will warn about data races anyway. But on the other hand this check provides a fast fail option and has not been noticed to affect the reduction performance.

Accessing the result buffer In the same way as the function to compute the linear global id must not be changed to prevent data races it must be ensured that each access to the result buffer is actually performed by using the function as index operand. Again the check is strictly speaking no longer necessary but might improve the runtime performance and seems not to have any other negative effects on the reduction process.

Struct initialisation The randomly generated test cases follow all the same pattern to initialise the structs in the main kernel function (Listing 3.33a). During the reduction the chances are high that some part of the initialisation are removed. Normally, the static analysers should warn about the usage of uninitialised values but due to the complex nature of the test cases this was not reliable and a lot of reductions ended with undefined behaviour. Before Oclgrind has been added to the interestingness tests the hard coded sequence of checks in Listing 3.33b has been used to ensure that the structs are always properly initialised.

False positives of Oclgrind The old plugin to detect uninitialised values of Oclgrind was a common source of false positive warnings. Especially the copy assignment of the struct in the kernel function caused problems as uninitialised padding values are legally copied. As a workaround the number of generated warnings for uninitialised values was compared with the number of work items (Listing 3.34). If either no warning was emitted or the number of warnings was equal to the number of work items it had been assumed that all warnings were due to the copy assignment and the kernel was considered as valid.

Static analysis

Yang et al. [Yan+11] utilised the GCC compiler as lightweight static analyser. Its compilation warnings and errors have been scanned for indicators of undefined behaviour. Since GCC does not provide an OpenCL front-end at the moment the Clang compiler it has been replaced with the Clang compiler in the scope of this project. In general this should even be beneficial since the Clang compiler aims to provide better warnings than GCC.[45] But, empirically it has become clear that especially the analyses related to the use of undefined values stand behind the ones of GCC.

Moreover, even Yang et al. equipped with the higher detection rate of GCC still report undefined behaviour in about 29 % of the programs if solely the compiler warnings and errors are checked. For this reason they use additional static analysers like *Frama-C*.[46] Like GCC this tools is not compatible with OpenCL and thus cannot be used in the framework of this project. The only alternative seems to be the *Clang Static Analyzer*.[47] It

[45] `http://clang.llvm.org/diagnostics.html`, visited on 04/06/2015.
[46] `http://frama-c.com`, visited on 21/08/2015.

```
struct S5 c_2049;
struct S5* p_2048 = &c_2049;
struct S5 c_2050 = {
    /* Initialisers removed for the sake of the presentation */
};
c_2049 = c_2050;
```

(a) Preparation of structs

```
function struct_init()
{
  # Get name of struct that gets assigned
  # It is always the first struct in the entry kernel
  if ! struct_number=$(perl -ne 'END {if($success) {exit 0} else {exit 1}} if(/
↪ struct\s+S\d+\s+c_(\d+)\s*;/) {print $1; $success=1; exit}' $1) 2> /dev/null
  then
    return 0
  fi

  # Check if the pointer to the struct is initialised
  if ! perl -ne 'END {if($success) {exit 0} else {exit 1}} if(/struct\s+S\d+(:?\s*\*\
↪ s+|\s+\*\s*)p_\d+\s*=\s*&\s*c_'${struct_number}'\s*;/) {$success=1; exit}' $1 >
↪ /dev/null 2>&1
  then
    return 1
  fi

  # Check if there is an assigment for that struct
  if ! struct_number=$(perl -ne 'END {if($success) {exit 0} else {exit 1}} if(/c_'${
↪ struct_number}'\s*=\s*c_(\d+)\s*;/) {print $1; $success=1; exit}' $1) 2> /dev/
↪ null
  then
    return 1
  fi

  # Check if assigned value has been initialised
  if ! perl -ne 'END {if($success) {exit 0} else {exit 1}} if(/struct\s+S\d+\s+c_'${
↪ struct_number}'\s*=\s*{/) {$success=1; exit}' $1
  then
    return 1
  fi

  return 0
}
```

(b) Preparation of structs

Listing 3.33 – Static test for initialised structs. The pattern in (a) is common to all automatically generated kernels in this project. Since the static analysers failed to detected situations where the structs remained uninitialised a hard-coded static check has been added to the interestingness tests (b). If no struct at all is left over in the kernel the check is immediately successful. Otherwise the copy assignment of the struct, the initialisation of the pointer and the initialisation of the second struct are controlled.

```
function get_work_items()
{
  if ! work_items=$(perl -ne 'if(/^\/\/.* -g ([0-9]+),([0-9]+),([0-9]+) -l ([0-9]+)
 ↪ ,([0-9]+),([0-9]+)/) {print $1*$2*$3; exit 0} else {exit 1}' $1) 2> /dev/null
  then
    return 1
  fi

  echo ${work_items}
  return 0
}

function check_oclgrind_uninitialised()
{
  count_uninitialised=$(grep -c 'Uninitialized value read from private memory
 ↪ address' $1)

  if [[ ${count_uninitialised} != 0 && ${count_uninitialised} != ${WORK_ITEMS} ]]
  then
    return 1
  fi

  return 0
}
```

Listing 3.34 – Static check to ignore false positive warnings of Oclgrind. The original plugin of Oclgrind to detect the usage of uninitialised values emitted false positive warnings for the copy assignment in the kernel which is executed once for every work item. As a workaround a kernel was only considered invalid if at least warning about the usage of uninitialised values was generated but the number of warnings was not equal to the number of work items. Otherwise the assumption had been that all warnings were due to the false positives.

is implemented as plugin of the Clang compiler and does a more precise analysis of the source code than the Compiler itself. Yet, the Clang Static Analyzer is less powerful than Frama-C.

The warnings of both tools are compared to a predefined blacklist of warnings (Listing 3.35) which are treated as indicators to reject a particular transformed variant. In addition to indicators of undefined behaviour the warnings are also checked for situation which might lead to dynamic failures like assigning an integer value other than zero to a pointer and situations which introduce non-determinism, e.g. comparing a pointer to an integer value other than zero. Especially the latter kind of expressions can hide instances of undefined behaviour since the comparison is non-deterministic and thus a dynamic test might not be able to detect the instance of undefined behaviour directly when it is introduce through a transformation.

Due to the limitations of the static tools some of the reduced test cases still contained undefined behaviour. The three most common causes have been identified as the usage of undefined values and invalid pointers as well as array out-of-bounds accesses. The following paragraphs provide definitions and examples for the different kinds of undefined behaviour.

Usage of undefined values In Appendix J.2 the C99 standard states that the behaviour of a program is undefined if "[t]he value of an object with automatic storage duration is used while it is indeterminate" [C99, § J.2]. Further, the value of an object is indeterminate if it is never initialised or when its lifetime has ended. Unfortunately the Clang compiler itself had the worst detection rate for the usage of undefined values during the project. More often the Clang Static Analyzer is able to find undefined values which were not detected by the compiler itself. Nevertheless, both tools cannot compete with GCC (Listing 3.36).[48] As soon as the context is not trivial and involves multiple levels of indirection undefined behaviour remains hidden.

Deference of null pointers Invalid pointers either point to memory locations that cannot be accessed (any more) or are null pointers. They are often created in various reduction steps by changing or removing assignments, or by explicitly assigning `NULL` to a pointer typed variable. As dereferencing an invalid pointer introduces undefined behaviour [C99, § 6.5.3.2.4] it is vital to have a precise analysis. However, none of the static analysis tools is able to reliably detect uses of invalid pointers. Neither Clang nor GCC report any of the two dereferences of the null pointer in Listing 3.37. The Clang Static Analyzer at least finds the undefined behaviour at location Ⓐ.

[47] `http://clang-analyzer.llvm.org`, visited on 21/08/2015.

[48] For the sake of the comparison the kernels have been transformed into equivalent C programs.

```
if ('warning:␣empty␣struct␣is␣a␣GNU␣extension' not in oC and
  'warning:␣use␣of␣GNU␣empty␣initializer␣extension' not in oC and
  'warning:␣incompatible␣pointer␣to␣integer␣conversion' not in oC and
  'warning:␣incompatible␣integer␣to␣pointer␣conversion' not in oC and
  'warning:␣incompatible␣pointer␣types␣initializing' not in oC and
  'warning:␣comparison␣between␣pointer␣and␣integer' not in oC and
  'warning:␣ordered␣comparison␣between␣pointer␣and␣integer' not in oC and
  'warning:␣ordered␣comparison␣between␣pointer␣and␣zero' not in oC and
  'is␣uninitialized␣when␣used␣within␣its␣own␣initialization␣[-Wuninitialized]' not
↪ in oC and
  'is␣uninitialized␣when␣used␣here␣[-Wuninitialized]' not in oC and
  'may␣be␣uninitialized␣when␣used␣here␣[-Wconditional-uninitialized]' not in oC and
  'warning:␣use␣of␣GNU␣?:␣conditional␣expression␣extension,␣omitting␣middle␣operand'
↪  not in oC and
  'warning:␣control␣may␣reach␣end␣of␣non-void␣function␣[-Wreturn-type]' not in oC
↪ and
  'warning:␣control␣reaches␣end␣of␣non-void␣function␣[-Wreturn-type]' not in oC and
  'warning:␣zero␣size␣arrays␣are␣an␣extension␣[-Wzero-length-array]' not in oC and
  'excess␣elements␣in␣' not in oC and
  'warning:␣address␣of␣stack␣memory␣associated␣with␣local␣variable' not in oC and
  'warning:␣type␣specifier␣missing' not in oC and
  "warning:␣expected␣';'␣at␣end␣of␣declaration␣list" not in oC and
  '␣declaration␣specifier␣[-Wduplicate-decl-specifier]' not in oC):
  return True
```

(a) Blacklist of warnings for the Clang compiler

```
if ('warning:␣Assigned␣value␣is␣garbage␣or␣undefined' not in oCSA and
  'warning:␣Undefined␣or␣garbage␣value␣returned␣to␣caller' not in oCSA and
  'is␣a␣garbage␣value' not in oCSA and
  'warning:␣Dereference␣of␣null␣pointer' not in oCSA and
  'warning:␣Array␣subscript␣is␣undefined' not in oCSA and
  'results␣in␣a␣dereference␣of␣a␣null␣pointer' not in oCSA):
  return True
```

(b) Blacklist of warnings for the Clang Static Analyzer

Listing 3.35 – Blacklist of warnings in the interestingness tests. The warnings in the list are either indicators of undefined behaviour in the test case, check for extensions that are not included in the OpenCL C standard or prevent situations which could lead to non-deterministic behaviour. Hence the interestingness test will fail if any of these warnings is found in the output of the emitting program.

```
struct S0 {
  uint f;
};

__kernel void entry(__global ulong *r)
{
  struct S0 s;
  struct S0 *sp = &s;
  r[0] = sp->f;
}
```

(a) Clang ✗, GCC -O1 ✓, CSA ✓

```
void set(ulong *p1, ulong p2) {*p1 = p2;}

struct S0 {
  int f;
};

__kernel void entry(__global ulong *r)
{
  struct S0 a, b, c = {3};
  a = c;
  ulong d;
  set(&d, b.f);
  r[0] = d;
}
```

(b) Clang ✗, GCC -O1 ✓, CSA ✗

```
struct S0 {
  long *f;
  short g;
};

void fn(struct S0 *p1)
{
  for(int i = 0; i < 9; i++)
    ;
  short *t = &p1->g;
  *t = *p1->f;
}

__kernel void entry(__global ulong *r)
{
  struct S0 a;
  struct S0 *b = &a;
  long c;
  struct S0 d = {&c};
  a = d;
  fn1(b);
  r[0] = b->g;
}
```

(c) Clang ✗, GCC -O2 ✓, CSA ✗

Listing 3.36 – Detecting undefined values in test cases. In (a) the variable `sp` points to the uninitialised struct `s`. Consequently an undefined value is written to the result buffer `r`. In (b) the struct `b` is not initialised but its member "`f`" is passed as argument to the function "`set`". So the undefined value is assigned to the variable `d` and later to the global result buffer `r`. Test case (c) is similar but contains one more level of indirection. The structs are all initialised but the member `f` is a pointer to the uninitialised variable "`d`". Inside the function `fn` the pointer is dereferenced and the undefined value is written to the second member of the struct. Later the value is also assigned to the global result buffer `r`. Clang does not detect the undefined behaviour in any of the cases and the Clang Static Analyzer only in (a). Further interesting is that GCC needs at least the optimisations level `-O2` to warn about the undefined behaviour in (c) whereas in the other two test cases `-O1` is sufficient.

```
struct S0 {
  long *f;
  short g;
};

void fn(struct S0 *p1)
{
  p1->g = *p1->f; // Ⓐ
}

__kernel void entry(__global ulong *r)
{
  struct S0 e = {0};
  struct S0 *ep = &e;
  fn(ep);
  r[0] = *ep->f; // Ⓑ
}
```

Listing 3.37 – Test case containing dereferences of a null pointer. The pointer member of the struct is explicitly initialised to zero and hence is a null pointer. The locations where the pointer is dereferenced are marked with Ⓐ and Ⓑ respectively.

Array out-of-bounds accesses To have a valid array access not only the base pointer has to be valid but also the offset must lie inside the allocated memory for this array [C99, § 6.5.6.8 and § J.2]. Some failed reductions revealed that by simply removing lines from the kernel source code invalid accesses can be created. For instance, Listing 3.38 demonstrates a case where between to successive loops the body of the first loop and the header of the second loop have been removed. As long as the loops use the same name for the loop variable the variant is syntactically correct. All of the static tools failed to reveal the invalid out-of-bounds access although in theory it would be possible to check the static bounds of the array and to compare it to the loop variable.

Dynamic analysis

Finally, to obtain a precise analysis dynamic validation techniques have to be used in the interestingness tests. For this reason Yang et al. [Yan+11] make use of *KCC*[49] and *Valgrind*.[50] KCC instruments the test cases at compile time while Valgrind uses runtime instrumentation. It injects custom library function during the execution of the program under test to keep track of memory allocations, usage and deallocation and can thus be used for a dynamic memory analysis. In contrast to the previous tools Valgrind has an

[49] `https://github.com/kframework/c-semantics`, visited on 21/08/2015.
[50] `http://valgrind.org`, visited on 21/08/2015.

```
struct S4 {
  char g_521[10];
  uint g_731[5];
};

__kernel void
entry(__global ulong *result) {
    struct S4 c = {{0}, {0}};
    ulong d = 0;

    for(int i = 0; i < 10; i++)
        d += c.g_521[i];

    for(int i = 0; i < 5; i++)
        d += c.g_731[i];
}
```

(a) Before transformation

```
struct S4 {
  char g_521[10];
  uint g_731[5];
};

__kernel void
entry(__global ulong *result) {
    struct S4 c = {{0}, {0}};
    ulong d = 0;

    for(int i = 0; i < 10; i++)
        d += c.g_731[i];


}
```

(b) After transformation

Listing 3.38 – Example for an array out-of-bounds access. By removing the grey lines in the kernel (a) the transformed kernel (b) exhibits undefined behaviour due to an array out-of-bounds access. Since both loops use the same name for the loop variable the program remains syntactically correct after the transformation.

equivalent for OpenCL called *Oclgrind* which is used in the interestingness tests in the scope of this project.[51]

The drawback of these dynamic tools is that they actually have to execute the program and the injection of the custom memory management system adds additional overhead. Especially for large and high-dimensional OpenCL kernels this slows down the validation process significantly. Nevertheless they seem to be necessary as the static analysers alone have not been able to provide a reliable detection for undefined behaviour. For instance situations in which the definedness of a value depends on dynamic properties cannot be detected by static tools. Furthermore Oclgrind is used to detect data races during the execution which would make the behaviour undefined. The simulation is stopped after the first warning is reported to keep the runtime of the tests as short as possible.

Finally the kernel is executed with and without optimisations on the actual device – at least to find wrong-code bugs. If the execution results are different between the runs the tests exists with the status code for success. If the results are not different or previously any error occurred the test fails.

3.6.2 Versions

The requirements on the interestingness tests changed a few times during the course of this project. First, only the evaluation on Linux systems was planned and thus the tests had been implemented as shell scripts. Later reduction had to be performed on Windows system too which required to rewrite the

[51] `https://github.com/jrprice/Oclgrind`, visited on 04/06/2015.

tests as batch files. Additionally the tests got more complex over time and evolved into a complete testing framework to support the entire reduction process rather than to provide solely support for the interestingness tests themselves. Therefore the latest version has been implemented in Python which has furthermore the advantage that no differentiation between Linux and Windows is necessary.

The initial shell scripts are not further described as they were equivalent to the examples provided along C-Reduce. The next section describes briefly the implementation of the batch scripts for the Windows as it was not entirely trivial to reproduce the behaviour of the original shell scripts. Finally, the version of the interestingness tests is explained in more detail and usage instructions are provided.

Native Windows interestingness tests

The semantic of the tests remained unchanged between the original shell scripts and the variant on Window. In general porting the behaviour was not an issues and the only difficulty was that batch scripts do not offer support to create custom functions. To keep the modular structure of the tests the function calls in the shell scripts have been simulated through calls to different positions in the batch scripts. These calls exit immediately after the part which was defined as "function body" is executed (Listing 3.39) and return to the original invocation. As a consequence return values had to be replaced with global variables.

Python test framework

The final version of the interestingness test is implemented in Python and is therefore compatible with both Unix and Windows. In addition to the plain interestingness scripts the implementation supports the entire workflow of test case reduction starting at the generation of kernels up to the actual reduction.

The different aspects are implemented as separate classes and methods such that it is no problem to use individual components on their own. Furthermore, platform specific parts of the tests have been abstracted which makes the scripts easily extensible to support new architectures. For instance, a general `OpenCLEnv` environment defines all functionality that is needed to run OpenCL kernels. For new platforms a subclass can be created which provides the concrete functionality.

When only an existing and ready-to-use kernel has to be tested regarding its interestingness the `openCLTest.py` script should be used. In the framework this file contains the actual test logic and has no external dependencies to other parts of the framework. For instance, the following command tests whether the specified kernel triggers a wrong-code bug:

```
call:check_oclgrind_uninitialised out_oclgrind.txt > NUL 2>&1
if %return_oclgrind_uninitialised% neq 0 exit /B 1

:: More content removed for the sake of the presentation

:: Variant is interesting
exit /B 0

:: ----------------------------------------------------------------------------------
:: Function section
:: ----------------------------------------------------------------------------------
:check_oclgrind_uninitialised
    FIND /C "Uninitialized␣value␣read␣from␣private␣memory␣address" < %1 >
 ↪ count_uninitialised.txt 2> NUL
    set /p count_uninitialised=<count_uninitialised.txt

    :: Either there is no warning at all...
    if %count_uninitialised% equ 0 (
        set return_oclgrind_uninitialised=0
        goto:eof
    )

    :: ... or one warning for each work item
    if %count_uninitialised% equ %WORK_ITEMS% (
        set return_oclgrind_uninitialised=0
        goto:eof
    )

    set return_oclgrind_uninitialised=1
goto:eof
```

Listing 3.39 – Extract from the interestingness tests on Windows. Functions are not supoorted in batch scripts and have therefore been replaced with calls to specific positions in the test file. After the "function body" has been executed the jump to `eof` ends the execution of the call and returns to the original invocation of the script. Moreover, return values had to be replaced with global variables.

Parameter	Description
CLSMITH_PATH	Path to the `clsmith` directory containing the CLsmith specific header files
CREDUCE_LIBCLC_INCLUDE_PATH	Path to the directory containing the OpenCL related header files for clang_delta
CREDUCE_TEST_PLATFORM	Id of the OpenCL platform which should be used in the test
CREDUCE_TEST_DEVICE	Id of the OpenCL device which should be used in the test
CREDUCE_TEST_CLLAUNCHER	Path to the cl_clauncher executable which should be used to run the kernels
CREDUCE_TEST_CLANG	Path to the clang executable which should be used in the test

Table 3.3 – Configuration options of the Python test framework. These options are used configure the test framework through environment variables. Some options can also be specified as command line options to the `findMiscompilations.py` script. The last two options are optional if the tools can be located without them. Further, the path to the `clsmith` directory is not required if the kernels are already preprocessed.

```
$ openCLTest.py --test miscompilation CLProg.cl
```

Additional configuration parameters like the test device have to specified via environment variables. All possible options are listed in Table 3.3.

To start with the generation of kernels or to perform the reduction the `findMiscompilaton.py` script[52] can be used. The following paragraphs described common use cases which can be performed with the script. Alternatively, all available options provided by the script can be display by specifying the `--help` option.

Generating kernels To generate `N` kernels the option `--generate N` can be specified. This option is mutually exclusive with the options to read kernels either from specified files or out of a directory. Further, the `--modes` parameter can be used to define which kind of behaviour should be included in the kernels. The combination of multiple modes is supported. The script then automatically invokes the CLsmith tool with the correct options. For instance, to generate 1 000 random kernels containing vector operations the

[52] For historical reasons the name of the script mentions only miscompilations and no other kind of behaviour which could be interesting. Meanwhile the script can be configured via a command line option and features various different tests.

script can be invoked as follows. Thereby the kernels will be stored in the directory `vec_kernels`.

```
$ findMiscompilations.py --generate 1000 \
                         --output ./vec_kernels \
                         --modes vectors
```

Preprocessing To create self-contained kernel files which do not depend on the CLsmith specific header files the script provides a preprocessing operation. In addition to the actual preprocessing the script also removes the origin information in the new file which causes problems in clang_delta. Existing kernels can be preprocessed with the command

```
$ findMiscompilations.py --preprocess --kernels CLProg.c
```

but it can be combined with generation of new kernels, too.

Dimension reduction Kernels which are executed with a large number of work items slow down the reduction process significantly. Though, in most cases running them only in a small number of work items does also expose the wrong behaviour. The dimension of a kernel can automatically be reduced when it is invoked with the `--reduce-dimension` argument. For instance the command

```
$ findMiscompilations.py --preprocessed --reduce-dimension \
                         --kernels CLProg.cl
```

takes the existing kernel `CLProg.cl` and reduces its dimension. The parameter `--preprocessed` is optional and signalises that the kernel is already preprocessed such that the additional headers are not needed. The algorithm to reduce the dimension is at the moment quite simple. It first tries to detect the wrong behaviour with just one work item. If this fails the number of item is steadily increased until this bug reappears. For future use this should be improved to a more sophisticated technique.

Reducing a kernel Finally, the reduction can be started with the `--reduce` argument. Additionally the parameter `--test` has to be passed to specify which tests should be performed, i.e. which kind of behaviour is considered interesting. The following command is represents a typical call of the `findMiscompilations.py` script when kernels have been generated and prepared in advance and only the reduction should be performed.

```
$ findMiscompilations.py --preprocessed --reduce \
                         --kernel-dir ./prep_kernels \
                         --output ./reduced_kernels \
                         --test miscompilation \
                         --log processed.log
```

The optional parameter `--log` defines a file to which the names of all kernels that have been processed are written. The script does not distinguish between successful reductions and failed reductions. This file can later be used as argument to the `--exclude-file` parameter to resume an aborted reduction from the last file that has already been processed.

3.7 Miscellaneous related issues

This section describes further problems which have been encountered but not fixed in the scope of this project. Either the issues still exist or have been fixed by other persons after they have been reported.

3.7.1 CLsmith

Although the CLsmith random program generator in combination with the cl_launcher host program to execute kernels have not been a direct target of testing during this project their application in test case reduction revealed minor problems.

Result buffer in cl_launcher uninitialised

One of the arguments that the cl_launcher application passes to the invoked kernels is a pointer to a global memory location. Each work item stores a hash representing its state after the execution into one place of this result array. The position is determined according to the linear global id of the work item. Further, it is guaranteed and invariant over all automatically generated kernels that each position in the buffer is written to exactly once and never read.

These strong guarantees made it unnecessary to initialise the buffer in the cl_launcher host program. However, the invariant properties no longer hold when the kernel is subject to the automated reduction. The transformations may remove the instructions which write to the result array or could even turn them in to read accesses. In both situations the comparison of the computed results becomes non-deterministic as the results depend on uninitialised values. Thus, for the purpose of test case reduction, a patched version of

```
struct S0 {
   uint f0; volatile short f1; uchar f2; volatile uchar f3;
   uint f4; volatile long f5; char f6; volatile ulong f7; short f8;
};

struct S0 tmp = {{0x65F44572L,0x0762L,0xEDL,0x49L,
                 4294967294UL,0x70E4EAF2CDD3E714L,-4L,3UL,-1L}};
```

Listing 3.40 – Excess elements in scalar initializer. The program generated by CLsmith contains wrong code to initialise a struct. Through the additional pair of curly brackets the compiler is told to initialise the first struct member with all given values. Since a member of type `uint` must not have more than one initialiser the warning is emitted.

cl_launcher is used which initialises the output buffer with zeros. It can be obtained from the "init" branch of a forked version of the GitHub repotory.[53]

Generation of invalid initialiser lists

For some of the programs generated by CLsmith the Clang compiler emits an "Excess elements in scalar initializer" warning. An example – referred to as "Bug_3" – is shown in Listing 3.40. Due to the extra pair of curly braces the compiler uses all given values as initialiser for the first member of the struct [C99, § 6.7.8.18] and default initialises all other members as if they had static storage duration [C99, § 6.7.8.21]. The first member is of type `unsigned int` and hence can only have one initialiser value which is why the warning is emitted. Moreover, providing more initialisers than expected is forbidden by the C99 standard and introduces undefined behaviour [C99, § 6.7.8.2].

The issue is specific to CLsmith, i.e. it does not occur in Csmith, and has been introduced through changes made in commit `be2b1df` (Listing 3.41). For variables of aggregate type the initialiser list generated as output from the `init_value` is already wrapped in curly brackets. Thus originally the initialiser list has been directly append to the output stream after the assignment expression. For some reason this has been changed in the mentioned commit such that now an additional pair of curly brackets is added explicitly in the `Output` function of the `ArrayOp` statement. The change seems to be related to code that has been added to the `MemoryBuffer` class to handle the output of vector expressions.

The problem has been reported but no simple solution could be found so far. As only a few kernels seem to be affected by this issue they have just been ignored in the course of this project.

[53] See `https://github.com/mpflanzer/CLSmith`, visited on 01/09/2015.

```
diff --git a/src/StatementArrayOp.cpp b/src/StatementArrayOp.cpp
index 25e2637..58eccc8 100644
--- a/src/StatementArrayOp.cpp
+++ b/src/StatementArrayOp.cpp
@@ -253,9 +253,9 @@ StatementArrayOp::Output(std::ostream &out, FactMgr* fm, int
 ↪ indent) const
                 if(init_value->term_type == eConstant && array_var->is_aggregate()){
                         output_tab(out, indent+1);
                         array_var->type->Output(out);
-                        out << " tmp = ";
+                        out << " tmp = {";
                         init_value->Output(out);
-                        out << ";";
+                        out << "};";
                         outputln(out);
                         output_tab(out, indent+1);
                         array_var->output_with_indices(out, ctrl_vars);
```

(a) Brackets are added explicitly after the changes are applied

```
diff --git a/src/CLSmith/MemoryBuffer.cpp b/src/CLSmith/MemoryBuffer.cpp
index f911262..b7759d0 100644
--- a/src/CLSmith/MemoryBuffer.cpp
+++ b/src/CLSmith/MemoryBuffer.cpp
@@ -99,7 +102,13 @@ void MemoryBuffer::OutputDef(std::ostream& out, int indent)
 ↪ const {
     ExpressionID(ExpressionID::kLinearLocal).Output(out);
     out << " == 0)" << std::endl;
     std::vector<const Variable *>& ctrl_vars = Variable::get_new_ctrl_vars();
-    output_init(out, init, ctrl_vars, indent + 1);
+    // TODO Vector params need to match the buffer indices, use init.
+    if (type->is_aggregate())
+      StatementArrayOp(NULL, this, ctrl_vars, std::vector<int>({0}),
+      std::vector<int>({1}), Constant::make_int(0))
+      .Output(out, NULL, indent + 1);
+    else
+      output_init(out, init, ctrl_vars, indent + 1);
     return;
   }
   output_tab(out, indent);
```

(b) Related changes to the `MemoryBuffer` output

Listing 3.41 – Changes causing the excess elements code generation. The output for the `init_value` consists already of a complete initialiser list including curly brackets (a). Thus the explicitly added brackets cause the "excess elements" warning. These changes seem to be related to the code which has been added to the `MemoryBuffer` (b) to handle vector expressions.

3.7.2 Clang – no target for triple spir-unknown-unknown

The Clang compiler is used during the installation of Oclgrind to generate pre-compiled header files to improve the simulation performance. The compiler is invoked with the target triples `spir-unknown-unknown` and `spir64-unknown-unknown` respectively. In version 3.6 of the compiler the compilation succeeds without warnings whereas at some point in the development branch for version 3.7 the support for the SPIR target breaks.[54] The compilation then aborts with the error message: "fatal error: error in backend: No available targets are compatible with this triple, see -version for the available targets."

The bug has been reported[55] but remained uncommented. However, commit `49f44ff` seems to fix this problem since the header files can be generated again with any version past this one. Since the bug cannot be reproduced with the 3.7 and the development branches any longer the report has been closed again.

3.7.3 Oclgrind

In addition to the problems with Oclgrind which have been mentioned before three more issues have been found. James Price, the maintainer of Oclgrind, fixed the issues shortly after they had been reported such that no patches had to be developed in the scope of this project.

Incorrect alignment checks

The LLVM `load` and `store` instructions take an optional parameter which specifies the alignment of the given memory address. According to the *LLVM Language Reference Manual* [LLVM15, '`load`' Instruction] overestimation of the alignment leads to undefined behaviour which is why Oclgrind checks whether the specified alignment is correct.

Until commit `e040b05` Oclgrind emitted false positive warnings (Listing 3.42) for allegedly incorrectly aligned pointer operands. The wrong behaviour could only be observed if optimisations had been disabled during compilation of the kernels.

The problem seemed to be that Oclgrind did not handle the alignment of `load` and `store` instruction correctly if the optional alignment parameter had been omitted. In these situations the alignment was wrongly calculated to be "-1" which then mismatched with basically any given address. Commit `da4b06e` fixes the problem by computing the necessary alignment for the given pointer type manually if it is not specified.

[54] For instance the bug can be reproduced with LLVM at commit `371e006` and Clang at commit `92ef7c4`.
[55] See `https://llvm.org/bugs/show_bug.cgi?id=24153`, visited on 03/08/2015.

```
Invalid memory store - source pointer is not aligned to the pointed type
    Kernel: entry
    Entity: Global(0,0,0) Local(0,0,0) Group(0,0,0)
      store i32* %321, i32** %316, !dbg !409
    At line 1106 of CLProg_112.cl:
      { /* block id: 49 */
```

Listing 3.42 – Exemplary false positive warning for allegedly incorrectly aligned pointer operands. The store instruction does not specify the optional alignment parameter causing Oclgrind to operate on an alignment of -1.

```
// -g 1 -l 1

union U0 {
    long f0;
};

void func_28(union U0 u) { u.f0 = 3; }

__kernel void entry(__global ulong *result) {
    union U0 u = {7};
    func_28(u);
    result[0] = u.f0;
}
```

Listing 3.43 – Test case which produced wrong results for functions with `byval` attributes. For this test case final result contained the value “3” altohugh it is only written locally to a copy of the union. The correct result is “7”.

Incorrect handling of the `byval` attribute

Before Oclgrind was considered for inclusion in the interestingness tests, it was used for testing as an OpenCL implementation in its own right. The kernel in Listing 3.43 produced different results when executed with and without optimisations. After reporting the bug it has been fixed with commit `3570fd7`.

With disabled optimisations a wrong final result containing the value “3” has been produced. The correct result must contain the number “7”. The behaviour indicated that somehow the local assignment of the value “3” to a copy of the union must have overwritten the correct result. The bug has been confirmed and attributed to the missing handling of the `byval` attribute of functions. If a pointer argument of a function is marked as “`byval`” an implicit copy of the pointee has to be created such that the callee is not able to modify the actual pointer destination [LLVM15, Parameter Attributes].

The mishandling is also present if the kernel is executed with optimisation but has simply no visible effect. For the compiler the function scoped

```
call void @func_28(%union.U0* byval %u) #3, !dbg !31
    %1 = bitcast %union.U0* %u to i64*, !dbg !30
    store i64 3, i64* %1, align 8, !dbg !30
    ret void, !dbg !30
```

(a) Without optimisations

```
tail call void @func_28(%union.U0* byval @entry.u) #1, !dbg !30
    ret void, !dbg !30
```

(b) With optimisations

Listing 3.44 – Executed LLVM IR instructions for the "`byval`" test case. The indented parts represent the instructions which represent the body of the called function. Without optimisations (a) the pointer to the union is converted into a primitive type and the value "3" is stored to the pointer destination. Since the `byval` attribute had been ignored by Oclgrind the original location of the union has been overwritten. In contrast, the optimisation passes in (b) removed the `store` instruction as it should not have any global side effects. Therefore the `byval` attribute had no visible effect at all and ignoring it did not change the program behaviour.

```
OCLGRIND FATAL ERROR (../src/core/WorkItem.cpp:583)
Insufficient private memory (alloca)
      Kernel: entry
      Entity: Global(0,0,0) Local(0,0,0) Group(0,0,0)
        %24 = alloca i32, align 4
      Debugging information not available.
```

Listing 3.45 – Fatal error in Oclgrind as the address space is exceeded. The number of allocations is the particular kernel exceeded the number of allocations that could be handled by Oclgrind.

assignment has no global side effects and thus gets removed during the optimisation passes. The difference in the behaviour can be observed when the executed LLVM IR instructions are compared (Listing 3.44).

Private address space too small on 32 bit architectures

This problem describes less a failure of Oclgrind but rather a restriction through the choice of configuration parameters at the point the fatal error (Listing 3.45) did occur.

Oclgrind's virtual memory management splits each address into an allocation id part and an offset. The consequence of this scheme are limitations for both number of possible allocations and the maximal size of each allocation. Initially, on 32 bit architectures 8 bit of the address had been reserved for the allocation id, resulting in a maximum of 256 allocations. Respectively, each allocation had a maximal size of 16 MB. While the limit of 256 allocations were enough for the global and local address space some of the generated kernel tried to allocate more memory regions.

After reporting this issue, the behaviour has been changed[56] such that the private address space reserves now 16 bit for the allocation id. Thus the maximal number of allocations increased to 65 536 with a maximal size of 16 kB per allocation. The partitioning for the local and global address space has not changed.

The rationale behind these changes is that there are only a few allocations but large in the local and global address space to share values between different work items or work groups. For instance, each work item could write to one position in a global buffer. In contrast allocations in the private address space, i.e. stack allocations, are typically small but kernels might create a lot of them to store temporary result or to initialise large structs after a SRoA transformation has been applied – which has been the case in which the issue has been detected.

[56] The commits `dc35974` and `53a2140` introduce a variable partitioning scheme such that not all address spaces share the same configuration parameters.

4 Experimental results

The primary goal of this project has been to evaluate the potential of automatic test case reduction in the field of many-core compilers. For this reason the results mainly cover the two key aspects, namely robustness and efficiency, of the reduction process. A reduction framework is considered robust if the final test cases still trigger the bug and are free from undefined behaviour. An additional robustness criterion, but one that is hard to evaluate in practice, is whether the final test cases still trigger the same bug as the original kernels. Moreover, the rate of false positive warnings about allegedly undefined behaviour should be low as otherwise many bugs might be missed. The efficiency of the automatic reductions is evaluated in terms of the runtime per reduction as ultimately the automation makes only sense if results are produced in an *acceptable* amount of time. The exact definition of "acceptable" depends on the particular situation: for low priority bugs it can be sufficient to run the reduction in parallel to the actual development work and therefore longer reduction times to do not cause problems. On the other hand for high priority bugs which possibly block the further development until they have been fixed a faster reduction as in the manual case is desirable.

4.1 Reduction results

This section provides general statistics about the reduction process and the reduced test cases without going into the details of the reductions themselves. This high-level view is useful to demonstrate the general capabilities of the test case reductions of OpenCL kernels independent of the exact fine-tuning. To evaluate the potential of the automated test case reduction out of 35 750 kernels 127 test cases for six modes of CLsmith resulted in

```
Stack dump:
0.      Running pass 'Function Pass Manager' on module ''.
1.      Running pass 'Combine redundant instructions' on function '@entry'
[1]    25338 segmentation fault  cl_launcher -p 0 -d 0 -f CLProg_1438.cl
```

Listing 4.1 – Compiler crash message for one of the reduced test cases. The test case triggered initially a wrong-code bug. During the reduction it must have turned into a compiler crash bug.

wrong-code generation and have been reduced on six different platforms.[57] The total number of reductions however adds up to 272 since some kernels exhibited result differences on more than one configuration, and thus have been reduced on multiple configurations. A complete overview over all performed reductions is shown in Table 4.1. Some of the reduced test cases contain undefined behaviour in terms of uninitialised values or data races which have not been detected and are therefore marked as "failure" or "data race" respectively. These kernels have been excluded from the further evaluations. Furthermore, one of the reduced test cases triggers a compiler-crash bug (Listing 4.1) instead of a wrong-code bug for which the reduction was intended. Nevertheless, initially this test case must have exposed a wrong-code bug because the original kernel computed distinct results depending on whether optimisations were enabled or not. The reasons behind both of these problems are discussed in Section 5.1.1.

Although kernels with different features corresponding to the different modes of CLsmith have been tested, the majority of the reduced test cases does not contain the initial characteristic any more. None of the reduced vector kernels comprised vector operations and in the same way there are no atomic reductions left in the kernels. Only a few kernels still perform data exchanges between the work items or diverge in the control flow. Moreover, it seems not to be the case that the misbehaviour of any of these kernels is related to the "special feature".

Moreover, to increase the runtime performance of test case reduction the number of work items for the kernels in the modes `basic` and `vectors` has been reduced to one prior to the reduction. Since these kernels do not comprise communication between the work items all work items produce the same result. Therefore it is enough to execute the kernel for only one work item to speed up the reduction process. All other kernels are executed with the number of work items that was randomly generated during the creation of the specific kernel.

[57] The platform and device names are anonymised because, for some devices, issues related to performance and correctness may not be disclosed.

Configuration	Success	Crash	Race	Failure
dev-a_basic_n1	24	0	—	12
dev-a_basic_n4	20	0	—	9
dev-a_basic_n8	20	0	—	13
dev-b_vectors_n1	5	0	—	1
dev-b_ar_n1	15	1	0	4
dev-b_itc_n1	18	0	0	5
dev-b_atomics_n1	8	0	5	4
dev-b_divergence_n1	20	0	—	5
dev-c_basic_n8	6	0	—	0
dev-d_basic_n8	28	0	—	14
dev-e_basic_n1	3	0	—	2
dev-e_itc_n1	7	0	2	4
dev-a_basic_n1*	15	0	—	2

Table 4.1 – Overview over the reduction results. The configuration determines on which platform the reduction has been executed, the mode of CLsmith when the kernels were generated and the number of parallel interestingness tests. Details about all used GPUs can be looked up in the respective section in Chapter A. Due to licensing restrictions it is however not possible to provide more concrete information such as a mapping between the bugs and the devices. The mode `inter_thread_comm` is abbreviated with `itc` and `ar` means `atomic_reductions`. The configuration marked with "*" did not use the Clang Static Analyzer in the interestingness tests. The first column "Success" contains the number of interesting test cases which do not contain undefined behaviour after the reduction. In contrast, The column "Failure" contains the number of test cases for which the reduction introduced undefined behaviour. Similar the column "Race" represents the number of test cases which contain data races after the reduction. The column "Crash" means that a test case initially triggered a wrong-code bug but during the compilation a point was reached were the compiler crashes during the execution.

Size [B]	< 400	< 700	< 900	< 1000	< 1500	< 2000
Test cases	1 %	23 %	58 %	68 %	89 %	98 %

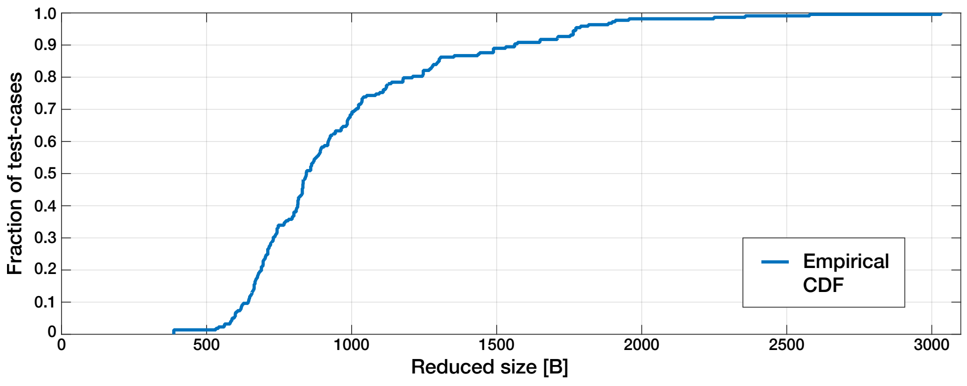


Figure 4.1 – Distribution of the sizes of the reduced test cases. The size of the reduced test cases varies between roughly 300 bytes for the smallest test case and 3 000 bytes for the largest. The majority of the test cases is smaller than 900 bytes.

Another critical aspect of automated reductions is the rate of duplicates among the reduced kernels and the effort required to detect these duplicates. A manual inspection of the kernels reduced in the scope of this project suggests that quite a few could be duplicates that trigger the same bug. But no two kernels were literally identical except when the same kernel has been reduced multiple times on different configurations. A possible explanation for this finding is discussed in Section 5.1.1.

4.1.1 Size of the reduced test cases

One of the most important things to report a bug is to be able to present a test case which is small enough for the developer to easily locate the root cause of the bug. Therefore it is important for the automatic reduction to remove as much as possible of unnecessary information from the original bug-triggering programs.

On average the evaluated test cases have been reduced by 99.2 % which relates to an average absolute size of 844 bytes of the final test cases. The best result was as small as 387 bytes whereas for some outliers the reduction stopped already around 2 000 bytes. The distribution over the sizes of the reduced test cases is visualised in Figure 4.1. Moreover, the results in Figure 4.2 show that the final size of the test cases does not vary significantly between the different modes of the kernels. The difference in the average size per mode are all within the standard deviations.

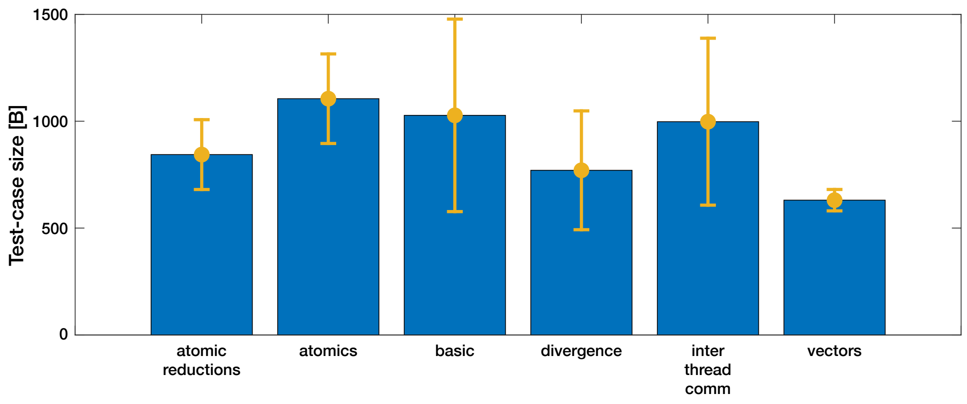


Figure 4.2 – Test case sizes grouped by the mode of the kernels. The final size of the reductions is not significantly different between the modes of the kernels. The variations in the results are all within the standard deviations.

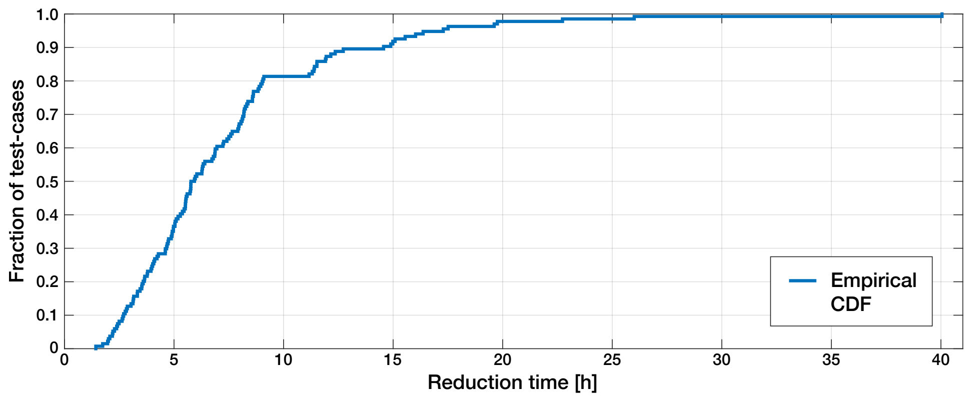


Figure 4.3 – Distribution of the runtimes of sequential reductions. The reduction times vary greatly between 1.4 hours in the best case and over 40 hours in the worst case. Nevertheless the majority of sequential reductions finishes within 6 hours.

4.1.2 Runtime of the reductions

Besides the final size of the reduced kernels, the reduction time is an important factor of the automated process. If it takes too long it might be more reasonable to reduce the kernel by hand if for instance the bug stalls the further development. The evaluated reductions with sequential interestingness tests have an average runtime of 20 893 seconds (a little under six hours) but there is a high fluctuation across all reductions. The fastest time which has been measured for a sequential reduction is only about 1.5 hours whereas some pathological cases took more than 20 hours to complete (Figure 4.3).

To reduce the time per reduction C-Reduce comprises a feature to run multiple interestingness test in parallel (see Section 2.1.4). The gain in performance has been evaluated by reducing the same test cases with different

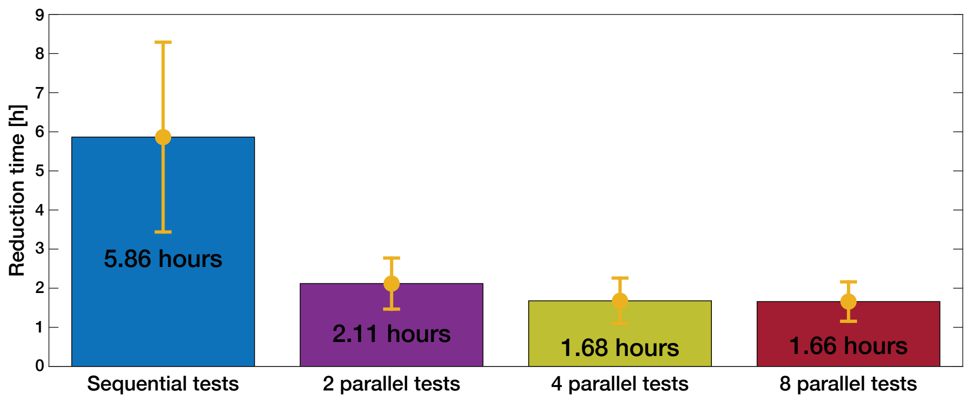


Figure 4.4 – Speedup through parallel interestingness tests. In the beginning the reduction time is inversely proportional to the number of parallel interestingness tests. After the level of parallelism reach four tests no further improvement is achieved. The error bars indicate the variations within one standard deviation to both sides.

levels of parallelism. Up to the point of four parallel interestingness tests the speedup is roughly proportional to the number of tests (Figure 4.4). Afterwards no further increase in the performance is reflected in the results.

The second approach to reduce the overall runtime of a reduction is to reduce the number of work items for which the kernel is executed. During the project this has been done for kernels in the modes `basic` and `vector` since these modes operate independent of the number of work items. When the runtimes of the reductions of these kernels with only one work item are compared to the reductions with a random number of work items a drop of about 30 % in the average runtime can be noticed (data not shown).

4.1.3 Reduction time compared to the test case size

In order to estimate whether some kernels are more worthwhile to be reduced than others the reduction time is plotted against the original size of the kernel (Figure 4.5). However, the results do not show a clear relationship between the original size of the kernels and the reduction time. At least for kernels with larger than average sizes the runtime tends to increase with the size but in general the reduction time and the original size of the kernel seem to be independent.

In addition to the relation of the reduction time to the original size the relation to the final size of the test cases has been evaluated. A positive correlation between the time and the final size can be useful to decide when to abort reductions if they are taking a long time. But as with the original size, the results do exhibit a clear rule (Figure 4.6). While the chances for a small test case seem decrease for reduction that take much longer than the

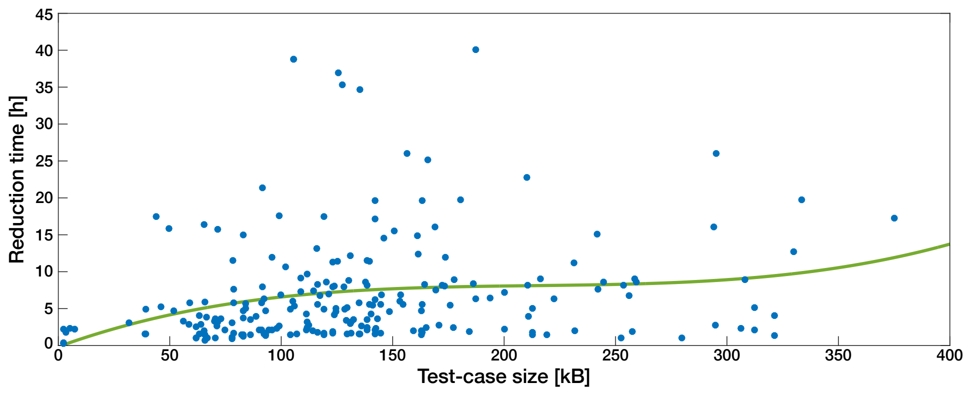


Figure 4.5 – Reduction time depending on the original kernel size. The results do not indicate any strong correlation between the original size f the test cases and the reduction time. For test cases much larger than the average the reduction time seems to increase with the size but in general no pattern is visible.

average for all other test cases the time and the final size seem not to follow any pattern.

4.1.4 Wrong-code bugs

The bugs described in this section are a small part of the outcome of the automated test case reduction on the various platforms. All of them have been reported to the respective vendors and moreover have been confirmed to be reproducible. Due to licensing restrictions for of one of the platforms it is not possible to provide information on which concrete platform the bug has been detected.

Wrong union initialisation

The test case in Listing 4.2 results in a wrong-code error on platform "dev-c". If executed correctly the final buffer has to contain the value "11" which has first been the initialiser of the union member of the struct. Later this member is assigned to the result buffer. This is also the result which is produced if the kernel is executed without optimisations. Yet, with optimisations the result buffer contains instead a zero value which means that likely the initialisation of the union is not handled correctly.

The test case had originally a size of 228 kB and was reduced to 1.9 kB through the automatic reduction. The manual postprocessing time after which the presented kernel has been obtained was about 30 minutes. Later on duplicates of this bug have been found which directly resulted in smaller test cases which could have been reported without any postprocessing.

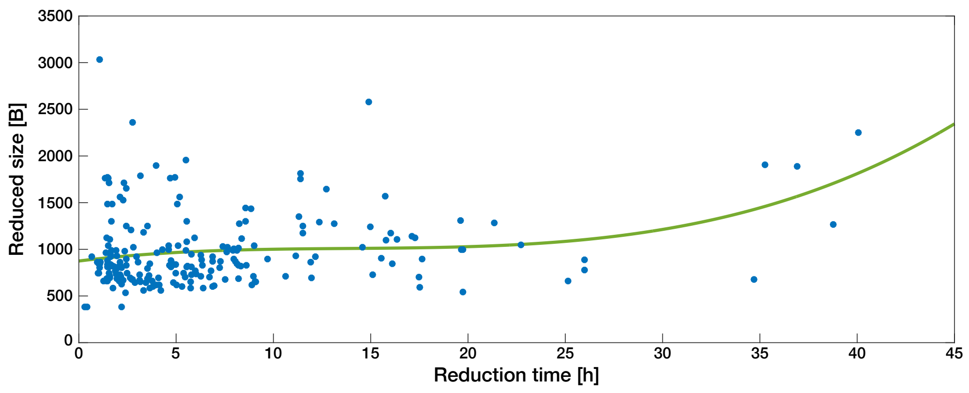


Figure 4.6 – Correlation between final test case size and reduction time. Despite the fact that exceptionally long reductions tend to give worse results there is no clear evidence of a relationship between the final test case size and the time of the reduction. Instead generally most reductions seem to finish with sized below 1 000 bytes.

```
// -g 1 -l 1

union U0 {
    char f0;
    ulong f1;
};

struct S7 {
    ulong g_107; ulong g_129; ulong g_347; ulong g_346;
    int g_1154[6][6]; ulong g_1290; ulong *g_1289;
    ulong g_1368; volatile uint g_1421; char g_1423;
    union U0 g_1444[3];
};

__kernel void entry(__global ulong *p1)
{
    ulong t;
    struct S7 c_2241;
    struct S7 c_2242 = {1,2,3,4,
                        {{5}},6,&t,
                        7,8,9,
                        {{10}, {11}, {12}}};
    c_2241 = c_2242;
    p1[0] = c_2242.g_1444[1].f0;
}
```

Listing 4.2 – Test case triggering a wrong union initialisation. If the union is initialised correctly the value "11" has to be written to the result buffer. Instead, if optimisations are enabled the initialisation seems to be ignored and "0" is written into the buffer.

```
// -g 1 -l 1

struct S0 {
  long a[664];
};

__kernel void entry(__global ulong *p1) {
  p1[0] = 1;
  struct S0 a, b = {{2}};
  a = b;
  barrier(CLK_GLOBAL_MEM_FENCE);
}
```

Listing 4.3 – Test case triggering a stack corruption. When the kernel is executed without optimisation the host program aborts with an error message about an exception during the kernel execution. With optimisations the correct result "1" is printed.

Stack corruption

The test case in Listing 4.3 causes the host application to crash with the error message:

```
Exception detected during test execution!
Exception occurred during kernel execution
```

The behaviour has been produced with disable optimisations on a device which was not part of the other evaluations and thus no anonymised name has been assigned.

The crash could be reproduced by the developers but they do not regard it as actual bug, though they agree that the program should not have crashed. The test case causes a stack corruption through an overuse of automatic variables in the private memory. On the used device only 16 kB stack space is reserved for automatic variables. The structs "`a`" and "`b`" and the initialiser list `{{2}}` occupy around 15 936 bytes ($3 \cdot 664 \cdot 8\,\mathrm{B}$) in total. The allocations for the entry wrapper and the spilled callee-save registers do then exceed the limit and corrupt the stack.

The developers confirmed that stack overflows are not handled very well in the current public release of the drivers but work is going on at the internal version to improve the situation. Further, it is not possible at the moment to query in advance where a given kernel is valid for a given platform since the compiler does not report how much memory would be needed. An enhancement request has been submitted to improve this situation in the future.

The original test case had a size of 140 kB and was automatically reduced to 448 B. Nevertheless it was possible to bring the size down to 169 bytes

```
// -g 1 -l 1

struct S0 {
  ulong f;
};

struct S1 {
  char g;
};

__kernel void entry(__global ulong *r) {
  struct S0 a = {1};
  struct S0 *b = &a;
  struct S1 c = {2};
  r[0] = a.f;
}
```

Listing 4.4 – Test case exhibiting a broken struct initialisation. When the kernel is executed without optimisation a zero is written to the result buffer. With optimisations the buffer correctly contains the value "1".

during the manual postprocessing. Since mostly unnecessary elements had to be removed the manual time was less than 5 minutes.

Wrong struct initialisation

The test case in Listing 4.4 is miscompiled on the device "dev-e" if optimisations are disabled. In the end the result buffer contains the value "0" whereas it should be set to "1". The problem seems to be an incorrect handling of the struct initialisation.

Originally the test case had a size of 158 kB and was automatically reduced to 767 B. After approximately 30 minutes of manual postprocessing the final size of 187 B was reached. Again duplicates have been found for which the postprocessing time would have been shorter.

Broken struct usage

The test case in Listing 4.5 triggers a wrong code generation in the optimisation passes of the OpenCL compiler of device "dev-e". Without optimisations the comparison correctly evaluates to `false` since `0x8000` is smaller than `0x8001`. With optimisations enabled on the other hand, the comparison evaluates to `true`.

The misbehaviour seems to be related to a wrong representation of the `ushort` members of the struct as if they had `short` type and the resulting overflow of the `signed short` range. While both constants easily fit into the range of `unsigned short` (0x0 to 0xffff) they are just large enough to overflow the `signed short` range. Through the overflow the less-than

```
// -g 1 -l 1

struct S {
  ushort a;
  ushort *b;
};

__kernel void entry(__global ulong *r) {
  struct S c;
  struct S d = {0x8001, &c.a};
  c = d;
  r[0] = 0x8000 >= *c.b;
}
```

Listing 4.5 – Test case exhibiting a broken struct usage. When the kernel is executed without optimisation a the correct value "0" written to the result buffer. With optimisations the buffer contains the value "1" which indidates a wrong handling of the comparison in combination with the struct.

relation between the numbers would be inverted which explains the wrong behaviour. This explanation is supported by the fact that the misbehaviour can no longer observed as the constants are further decreased.

The originally 120 kB large test case was automatically reduced to 1 000 B. After 20 minutes of manual postprocessing the final size of 175 B has been obtained.

4.2 C-Reduce

Except for the redesign of the implementation for Windows systems the C-Reduce framework has not been evaluated in general because it has been done before by Regehr et al. [Reg+12] and the usage has not changed. The effects of the transformations on the test cases on the other hand might be different due to the different structure of the OpenCL kernels in comparison to the previous C programs. Therefore this section first describes the course of the reductions regarding the transformations and second the performance of the new Windows implementation.

4.2.1 Transformation statistics

The statistic over all transformations of the evaluated reductions can help to identify unsuccessful transformations which unnecessarily slow down the reduction process. Currently the C-Reduce framework tries to apply 109 different kinds transformations during the reduction process. Table 4.2 lists statistics about the subset of transformations which are most relevant for the OpenCL kernels.

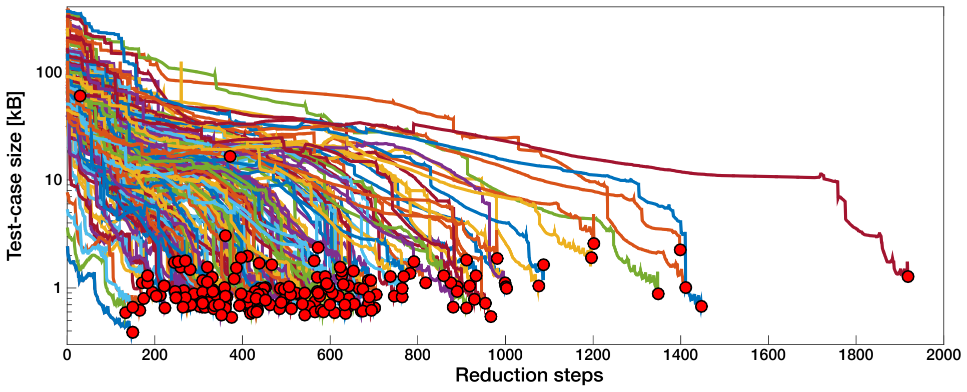


Figure 4.7 – Course of the evaluated reductions regarding the test case sizes. Each step in the diagram corresponds to a successful application of a transformation. Although alter steps belong to later phases in the reduction process they do not encode information about the actual runtime. The red points mark the final test case sizes. It has to be noted that the y-axis is scaled logarithmically.

The results of the "pass_lines" transformations which try to remove complete lines from the source file are only interesting in 3 % of all applications but nevertheless account for nearly 60 % of all successful transformations and 70 % of the total decrease in the test case size. On the other hand, the specialised "pass_clang" transformations have a high success rate but can only be applied in fewer situations and except for the "remove-unused-function" transformation do not remove much of the test cases. Some of these transformations even increase the test case size again. Further noteworthy are the low success rate of the "pass_clex" transformations, and the complete failure of the "pass_balanced" and "pass_ints" transformations.

These characteristics of the different kinds of transformations can also be visualised by plotting the course of the test case sizes during the reductions (Figure 4.7). Each step corresponds to one interesting transformation but abstracts from the actual time which the transformation plus the following interestingness test take. The diagram shows long phases during which the test case size decreases slowly and which are disrupted by a single rapid drop in the size. The former likely correspond to the specialised transformations whereas the sharp stages a produced through the removal of a large contiguous block. All over the course of the reduction small spikes indicate that a for instance a reformatting of the source code has first increased the size of the test case but the following transformation removed the additional characters plus existing parts of the test case.

4.2.2 Performance on Windows

In the scope of the project the C-Reduce framework has been modified to increase the compatibility with Windows systems (see Section 3.5.4). To

Pass	Subpass	Succ. [%]	Contrib. [%]	Del. [%]
pass_lines	1	3.97	18.93	37.79
pass_clang	remove-unused-function	64.09	0.69	26.40
pass_lines	0	4.16	8.13	17.81
pass_lines	2	1.79	8.74	7.05
pass_lines	10	3.04	22.89	6.24
pass_clang	remove-unused-field	87.33	10.72	2.64
pass_indent	$*$	96.60	1.28	1.67
pass_clex	$*$	0.27	12.71	1.12
pass_clang	remove-unused-var	81.06	2.84	0.39
pass_clang	empty-struct-to-int	88.93	0.20	0.05
pass_clang	reduce-pointer-level	24.01	1.78	0.02
pass_clang	replace-array-index-var	23.94	0.02	0.00
pass_clang	reduce-array-dim	69.88	0.05	0.00
pass_clang	simplify-struct	13.79	0.00	0.00
pass_clang	reduce-array-size	41.22	0.05	0.00
pass_clang	remove-array	53.97	0.03	0.00
pass_clang	union-to-struct	79.23	0.15	0.00
pass_clang	remove-pointer	93.32	1.31	0.00
pass_ints	$*$	0	0	0
pass_balanced	$*$	0	0	0
pass_clang	simplify-if	52.36	0.10	−0.04
pass_clang	simplify-comma-expr	91.67	0.35	−0.18
pass_clang	aggregate-to-scalar	54.20	2.48	−0.24

Table 4.2 – Excerpt from the transformation statistic of C-Reduce. The table provides an overview over the most interesting results from the statistic over all transformations performed during the evaluation. The third column shows the fraction of transformations per category after which the interestingness test has been successful. The fourth column represents the ratio of the number of successful applications of the given transformation and the overall number of successful transformations. The last column displays how much the given transformation removes on average from a source file during the reduction. An asterisk in the second column indicates that different subpasses have been combined into one entry. The success rate is then averaged and the contributions and the deletions are summed over all subpasses.

Configuration	Serial runtime [s]	Parallel runtime [s]
Linux	522	166
Windows	2 051	948
MSYS	2 294	1 239
MSYS*	2 458	1 347
Cygwin	2 641	1 484

Table 4.3 – Runtimes for different configurations of C-Reduce on Windows. The "Windows" configuration represents the new implementation of C-Reduce. The other configurations are named after the environment which has been used to run the old implementation. In contrast to all other configurations the "MSYS*" configuration still uses system calls for most of the file system operations. Those have been replaced by faster Perl functions in the other configurations. During the parallel execution the number of simultaneous interestingness test was limited to five.

measure the benefits in terms of the runtime performance the new implementation has been compared against different configurations of the old version. For this purpose the first test ("test0") which is included in the C-Reduce sources is used as a representative interestingness test to reduce the test case `file1.c` on the different configurations. Since the new implementation has no support for shell scripts it has been converted into a semantically equivalent batch script. All other configurations use the original shell script.

In the first experiment the runtimes have been measured for a serial execution of all interestingness tests. Afterwards the test has been repeated with activated test case parallelisation since this is a potentially weak spot in the native Windows implementation. The results of both runs are shown in Table 4.3. The number of parallel tests has been limited to five as this was the number virtual cores on the Windows machine. Additionally, both measurements include the runtime on a Linux system as baseline to provide an orientation of the performance on Windows in general.

Not surprisingly the Linux configuration achieves the best results. The new implementation places second followed by the "MSYS" configurations and lastly "Cygwin". In terms of speedup through the parallelisation again the Linux configuration is top ranked with a roughly three times shorter runtime. The new, native solution is still able to reduce the runtime down to slightly less than a half whereas the other configuration have only an improvement of 1.8 compared to the serial execution.

Additionally, the time to run a single instance of the interestingness test in combination with the test case has been measured with 1.4 seconds for the batch script used for the native implementation. This is already 0.3 seconds faster than running the original shell script with Cygwin, but in contrast, the executing the shell script on a Linux machine takes only 0.8 seconds on average.

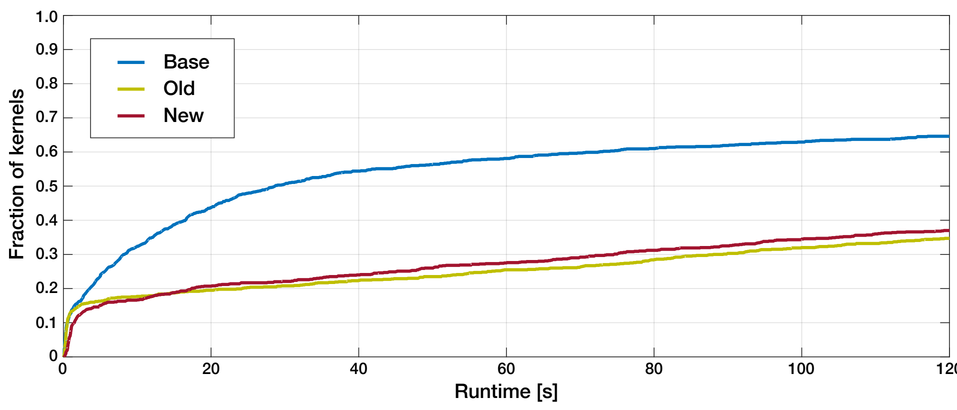


Figure 4.8 – Distribution of the runtimes for the undefined value plugins in Oclgrind. Approximately 65 % of the runs of Oclgrind without any plugin (Base) finish within the time-limit of 120 seconds. If either of the plugins to detect uninitialised values is loaded the rate of completed runs drops down to 35 %. Kernels with a runtime up to a few seconds are mostly unaffected by the overhead of the plugins. Up to a runtime of 40 seconds the number of executions that finish within a given time increases more slowly and constantly compared to the baseline without plugins such that the increase of the influence of the plugins is disproportionally high. For kernels with even longer runtimes the plugins add a constant overhead.

4.3 Uninitialised value detection in Oclgrind

As with C-Reduce, optimising the general performance of Oclgrind has not been not part of this project. Therefore the evaluation concerns only the efficiency of the new plugin to detect unintialised values which has been implemented in the course of the project. Its runtime has been compared to Oclgrind without any plugins as baseline and to the runtime of Oclgrind in combination with the old plugin. For this reason the three configurations have been used to simulate the same 1 000 randomly generated kernels. For each kernel the execution time was restricted to 120 seconds after which the particular kernel was marked as "timeout".

This limit affected approximately 35 % of the runs of Oclgrind without any plugin (Figure 4.8). In comparison, on a GPU only 20 % of the executions took longer than 120 seconds. If either of the plugins to detect uninitialised values has been used the relative number of timeouts increased to roughly 65 % without any significant difference between the old and the new plugin.

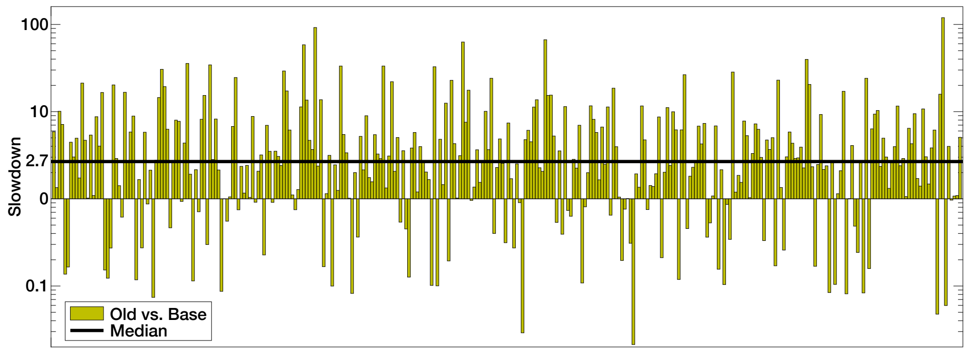
Afterwards the runtimes of the remaining 320 kernels which had no timeout on all of the platforms have been compared directly to measure the relative slowdown of the plugins (Figure 4.9). The first diagram shows the slowdown of the old plugin over Oclgrind without plugins. There are a few cases in which the runtime performance increase but the median over all results indicates an average slowdown of 2.7. The slowdown of the new plugin over the baseline is around 4.6 and there are fewer situations in which the plugin improved the performance. Lastly the new plugin is directly compared to the old plugin resulting in a 1.3 times worse runtime of the new plugin.

4.4 Interestingness tests

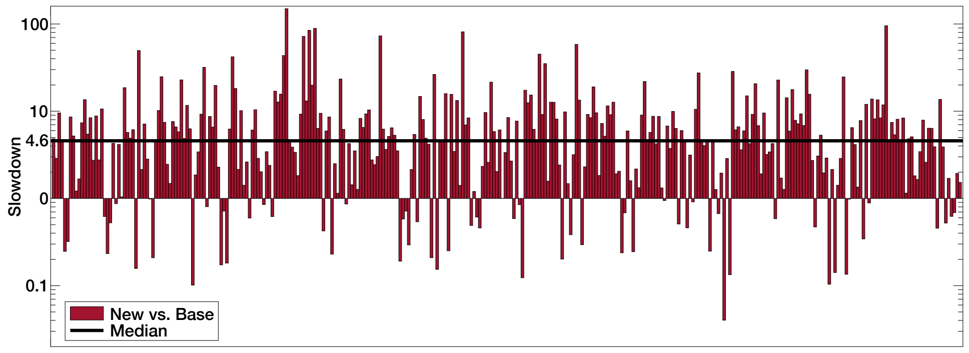
A major concern of the interestingness tests is their runtime. Since on average 30 000 tests are performed during a single reduction the overall runtime would increase by nearly one hour even if every tests takes only a tenth of a second longer. Currently the Clang Static Analyzer and Oclgrind are used in the interestingness tests and both tools can take a long time for complex or large kernels. If only the correctness of the interestingness tests is considered the Clang Static Analyzer is not needed as Oclgrind is able to perform the same sanity checks. For this reason it has been evaluated if it is beneficial in terms of the runtime of the reduction not to use the Clang Static Analyzer.

The diagram in Figure 4.10 shows clearly that the reduction times for the interestingness tests which invoke the Clang Static Analyzer prior to Oclgrind are shorter than for the interestingness tests which make only use of Oclgrind. On average the time per reduction increased by two and a half hours in the latter case which corresponds to an average increase of 0.3 seconds per interestingness test.

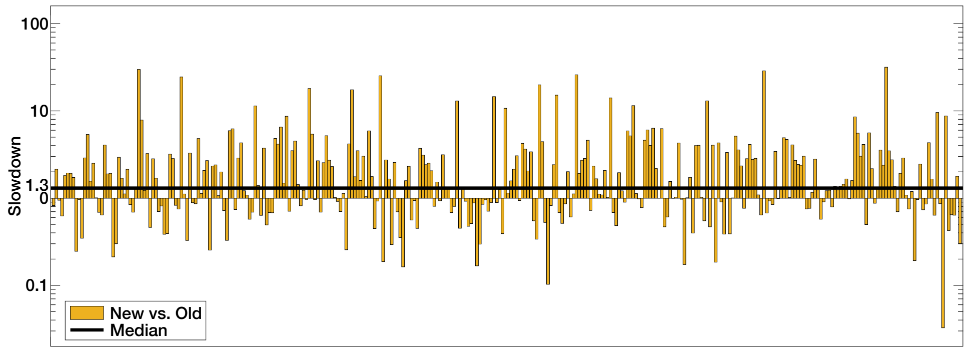
In addition to the positive effect of the Clang Static Analyzer on the runtime this experiment revealed also a drawback of the tool. For two of the test cases in Figure 4.10 it reported false positive warnings about an alleged dereference of a null pointer. The warning prohibited the reduction of these test cases for the tests which made use of the analyser. In contrast, the analysis of Oclgrind was correct and the reduction have been successful.



(a) Old plugin vs. base



(b) New plugin vs. base



(c) New plugin vs. old plugin

Figure 4.9 – Relative slowdown of the uninitialised value plugins. The diagrams show the relative slowdown between different configurations of Oclgrind. The comparison includes all of the 1 000 automatically generated kernels which did not receive a timeout with any configuration. This reduced the number to 320 kernels. The figures (a) and (b) show the slowdown of both versions of the uninitialised value plugin compared against Oclgrind without any plugin. The third figure (c) displays the additional slowdown of the new plugin in comparison to the old plugin.

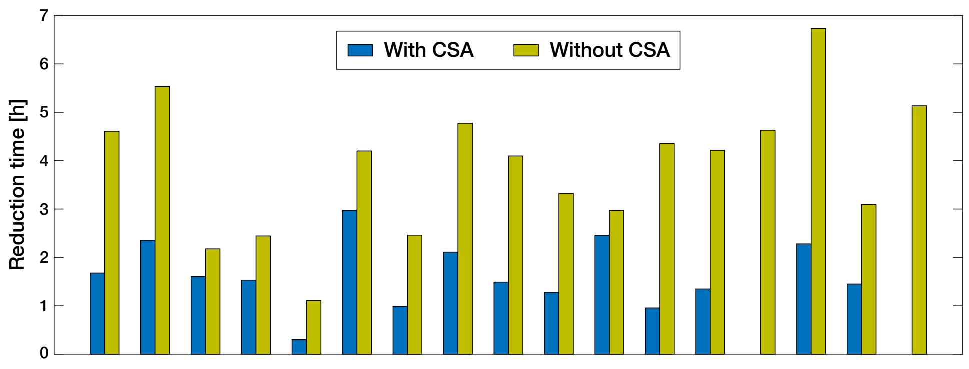


Figure 4.10 – Reduction times for interestingness tests with and without CSA. The runtime of the reductions is significantly shorter if the Clang Static Analyzer is included in the interestingness tests and run before Oclgrind. For two of the test cases the Clang Static Analyzer reports a false positive warning about a dereference of a null pointer such that the reduction could only be performed without the analyser.

5 Discussion

This chapter evaluates and assesses the results obtained via the experiments of Chapter 4 with respect to the applicability of automatic test case reductions in the area of many-core compilers. Key aspects of the discussion are the robustness of the different tools along the reduction process and their time efficiency. For this reason the reduced test cases and the runtimes of the reductions have been analysed and compared to existing results where applicable.

First the results of the reductions are reviewed at an abstract level without going into the details of the of the different steps and used tools. Afterwards the influence of the C-Reduce and the Oclgrind program on the reduction process is examined. Finally, the interestingness tests in particular are evaluated.

5.1 Test case reductions

During the evaluation phase of the automatic test case reduction more than 100 OpenCL kernels have been successfully reduced. Most of the reduced test cases did not comprise any of the features related to the mode in which the kernels had been generated, though some bugs required the presence of barrier operations. These observations are in accordance with the results described by Lidbury et al. [Lid+15b]. The authors also did not find any bugs related to inter thread communication or atomic operations, except that barriers were part of some bugs. Moreover, that in most of the modes only "basic" bugs have been found implies that the size of the reduced test cases should not vary which is also represented in the results of this project.

5.1.1 Robustness of the test case reductions

Despite the fact that most of the reductions were successful, the few reduced kernels which contained undefined behaviour show that defining accurate interestingness test is not trivial. There exist a large number of sources of undefined behaviour and not all can be checked easily or without restricting the reduction process too much. The strategy during this project has been to be conservative about warnings and to add them only to a blacklist when they actually would have prevented an instance of undefined behaviour. Starting directly with a large blacklist of warnings can result in unnecessary restrictions of the reduction process and might finally lead to worse reduction results. However, this approach of defining the warnings that are considered as harmful incrementally during the development of the interestingness tests has the drawback that sometimes undefined behaviour is not caught and makes its way into the reduced test cases as it was the case in the evaluation. After a manual inspection of the reduced test cases – which has to be performed anyway – the undefined behaviour has been located and the interestingness tests have been extended by the new indicators of this kind of undefined behaviour. The next step would be to rerun the particular reductions to verify that the undefined behaviour no longer occurs.

Besides the few cases in which the reduction ended because of undefined behaviour, one test case initially triggered a wrong-code bug but changed into a compiler-crash bug during the reduction. This relates to the phenomenon of "bug slippage" which is mentioned by Chen et al. [Che+13]. In their experiments they found that for GCC bug slippage is not a major concern as only one instance had been observed. This seems to be in agreement with the results of this project where the test case triggering the compiler crash is the only known occurrence. However, in the scope of this project it has not been evaluated whether all of the reduced test cases trigger the same wrong-code bug as the original test cases. The only possibility to determine whether the original bug has been preserved during the reduction seems to be to fix the bug and to rerun the interestingness test on the original test case. This is also the method which Chen et al. [Che+13] used to analyse their reductions for bugs in the GCC compiler. Nevertheless, this provides only a definite answer to the question of bug slippage if the repeated interestingness test succeeds after the bug has been fixed, in which case it is clear that no bug slippage has occurred. If the repeated test fails then either there was bug slippage, or the original test case was prone to *more than one* compiler bug – which is entirely possible if the large number of overall detected bugs is considered. Furthermore, the latter scenario shows that for an ultimate answer a new version of the compiler has to be used in which solely the bug in question has been fixed, which is nearly infeasible to achieve.

Further, the issue of duplicates, i.e. different test cases which have the same root cause, has not been addressed in this project. Extracting the problem from a test case is mostly not feasible without the chance to actually debug the compiler. Again Chen et al. [Che+13] report that during their evaluation about 2 % of the test cases became literally equivalent after the reduction which makes it easy to discard them. On the other hand, a comparison of the reduced test cases in this project did not reveal any such duplicates. Depending on the specific members of the struct which is used as replacement for global variables the reductions seem to terminate in different but roughly equivalent local optima. For instance, for a number of test cases it has been observed that the order of the remaining members after the reduction varies, but that one of them is always a pointer. For a human it seems therefore no problem to mark all these cases as duplicates but doing it automatically is not as easy as with literally identical test cases. Besides this possible explanation why it might be unlikely to find exact duplicates it has to be noted that due to the comparably small number of overall test cases only three duplicates can be expected if the rate of 2 % is assumed.

5.1.2 Performance of the test case reductions

To report a bug one has to be able to present a test case which reproduces the wrong-behaviour but which is also small enough to reveal the root cause of the problem without spending to much time in analysing the test case. In practise it is hard to guess when a test case is *small enough* to be reported and it seems almost impossible to evaluate the performance of the automatic reduction based on such a loose definition. However, the GCC developers state that “[m]ost test cases can be reduced to fewer than 30 lines!”[58]

On average the OpenCL kernels are reduced by two orders of magnitude to an average size of 844 bytes. This is roughly three times larger than the results which Regehr et al. [Reg+12] obtained by when they reduced C programs with C-Reduce. While the reduction process has not changed significantly a possible explanation for the larger final test cases might be the different characteristic of the test cases. The automatically generated OpenCL kernels rely heavily on structs as a replacement for global variables [Lid+15b]. The consequence is that often struct members cannot be removed even if they do not directly contribute to the bug because this would change the alignment or padding inside the struct which itself can be the reason for miscompilations.

Yet, the automatically reduced test cases are most of the time small enough to be reported directly without the need for further manual reductions. The test case shown in Listing C.7 has a size of 844 bytes which is exemplary for the evaluated reductions. With 39 lines it is slightly larger than the GCC developers expect but with a bit of reformatting of the spacing it would also fit into 30 lines. On the other hand, the test case is far from optimal in terms of its size. With 15 minutes of human reduction effort, it was found

[58] See `https://gcc.gnu.org/bugs/minimize.html`, visited on 29/08/2015.

```
// -g 1,1,1 -l 1,1,1
struct S1 {
  volatile ushort g_300;
};

__kernel void entry(__global ulong *p1) {
  struct S1 e = {1};
  *p1 = 0;
  for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 4; j++) {
      for(int k = 0; k < 6; k++) {
        *p1 += e.g_300;
      }
    }
  }
}
```

Listing 5.1 – Automatically reduced test case with manual postprocesing. After the additional manual reduction the previously automatically reducerd test case from Listing C.7 has now only a size of 279 bytes. It still triggers a wrong-code bug on device "dev-a". The correct result is "0x48" but with optimisations enabled the computed output is "0x52".

that the test case could be further reduced by a factor of three to its final size of 279 bytes (Listing 5.1). To achieve this result, multiple correlated and non-trivial changes had to be performed at once. For instance had the functions to be inlined and the access to the result buffer had be changed from an indexed access into an array to a dereference of a pointer. These changes were too complex than they could have been performed by the automatic reduction as thus the result remained non-optimal.

Even though the automatic reduction got stuck in a local optimum the manual postprocessing time of 15 is much less than it would have taken to reduce the original test case. In the scope of this project these short additional manual times could be confirmed in a few more trials to reduce the results of the automatic reduction process further. For some "simple" bugs no more than five minutes were necessary to obtain the smallest result whereas more complex bugs took up to thirty minutes which is still acceptable.

In addition to the size of automatically reduced test cases and the manual effort needed after automatic reductions, the third important aspect of the performance is the time that automatic reductions take themselves. The reduction time of nearly six hours on average for the sequential interestingness tests seems to long as it could be useful for all kinds of bugs. For high priority bugs which possibly block the further development it might be better to reduce test cases manually since a skilled developer is likely to be faster. On the other hand, for low priority bugs which can be fixed at any time there is no disadvantage of proceeding with actual development while test cases are reduced in the background. Moreover, another argument to aim at faster reduction times is that in the many-core context, where a device might not support parallel jobs, it is not necessarily possible to run lots of reductions on the same device in parallel to reduce the overall time.

With the parallel interestingness tests enabled the results show a significant decrease in the reduction time. The average of 1.7 hours with four parallel interestingness tests is comparable to the runtimes which Regehr et al. [Reg+12] report for using C-Reduce in combination with the dynamic instrumentation tool KCC. It has to be noted that the average runtime of 1.7 hours has been measured for executions with the reduced number of only one work item which is equivalent to one run of a C program in the paper of Regehr et al. Without the change in the number of work items the results suggest an approximately 40 % longer reduction time. A more detailed analysis of the influence of the parallel interestingness tests in combination with the number of work items is beyond the scope of the project but should be performed in the future.

The reason why the results show no further improvement after a parallelism of four interestingness has been reached can be explained by the high consumption of RAM by the Oclgrind tool. By inspecting a small amount of the reductions processes manually it has been noticed that already one instance of Oclgrind can consume multiple gigabytes of main memory. In the parallel scenario multiple instances filled all 16 GB of RAM and thus slowed down the reduction due to swapping memory or because interestingness tests were aborted by the operating system due to the excessive memory consumption. After the problems with eight parallel interestingness tests and 16 GB main memory had been noticed a few reductions have been tested on a machine with 32 GB of RAM. But also this seems not to be enough and suggests that eight parallel interestingness tests are simply not realisable with Oclgrind.

Finally, to maximise the benefit of the automatic reductions especially for the slow sequential reductions it would be useful to have some guideline which test cases should be reduced first or after which amount of time the reduction could be aborted as it would not a small test case anyway. Unfortunately the results contain no clear indication for a dependence between the size of the original test case and its reduction time or a relationship between the final size and the reduction time. It rather seems to be the structure of the test case which determines the duration of the reduction but no strong evidence for this assumption has been found in the scope of this project.

5.2 C-Reduce

Except of the new implementation for Windows systems the principles behind the C-Reduce framework have not been changed in the scope of this project and hence are not evaluated. Yet, the input to the reduction process is different which can effect the effectiveness of the transformations, although the OpenCL kernels have much in common which the previous C programs. In order to optimise the performance of the reductions some specialised transformations for features not present in the OpenCL C language are automatically deactivated by C-Reduce and the performance of the more general transformations has been evaluated manually.

5.2.1 Performance of the transformations

The obtained results show that finding the *best* configuration of the transformations is a hard problem as it spans across multiple dimensions. The success rate of a given transformation influences directly which portion of the time spent on the transformation and respectively the following interestingness test actually contributes to the final test case. However, it is not generally possible to deactivate these transformations. For instance the "pass_line" transformations which try to remove contiguous blocks have a low success rate because their search space is huge. Yet they form nearly 60 % of all successful transformations and remove the largest portion of the test case. On the other hand it is also not possible to deactivate all transformations that do not remove much of the test case. Most of the specialised functions only restructure the test case and can even increase its size again but help to prevent circumvent local optima.

To find a final solution to the optimal set of transformations a more detailed evaluation has to be performed but already the current results give some hints. The "pass_clex" transformations remove specific tokens or pattern of tokens from the test case and thus cannot remove large contiguous chunks. Although they account for 13 % of the successful transformations they removed only 1 % of the test cases and had a extremely low success rate of 0.3 %. It might therefore be worth to compare reductions with and without these transformations in terms of the runtime and final test case size to determine whether it would be beneficial to deactivate this group. Furthermore, the "pass_ints" and "pass_balanced" transformations had not single successful application although some of the reduced test case contained valid instances for these transformations. For future reductions the reason for the failed transformations should be examined to improve the reduction results. Lastly and not related to the runtime performance, the renaming operations of variables have not always been useful. While they simplify the names of variables if they were generated randomly (`g_491` $\rightarrow$ `b`) they make it more difficult to understand test case in which the variables have semantic names (`g_comm_values` $\rightarrow$ `e`).

Furthermore, the evaluation of the course of the test case sizes during the reduction process reveals no explicit shape which could be used to optimise the used transformations. Each reduction seems to have its unique sequence of transformation instances and the original test case does not correlate to the final reduced size. But nevertheless there exist common pattern such as long phases of slow decrease or abrupt drops which might help to extract features for the corresponding test case. If further analyses will confirm the existence of such characteristics they could for instance be used to detect duplicates among the reduced test cases.

5.2.2 Performance on Windows

Maybe the most important advantage of the modification to allow C-Reduce to run natively on Windows machines is the simpler configuration and setup. Two years ago, a discussion about C-Reduce on Windows systems popped up on the mailing of C-Reduce[59] The author tried to build C-Reduce in combination with the Cygwin environment but ran into several problems. In contrast, with the new CMake build system all tools can be built without the need to manually modify configuration files or to worry about how to setup a suitable development environment to run the build files.

Back in 2013, John Regehr finally managed to run the same test which has been used in this project (Section 4.2.2) under the Cygwin environment.[60] However, the runtime of “a little more than 12 hours” is far worse than all of the times measured during this project and thus cannot be used as a baseline for the comparison. Nevertheless, the serial reduction in combination with the Cygwin environment is still the slowest although the MSYS system decreases the runtime only by 13 %. The native implementation on the other hand yields an improvement of 22 % compared to the Cygwin system.

The boost in the performance can be attributed to the usage of the `CreateProcess` mechanism instead of the Fork-Exec model. The latter had to be emulated by the used environments since Windows does not support it natively. The smaller overhead of creating new processes in the new implementation leads to an even better improvement of 36 % over the “Cygwin” configuration if the parallel scenario is regarded. Though the “MSYS” configuration is also able to improve compared to the serial reduction it gains only additional 3 %.

Despite the increase in performance the reductions on Windows systems are with a factor around four still significantly slower than on Linux. But not only the C-Reduce implementation itself is responsible for the slowdown. The results show that the semantically equivalent batch script is nearly two times less efficient than the shell script on a Linux machine. Still it improves the runtime by 17 % if compared to running the shell script on Windows.

[59] See `http://www.flux.utah.edu/listarchives/creduce-dev/2013-July/000418.html`, visited on 25/08/2015.
[60] See `http://www.flux.utah.edu/listarchives/creduce-dev/2013-July/000439.html`, visited on 25/08/2015.

5.3 Oclgrind

In the course of this project Oclgrind has become a central part of the interestingness tests. Before it was used as dynamical analysis tool all reductions ended with test cases which contained undefined behaviour. Oclgrind complements the checks of the static tools and is currently the only tool to prevent data races in the kernels. The results in Table 4.1 still contain a few cases of failed reductions and test cases exhibiting data races but these issue have been fixed and the latest version successfully prevents undefined behaviour in all these cases.

Although Oclgrind is, compared to the static checkers, quite slow and has a high memory consumption there is currently no alternative. One possibility to increase the performance of Oclgrind could be run the tool in a staged manor instead of with all plugins at the same time. Fast and lightweight plugins could be executed first while slow plugins and those with a hight memory consumption would only be executed if the previous analyses succeeded. This approach introduces additional overhead for tests which finally succeed or only fail after the majority of the checks has been performed but the assumption is that most of the transformations will produce invalid test cases. In this situation the shorter runtime for most of the interestingness tests could outweigh the longer runtime for the tests which are actually successful. The evaluation of this different approach has not been done in the scope this project as the main goal was to evaluate the potential of reductions of OpenCL kernels in general than to focus on optimisations of specific tools. It might especially be useful if in the future more Oclgrind plugins are involved in the reduction process.

5.3.1 Data race detection

To prevent data races during the reduction of kernels generated in the modes `atomics`, `atomic_reductions` and `inter_thread_comm` the races detection plugin of Oclgrind has been used. Initially uniform write-write races had been tolerated which why some of the reduced test cases are marked as “Race” (see Table 4.1). Later the option `--uniform-writes` has been added to the invocation of Oclgrind to warn also about these types of races where multiple work items write the same data concurrently. Since the generated test cases are entirely race free the reduction should not add any kind of race even if it might not cause direct problems. In the new configuration Oclgrind also warns about races in all reduced test cases which have been manually marked as invalid.

In principle the race detection plugin of Oclgrind is capable to detect all *dynamic* data races which actually occur during the execution of the kernel. It cannot warn about all data races that are *statically* present in the source code. For instance data races in dynamically dead code regions will never occur during the execution and hence no warning is emitted. But this should be no issue for the reduction since dead code cannot introduce undefined behaviour and is likely to be removed during the reduction anyway.

Nevertheless, in practice the plugin has some weak spots as mentioned in the corresponding Wiki entry.[61] Especially, races between atomic and non-atomic operations in different work groups can remain undetected and the `atomic_cmpxchg` instruction is not handled at all. Until these issues are addressed there are mainly two possible alternatives to shield against data races. One is to include the *GPUVerify* tool [Bet+15; Bar+14][62] into the interestingness tests. It provides methods for a formal analysis of GPU kernels and is therefore also able to signalise potential data race. Other than Oclgrind it is a static analyser and hence warns about data races in non-executed code segments too. This means that the warnings might be false positives which can prevent the interestingness test to succeed even if the kernel would not have resulted in undefined behaviour.

The second way to prevent undefined behaviour caused by data races during the reduction is to add some hard-coded check to the interestingness tests. These checks must stop transformations which would introduce data races into the kernel for instance by removing barriers or guards around atomic blocks. While this has the potential to solve the problem without the need for a sophisticated and time costly analysis it can handicap the reduction itself leading to larger results. Further, it can be impractical to write checks for all possible situations that could introduce undefined behaviour.

The task to determine the best option to handle the detection of data races is left over for future projects as it would have gone beyond the scope of this project. During the evaluation most of the reductions were performed for modes of CLsmith which do not include concurrent data structures and thus only a few test cases where effected by data races. For the later evaluations they have been ignored.

5.3.2 Uninitialised value detection

The main goal of the new plugin to detect uninitialised values during the reduction of the randomly generated OpenCL kernels was to improve on the high rate of false positives which have been emitted by the old plugin. In fact the tests during the development of the plugin have not produced any false positives and also in the reductions run for the evaluation no false positives have been noticed. This means that the decision to implement the shadow propagation often at the coarse level of operand finally paid off.

[61] See `https://github.com/jrprice/Oclgrind/wiki/Data-Race-Detection`, visited on 27/08/2015.

[62] See `http://multicore.doc.ic.ac.uk/tools/GPUVerify/`, visited on 27/08/2015.

Nevertheless some of the reduced test cases still exhibited undefined behaviour. The plugin failed to report uninitialised values for loads from undefined addresses. In general it is very unlikely for arbitrary undefined addresses to point to an allocated and valid area in the memory and thus to remain unnoticed by the separate address check of Oclgrind. But, especially in situation where the address is computed via the `getelementptr` instruction and one of the index operands is undefined the chances are high that the operand just happens to be zero (Listing 5.2). Accessing the first element of an array for instance, is always a safe operation and thus is not detected by the address check or out-of-bounds check of Oclgrind. Since the address operand of the `load` instruction was not checked a clean value was loaded from memory and the propagation chain of the undefined address operand stopped without emitting a warning.

The issue has been fixed in the latest version of Oclgrind and already reduced test cases have been checked again. All which did not pass the second check have been marked as “failed” and are excluded from the evaluation results. Further, after the problem had been addressed no more reduced test cases contained undefined behaviour as a result of the usage of undefined values. Hence the false negative rate is equally low as the false positive rate.

But still there exist cases in which even a perfect plugin could not detect the undefined behaviour. The problem is that Oclgrind depends on the Clang compiler to generate the LLVM intermediate representation which is then used to simulate the kernel execution. If the kernel contains undefined behaviour the compiler does not have to make any guarantees for the compiled program and even does not have to notify the user about potential problems [C99, § 3.4.3.2]. Therefore it is possible that the OpenCL code of the kernel indeed contains undefined behaviour which disappears after the translation into the LLVM IR. Although this has not been observed during the reductions accidentally one of the constructed regression tests for the new plugin exposed this phenomenon (Listing 5.3).

The example does not produce “wrong” code if all optimisations are disabled. This is the default in the interestingness tests which reduces the risk to run into this kind of problem. But nevertheless there is no guarantee that the compiler does not alter the “misbehaviour” of a program even if optimisations are disabled.

In terms of runtime efficiency a slowdown over the old plugin has been expected and is also reflected in the results. Yet, the roughly five times longer runtime compared to Oclgrind without plugins is a fair result. It is much better than Valgrind’s Memcheck plugin for which the overall slowdown is four times higher. Even if only the parts of the Memcheck plugin which directly correspond to the handling of uninitialised values are taken into account a ten times longer runtime is expected. Compared to MemorySanitizer the new Oclgrind plugin performs slightly worse since MemorySanitizer had only a three times overhead. This might be partly due to overheads of the

```
__kernel void kf(__global ulong *r)
{
  ulong a[] = {1};
  ulong i;

  r[0] = a[i];
}
```

(a) Test case

```
@kf.a = private unnamed_addr constant [1 x i64] [i64 1], align 8

define void @kf(i64* %r) #0 {
  %1 = alloca i64*, align 8
  %a = alloca [1 x i64], align 8
  %i = alloca i64, align 8
  store i64* %r, i64** %1, align 8
  %2 = bitcast [1 x i64]* %a to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %2, i8* bitcast ([1 x i64]* @kf.a to i8*),
    i64 8, i32 8, i1 false)
  %3 = load i64, i64* %i, align 8
  %4 = getelementptr inbounds [1 x i64], [1 x i64]* %a, i32 0, i64 %3
  %5 = load i64, i64* %4, align 8
  %6 = load i64*, i64** %1, align 8
  %7 = getelementptr inbounds i64, i64* %6, i64 0
  store i64 %5, i64* %7, align 8
  ret void
}
```

(b) Generated LLVM IR

Listing 5.2 – Problem of losing shadows on load instructions. The variable "`i`" in the test case (a) is uninitialised but used to index the array "`a`". If by pure chance the variable has a value of zero Oclgrind did not report a warning for the usage of an uninitialised value. The problem was that the shadow propagation of the undefined variable stopped at the highlighted `load` instruction in the LLVM IR (b). Despite that the address operand "`%4`" is undefined a clean value is loaded from memory and used as new shadow for the `load` instruction. The information about the undefined address was lost.

```
__kernel void test(__global ulong *r)
{
  ulong v;
  int one = 1;

  if(one == 2) {
      // Dynamically dead code
      v = 42;
  }

  *r = v;
}
```

(a) Test case

```
define void @test(i64* nocapture %r) #0 {
  store i64 42, i64* %r, align 8, !tbaa !8
  ret void
}
```

(b) Generated LLVM IR

Listing 5.3 – Example of undefined behaviour in the Clang compiler. The behaviour of the test case (a) is undefined since dynamically the variable "v" is never initialised. This frees the compiler from the requirements to produce a valid program but unfortunately it decides to simply initialise the variable as if the dead code would be executed. To reproduce the behaviour at least optimisations `-O1` have to be enabled.

object-oriented and less optimised design of Oclgrind. But another important difference is that Oclgrind has to simulate the execution of the kernels whereas the programs instrumented with MemorySanitizer are directly executed through the operating system. So all in all, the performance penalties are less than expected and competitive to the other available tools.

Moreover, the slowdown is unlikely to affect the result of the reductions by causing interestingness to fail due to the time-limit. The results show that approximately the same portion of the executions ran into timeouts regardless of the plugin which had been used.

The effect that the execution of some kernels was faster when the plugins were activated seems to be counterintuitive on the first sight. One possible explanation is a change in the caching behaviour which could be provoked by the additional shadow data. Or the compiler might be able to apply different and more efficient optimisations due to the added instructions of the shadow propagation. Further, the larger number of instructions is not necessarily proportional to the final overhead of the plugin. The CPU might be able to fit additional instructions into otherwise stalled pipelines when for instance Oclgrind itself has to wait for new data for the kernel.

Other than the runtime the memory consumption has not been measured for a representative comparison since it is much easier to derive from the implementation details than the effects on the runtime. The current version of the plugin maintains a one-to-one mapping between application data and shadow data and thus doubles the total amount of memory which is allocated throughout the execution. This, however, does not mean that the amount of allocated application memory at any given point during the execution is maximal twice as high as normally. Due to the lazy deallocation of shadow memory it can still be allocated when the application memory has already been released.

5.4 Interestingness tests

The robustness and accuracy of the interestingness tests has been a central aspect of this project. As early trials in the beginning of the project revealed is it essential for the reduction to have precise interestingness tests. There exists a large number of situations in which the behaviour of a reduced test case can be undefined. Further, it is not enough to keep the final test case free from undefined behaviour because as soon as it is introduced during the reduction it interferes with the actual misbehaviour of the kernel and leads to false interestingness tests.

The results in Table 4.1 show that the developed interestingness tests are sufficient to prevent invalid reduction results for the randomly generated kernels. The data race issues and the general failures due to undetected uninitialised values have been patched as soon as they have been noticed during the evaluation. Therefore all of the manually identified problems in the evaluated kernels are reliably detected by the latest version of the interestingness tests.

Yet detecting all cases of undefined behaviour is only one side of the coin. It has to be complemented with a low rate of false positive warnings. While false negatives result in invalid test cases false positives prevent the reduction completely. For this reason they might be considered less harmful since at least no time is wasted on a finally invalid kernel. Nevertheless, the aim is to keep their number small as otherwise potentially interesting bugs are missed. In the current interestingness tests the largest source of false positive warnings is the Clang Static Analyzer. In particular for complex and large kernels the static analysis is stretched to its limits and the experiments without the Clang Static Analyzer revealed two wrong-code bugs which have been missed due to a wrong warning about an alleged dereference of a null pointer. Still it is not possible to remove the tool from the interestingness tests as it is an important factor for their runtime performance. Without the static check before Oclgrind is executed the runtime per test increased by 0.3 seconds on average which accumulates to 2.5 extra hours with approximately 30 000 tests per reduction. Although the Clang Static Analyzer is slow for large kernels – which was the motivation for dropping it – its static analysis can in general be immensely faster than the dynamic execution of Oclgrind. Therefore, a compromise between accuracy and performance seems to be to determine the interestingness, i.e. whether it is free from undefined behaviour and triggers a bug, of each generated test case initially without the Clang Static Analyzer such that no test case is wrongly rejected due to the false positive warnings about allegedly undefined behaviour. Afterwards the reductions are started with the Clang Static Analyzer and it is only deactivated for those test cases where the reduction cannot be started otherwise because of a false positive warning in the original test case. This way most of the reductions profit from the speedup through the Clang Static Analyzer and at the same time no wrong-code bug is overlooked.

Another alternative to optimise the runtime of the interestingness tests could be to tighten the time-limit for each check in the test to reduce the worst-case time. To determine a good time-limit requires a deeper analysis of the effect of the time-limit on the reduction process than it has been conducted in the scope of this project. It seems not trivial whether a stricter time-limit affects only the interestingness tests after a few “bad” transformations or all of the interestingness tests during the reduction. Since aborted test cases are considered as invalid, the entire reduction could be prevented in the latter case if no test succeeds within the given time.

Despite the success of the interestingness tests at least undefined behaviour as result of invalid arithmetic expressions [OCL12, § 6.12.3; C99, § 6.5.5] is not checked. Examples are signed integer overflow or the `clamp` function of OpenCL if the provided minimum is larger than the maximum. These sources of undefined behaviour cannot be detected with the static tools – except in trivial cases were the operands are constants – and Oclgrind currently provides no plugin to warn about them during the simulation of the kernel. If in the future such a plugin exists it can be activated for the interestingness tests to eliminate the risk of losing reduction results due these kinds of undefined behaviour.

Subsequently it has to be clarified to what extent undefined behaviour has to be reported and finally excluded from the test cases. At the moment the static tools report any occurrence of undefined behaviour and all warnings lead to an invalidation of the test case. In contrast, the undefined value plugin of Oclgrind warns only about undefined behaviour if it has visible side effects in terms of the control flow or the global resources. Thus the test cases can for instance operate with uninitialised values as long as the results of these operations do not influence the values of global variables. Though, strictly speaking this provokes undefined behaviour it might help during the reduction process to escape from local optima by allowing invalid intermediate steps that are unlikely to change the behaviour of the compiler. However, no guarantees in terms of the effects on the compilation process can be made for these situations. To find an answer to this question is beyond the scope of this project but should definitely be addressed in the future.

6 Conclusion

This project demonstrates that the principle of automatic test case reductions can be successfully applied in the area of many-core compilers. Moreover, the technique of compiler fuzzing has only recently been adopted to the OpenCL domain [Lid+15b] and a large number of defects can be found. This increases the importance of automatic reductions as it is nearly infeasible to perform manual reductions for all generated kernels.

The obtained results regarding the final size of reduced test cases and the runtime of reductions are comparable to prior evaluations of reductions of C programs [Reg+12]. The slightly larger size of reduced OpenCL kernels has been attributed to their different structure and an extensive use of structs. In the setting with parallel interestingness tests the runtime was similar to the times reported for C-Reduce in combination with the dynamic instrumentation framework KCC which has been replaced with the Oclgrind simulator for OpenCL kernels. These dynamic analysis tools are the main factor for the runtime of the interestingness tests. This project already performed several benchmarks to determine the best combination of tools and configurations for the interestingness test but especially the influence of reducing the number of work items in combination with parallel interestingness tests has to be analysed in greater detail.

In terms of the robustness of interestingness tests the results are also positive. Even though during the evaluations a few invalid test cases have been observed, it was no problem to adapt the tests to prevent further instances of this kind of undefined behaviour. The new Oclgrind plugin which has been developed during this project forms a large part of the check for undefined behaviour and improved significantly over the old plugin. If in the future more plugins are added for right now unhandled kinds of undefined behaviour such as arithmetic exceptions, it should be considered to combine them into one larger plugin to reduce their overhead.

Finally, the project confirmed the need for techniques to order automatically generated and reduced test cases according to the kind of bugs they are triggering. Without automatic preselection it is a time consuming task to look through the vast amount of reduced programs manually, only to find that most of test cases are duplicates. This problem has already been addressed by Chen et al. [Che+13] who developed methods to perform such an ordering. In future project these ideas should be evaluated for reduced OpenCL kernels as well. Furthermore, in the scope of this project the analysis of the course of transformations during the reduction process indicated a relation between the structure of a test case and its reduction itself. It might therefore be worth to evaluate whether the information about performed transformations can be used to group test cases.

A

Setup of the test environments

In the course of this project the reduction of OpenCL test cases has been tested on three different system. This chapter describes the steps which were necessary to set up the different environments, install the required tools and dependencies, and finally run the reductions. Hence it can be used as user manual or reference guide if the reduction framework has to be installed on a new system or if at some point in the setup something does not work as expected. If any problems have been encountered during the setup in the scope of this project they are mentioned here and solutions are described if available.

A.1 Linux

The central operating system for the automated test case reduction has been Linux during this project. The main reason is that nearly all the test machines are running with a Linux system and further most tools have been designed with Linux as their primary target. To make the test result reproducible this section describes how the tools needed for the automated test case reduction can be set up in a Linux environment.

A.1.1 Prerequisites

The instructions to configure all necessary tools have been tested on fresh Linux installations as well as on systems which already comprised different version of the required applications. As long as the correct versions are specified in the configuration steps no complications have been observed. The following tools are the basic requirements to build and install the programs for the automated test case reduction. It is thus recommended to check the availability of these tools first.

GCC Most of the programs have to be built from source. This requires at least GCC 4.8 or a compatible compiler.

Git Often only the current development versions of the tools contain important bugfixes or additions. Therefore it is recommended to checkout the sources directly from the respective Git repositories.

Perl The C-Reduce scripts are written in Perl to be portable across different platforms. To run the C-Reduce algorithm at least Perl version 5.10.0 is required.

Flex Some transformations are based on a lexical analysis of the test case to remove specific tokens or patterns. The lexer is part of the C-Reduce tool and thus has to be built from source. It is described in a special language to generate custom scanners. The format is interpreted by the *Flex* tool which converts the descriptions in compilable C sources. It can be downloaded from the project website[63] which requires to compile it manually or alternatively for instance on Ubuntu it is available through the "flex" package.

CMake On Linux LLVM and Clang can be configured via the provided Autotools configuration files but the *CMake* configuration offers more flexibility and is the recommended way to use.

Python The build system of LLVM and the interestingness depend on Python. While LLVM only requires Python 2 the interestingness tests have been written for Python 3 as some features are not present in the earlier version of the language.

OpenCL First of all the OpenCL header files and the OpenCL library are required to build the host program which launches the kernels during the reduction process. The header files can either be downloaded from the Khronos registry[64] or on Ubuntu systems they are also provided through the "opencl-headers" package. The library is most likely provided by the vendor of the GPU device but can alternatively also be downloaded from the Khronos registry and has to be built manually in this case. Lastly, to perform the reduction for an OpenCL capable device its OpenCL driver has to be installed.

[63] See `http://flex.sourceforge.net`, visited on 11/08/215.
[64] See `https://www.khronos.org/registry/cl/`, visited on 10/08/2015.

A.1.2 LLVM and Clang

LLVM and Clang are used in various situations of the test case reduction. On the one hand Clang is used in the interestingness test to check the validity of a given source code. Therefore it is first invoked as compiler and afterwards the static analysis features are used to detect potentially undefined behaviour which the less complete analysis during the compilation has not reported. On the other hand Oclgrind and clang_delta depend on these projects. Oclgrind uses the compiler infrastructure to generate the LLVM IR instructions and interprets them to simulate the execution of OpenCL kernels. The clang_delta tool relies on the AST generated by Clang to perform some language specific transformations.

It is recommended to use version 3.7 of LLVM and Clang which is the latest stable release.[65] The most flexible way is to clone the source of LLVM and Clang as Git repositories from the LLVM website.[66] For both repositories the branch "release_37" has to be checked out to build the release version 3.7 of LLVM and Clang. The Clang repository should be cloned into the `tools` subdirectory of the LLVM sources and also should be renamed to "`clang`". Only then the Clang compiler is automatically detected by the LLVM build system.

The alternative is to obtain the sources of LLVM and Clang from the "LLVM Download Page".[67] As for the Git repository the source directory of the Clang compiler has to be renamed to "`clang`" and moved into the "`tools`" subdirectory of the LLVM source tree. Only then the build system of LLVM will be able to automatically detect and build the Clang compiler.

To keep the source tree clean the configuration files should be created in an extra build directory. The recommended way of configuring the LLVM and Clang built is to use the provided CMake configuration files. To save space and to speed up the build process it should be configured as a release version and only the "X86" target should be selected. For instance, Unix Makefiles with these settings can be generated with the following `cmake` invocation

```
$ cmake -G "Unix␣Makefiles" -DCMAKE_BUILD_TYPE=Release \
        -DLLVM_TARGETS_TO_BUILD=X86 SRC_ROOT
```

where `SRC_ROOT` has to be substituted with the path to the top-level LLVM source directory. Additionally, the parameter `-DCMAKE_INSTALL_PREFIX` can

[65] Currently the development versions of LLVM and Clang are not compatible with the sources of C-Reduce (as of 25/09/2015). Therefore the latest stable release is recommended.

[66] See `http://llvm.org/docs/GettingStarted.html#git-mirror`, visited on 10/08/2015.

[67] See `http://llvm.org/releases/download.html#3.7.0`, visited on 21/08/2015.

be used to define a custom install location. The alternative to the CMake configuration is to invoke the `configure` script from the source directory of LLVM – preferably also out of a separate build directory.

The following instructions assume that the Clang compiler is globally reachable through the system `PATH` environment. If Clang has been installed to a custom location this has to be added to the `PATH` variable *before* any of the other directories which might contain a version of the executable. Otherwise the tools in the later steps in these instructions will pick up the wrong version of LLVM and Clang.

A.1.3 Oclgrind

The *Oclgrind* tool forms the basis of the interestingness tests. Through its capabilities as OpenCL simulator it can be used to perform a dynamic analysis of the kernel under reduction and warns about undefined behaviour. The only drawback is the long runtime on large kernels.

Like LLVM Oclgrind has to be build from source. The recommended way is to clone a forked version[68] of the Git repository and to checkout the “dev” branch as not all patches have been merged into the main project.[69] Especially the more precise plugin to detect uninitialised values (see Section 3.4) is needed to prevent undefined behaviour during the test case reduction.

When Oclgrind is built directly from the Git repository the `configure` script has to be created via `autoreconf -i` before it can be used. Again it is recommended to create a separate directory for the build files to keep the source tree clean. Possible configuration parameters are `--prefix` to set a custom path for the installation location, and `--with-llvm` and `--with-clang` respectively if LLVM and Clang are not contained in the `PATH` environment or if a dedicated version should be used. The provided CMake configuration files are primarily to support the build process on Windows systems and are not recommended as default build system.

A.1.4 CLsmith and cl_launcher

The CLsmith application is not a direct part of the automated test case reduction. But the purpose of this project was in particular to reduce those random kernels generated by CLsmith. The cl_launcher application on the contrary is used to launch the generated kernels in a standardised manner such that the reported result can be used to detected miscompilations of the OpenCL compilers.

The sources for both tools can be obtained as usual as Git repositories. Originally the cl_launcher program did not initialise the final result buffer which led to problems during the automatic reduction. Therefore a patched

[68] See `https://github.com/mpflanzer/Oclgrind`, visited on 10/08/2015.
[69] See `https://github.com/jrprice/Oclgrind`, visited on 10/08/2015.

version of the repository is made available.[70] To compile the new version of cl_launcher the "init" branch has to be checked out.

This time the `configure` script has to be called from inside the root directory of the cloned repository. Afterwards the first step is compile the underlying *Csmith* files by invoking `make` in the same directory. This is expected to fail with an error during the linking phase. Finally, the CLsmtih executable and cl_launcher can be built by calling `make` from the `src/CLsmith` subdirectory. To keep the interestingness tests simple it is recommended to copy the cl_launcher application to a location which is contained in the `PATH` environment. The CLsmith tool and its dependencies can be copied over to a custom location `~/TARGET` by invoking the `cl_setup_test.py TARGET` script from the `scripts` subdirectory.

A.1.5 C-Reduce

C-Reduce is rather a framework of different tools than a single application. But this also means that there is no simple "one-click" installation. Before the actual reduction algorithms can be configured some other dependencies have to be installed.

Dependencies

The following modules and programs are necessary before the C-Reduce tool can be configured. They are broadly available, probably even through a package manager and alternatively can be built from source.

Perl modules The C-Reduce reduction routine uses the following modules which are not all part of a standard Perl installation. The easiest way is to install them through the *CPAN Client* by invoking `cpan -i MODULE` where `MODULE` is one of the following modules names.

- Benchmark::Timer
- Exporter::Lite
- File::Which
- Getopt::Tabular
- Regexp::Common
- Sys::CPU

Artistic Style, Delta and Indent All three tools comprise features to reformat source code files according to different styles. They are used by C-Reduce to alter the "shape" of the input files between transformations to make the delta reduction routine more efficient. For instance, on Ubuntu the three tools are available through the packages "astyle", "delta" and "indent". The alternative is to download the sources[71] and build the tools manually.

[70] See `https://github.com/mpflanzer/CLSmith` for the forked repository while the original one can be accessed at `https://github.com/ChrisLidbury/CLSmith`, visited on 10/08/2015.

Operating System	Ubuntu 14.04.2 LTS
Processor	Intel® Core™ i7-4770
Clockrate	3.4 GHz
Virtual cores	8
RAM	16 GB

Table A.1 – Exemplary hardware information of the Linux machines. The presented information is exemplary for the used machines. The concrete models may vary.

C-Reduce

Finally, after all dependencies have been resolved, the actual C-Reduce tool can be built. While the most important changes to make C-Reduce compatible with OpenCL have been merged into the main project[72] some improvements regarding the individual transformations are only available through the forked version.[73] For the latest release version 3.7.0 of LLVM and Clang the "dev" branch is provided and contains all developed patches.[74]

The next step is to configure the C-Reduce build – a separate build directory is recommended – through the `configure` script. If LLVM cannot be detected automatically or if C-Reduce has to be built against an other than the default version the path can be specified via the `--with-llvm` parameter. A custom install location can be specified through the parameter `--prefix`. The configuration via the CMake build system is not recommended on Unix systems as the files do not comprise all features of the Autotools build and are intended to support builds on Windows.

Lastly, the generated Makefiles can be used to compile and install C-Reduce by invoking `make` and `make install`.

A.1.6 Configuration overview

During the project different Linux machines have been used to generate the test cases and also to run the reductions. Table A.1 lists exemplary the specifications for the used machines. Further, the different machines had access to different GPUs and driver versions. An overview over all GPUs used as OpenCL devices from the Linux machines is provided in Table A.2.

[71] See `http://astyle.sourceforge.net/`, `http://delta.tigris.org/` and `http://www.gnu.org/software/indent/`, visited on 11/08/2015.

[72] See `https://github.com/csmith-project/creduce`, visited on 11/08/2015.

[73] See `https://github.com/mpflanzer/creduce`, visited on 11/08/2015.

[74] For the current development sources the "llvm-svn-compatible" would have to be merged with the "opencl-extra" branch.

Device	*GeForce GT 630*
Platform	NVIDIA CUDA
Hardware version	OpenCL 1.1 CUDA
Software version	340.24
Address bits	32
Parallel compute units	1
Device	*GTX TITAN X*
Platform	NVIDIA CUDA
Hardware version	OpenCL 1.2 CUDA
Software version	352.21
Address bits	64
Parallel compute units	24
Device	*NVS 300*
Platform	NVIDIA CUDA
Hardware version	OpenCL 1.0 CUDA
OpenCL C version	OpenCL C 1.1
Software version	340.76
Address bits	32
Parallel compute units	2
Device	*GeForce GTX 980*
Platform	NVIDIA CUDA
Hardware version	OpenCL 1.1 CUDA
Software version	346.47
Address bits	32
Parallel compute units	16
Device	*Quadro K5200*
Platform	NVIDIA CUDA
Hardware version	OpenCL 1.1 CUDA
Software version	346.47
Address bits	32
Parallel compute units	12

Table A.2 – List of all OpenCL devices used from Linux machines. If the OpenCL C version is not specified explicitly it matches the hardware version.

A.1.7 Using C-Reduce

To reduce OpenCL test cases it is recommended to download the *libclc* project.[75] This prepares the Clang compiler to be able to parse OpenCL input and hence improves the reduction results of clang_delta. Moreover, the necessary interestingness tests are provided as a separate repository[76] containing various helper scripts to simplify the reduction process. Both repositories only need to be cloned, no further configuration is required.

The next step is to set up all environment parameters. An example configuration script is provided along with the interestingness tests. After the parameters have been adapted to the actual values the script can be "sourced" in the terminal to export all values. Depending on the actual task and the status of the input files not all parameters have to be specified. Yet, after following the instructions of this guide everything has been set up such that all parameters can be specified.

After the environment has been configured the reduction can be started as usual through the helper scripts (see Section 3.6.2) or directly by invoking `creduce` with a test script and the to-be-reduced file.

A.2 Chromebook 2

Out of the box the Chromebook 2 ships with Google Chrome OS and is intended as a fast and mobile office station or to surf in the internet. In this standard configuration the provided functionality is rather restricted. However, it is possible to boot a full-size Linux distribution on the Chromebook to make it better suited for development tasks. Further, OpenCL drivers are provided such that the two ARM® Mali™-T628 GPUs can be utilised as universal computing devices. The following sections describe the necessary steps to set up a test environment to run automated test case reductions. The steps to prepare the Linux system and to install the OpenCL drivers closely follow the guide from Anton Lokhmotov at the ARM Community.[77]

A.2.1 Preparing the Chromebook 2

First, the Chromebook has to be put into its *Developer mode* to enable shell access and booting from the external Micro SD card. To enter the Developer mode the ESC and the Refresh (F3) key have to be pressed while the Chromebook is powered on. This will start the *Recovery mode* of the Chromebook. From this screen the Developer mode can be activated by pressing Ctrl-D. Now, every time the Chromebook boots it will warn about

[75] See `http://libclc.llvm.org`, visited on 11/08/2015.
[76] See `https://github.com/mpflanzer/interestingness-tests`, visited on 11/08/2015.
[77] See `http://community.arm.com/groups/arm-mali-graphics/blog/2014/12/18/installing-opencl-on-chromebook-2-in-30-minutes`, visited on 05/08/2015.

```
#!/bin/sh
GENTOO_DIR=/home/chronos/user/gentoo
mount -t proc /proc $GENTOO_DIR/proc
mount --rbind /sys  $GENTOO_DIR/sys
mount --rbind /dev  $GENTOO_DIR/dev
cp /etc/resolv.conf $GENTOO_DIR/etc
```

(a) `setup.sh`

```
#!/bin/sh
GENTOO_DIR=/home/chronos/user/gentoo
LC_ALL=C chroot $GENTOO_DIR /bin/bash
```

(b) `enter.sh`

Listing A.1 – Scripts to setup and enter the Linux environment. The setup script has to be executed once after each reboot whereas the enter script is used each time a new shell should be created in the Linux environment.

the activated Developer mode. This warning can be skipped by pressing Ctrl-D, tough the process will also be continued automatically after waiting for 30 seconds.

A.2.2 Installing Gentoo Linux

Gentoo Linux has been chosen since it is a lightweight and highly customisable distribution which will easily fit into the limited space of an SD card. Moreover, everything is compiled from source such that most of the software will be up-to-date with the current development versions.

Initially the SD card has to be formatted with an `ext3` partition. This can be done on the Chromebook itself by opening the Chrome OS developer shell (crosh) in a new tab (Ctrl-Alt-T) and entering a proper shell with the `shell` command. From this shell the `fdisk` tool can be used to create the new partition on the Micro SD card.

Afterwards the Gentoo system can be downloaded and copied onto the SD card. The following instructions assume that the SD card has been mounted under `~/gentoo`. The Samsung Exynos processor of the Chromebook is based on the ARMv7 architecture and features a hardware floating point unit (FPU). Thus the latest Gentoo distribution with "hardfp" support should be downloaded[78] and extracted into the `~/gentoo` directory.

At this point the basic configuration of the new systems is completed. The two scripts in Listing A.1 can be used to set up and respectively enter the Linux environment. The setup script has to be run once after each reboot and the enter script is used to switch to the Linux environment from the shell embedded in the Chromebook. Both scripts can also be saved in the `~/gentoo` directory.

Before new software can be installed on the Gentoo system the package manager has to be configured. This is done by creating a directory tree as shown in Figure A.1 and running the commands from Listing A.2 to prepare and update the portage tool. After the initial setup installing a new package

[78] See `http://distfiles.gentoo.org/releases/arm/autobuilds/current-stage3-armv7a_hardfp/`, visited on 05/08/2015.

```
/etc/
└── portage/
    ├── make.conf
    ├── profile/
    ├── package.use/
    ├── package.unmask/
    ├── package.accept_keywords/
    └── package.keywords/
        └── dependences
```

(a) Portage directories and files

```
$ echo "MAKEOPTS=\"-j4\"" >> /etc/portage/make.conf
$ echo "ACCEPT_KEYWORDS=\"~arm\"" >> /etc/portage/make.conf
$ touch /etc/portage/package.keywords/dependences
```

(b) Configuration commands

Figure A.1 – Directory structure of the portage package manager. The file `make.conf` can already exists. In this case the first two commands just append to the file. Otherwise it gets created. The directories have to be created manually and the third command creates the empty file `dependences`.

should be straightforward by running the `emerge` command with the desired package as argument. Some packages have to be unmasked before they can be installed which requires two extra steps which is also explained in Listing A.2.

Optionally a new user can be created to maintain separate directories for user and system files. The user can be created with the `useradd` command, e.g. `useradd -m -G wheel,video moritz`. Since the Gentoo system is entered via the `chroot` command there is no real login system to change the current user. Instead the command `su - moritz` can be used to start a new login shell for the specified user.

```
$ emerge --sync --quiet
$ emerge --oneshot portage
$ eselect news read
```

(a) Update portage

```
$ emerge dev-vcs/git
$ emerge --autounmask-write dev-vcs/git
$ etc-update
$ emerge dev-vcs/git
```

(b) Install software

Listing A.2 – Updating portage and installing new packages. The update steps in (a) have to be run at least once before any package can be installed. To install a new package (b) the first `emerge` command will be enough most of the time. Only if this fails the package has to be unmasked first by running the commands 2 and 3. The `etc-update` command will prompt for the desired action which has to be confirmed with "-3" followed by "y".

```
KERNEL=="mali[0-9]", GROUP="video" MODE="0660"
```

Listing A.3 – The udev rule to change the group of the Mali GPU. Depending on the actual groups the user is member of the "video" label has to be adapted to a matching group. The rule has to be stored in a file named `10-mali.rules` in the directory `/etc/udev/rules.d/`.

It has to be noted that per default only the root user has access to the Mali GPU device in the Chromebook. This means either the OpenCL kernels have to be launched with the `sudo` command or the group of the Mali device is changed from "misc" to "video" or any other group which the new user belongs to.[79] To make the changes persistent over reboots a new udev rule can be created. For this purpose a new file `10-mali.rules` (Listing A.3) has to be created in `/etc/udev/rules.d/`. This file will be loaded each time the Gentoo environment is entered and changes the group of the Mali GPU to a group that is accessible to the user.

A.2.3 Enabling SSH access to the Chromebook

As alternative to working directly on the Chromebook or to copy files from and to the Chromebook it is possible to set up an SSH server. In an article[80] in the ARM® Connected Community Anton Lokhmotov describes a method to make configuration of the SSH server persistent over reboots and how to enable password based authentication. In addition to activating the Developer mode of the Chromebook this method requires to turn of the *Root Filesystem Verification* to make the main partition writeable. Instead Sébastien Santoro describes in his blog[81] how to start the SSH server without modifying the root system at the expense of less configuration options and having to restart it manually after a reboot. In this report the non permanent setup is described.

The SSH server is a direct feature of the Chromebook, not the Gentoo system, and thus can be enabled from the built-in developer shell of the Chromebook. If not already done a tab with the "crosh" shell has to be opened by pressing Ctrl-Alt-T and the developer shell must be entered by executing the `shell` command. Afterwards the SSH keys which will allow the public key authentication on the server can be created by running the following commands:

[79] Credits for this solution go to Krishnaraj Bhat who posted the necessary steps as a comment in the Mali Developer Community. See `http://community.arm.com/message/14394#14394`, visited on 05/08/2015.

[80] See `http://community.arm.com/groups/arm-mali-graphics/blog/2015/02/25/running-opencl-on-chromebook-remotely`, visited on 06/08/2015.

[81] See `http://goo.gl/s0aULG`, visited on 06/08/2015.

```
$ sudo mkdir -m 0711 /mnt/stateful_partition/etc/ssh
$ cd /mnt/stateful_partition/etc/ssh
$ sudo ssh-keygen -t rsa -f ssh_host_rsa_key
$ sudo ssh-keygen -t dsa -f ssh_host_dsa_key
```

As the name suggests, everything inside the `/mnt/stateful_partition` is preserved over a reboot so the keys for the server only have to be created once. Finally, the public key which will be used to login to the Chromebook has to be added to the `~chronos/.ssh/authorized_keys` file. Password based authentication is disabled per default for the SSH server and since the root partition is not writeable the configuration cannot be changed. Therefore only key based login is possible. The `authorized_keys` file is also not wiped during a reboot.

Lastly, incoming connections to port 22 have to be enabled and the SSH server has to be started:

```
$ sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
$ sudo /usr/sbin/sshd
```

The IP tables are reset during a reboot and the server will be shut down. This makes it necessary to rerun the above commands after each restart of the Chromebook.

To be able to close the lid of the Chromebook without that it enters the sleep mode the power management has to be deactivated. This can be done with the `stop powerd` command from inside the Developer shell. It has to be noted that this also disables the keys to regulate the display brightness and the power button. Therefore the display should be completely dimmed before the power management is turned off. Since the function keys are disabled one has either to reactivate the power management blindly by typing `start powerd` or via a remote SSH connection. Moreover, without a working power button simply restarting the Chromebook is not an option. Luckily the key combination ESC, Refresh (F3) plus power button still boots into recovery mode. This also reactivates the power management and one can turn the Chromebook off and on again to get back into the normal system.

A.2.4 Installing the OpenCL drivers

After entering the Gentoo environment the OpenCL dependencies can be installed. It is assumed that the next steps are performed as the root user. The OpenCL drivers are available from the Mali Developer Center.[82] On the website the drivers for the "Mali-T62x" on the "ARMv7" CPU have to

[82] See `http://malideveloper.arm.com/resources/drivers/mali-t6xx-gpu-user-space-drivers/`, visited on 05/08/2015.

be selected. After extracting the archive into a temporary location it has to be checked that all files are owned by root, belong to the group root and have the permissions set to 755. The prepared files have to be moved to the `/usr/lib` directory and the temporary directory can be removed. Finally, the OpenCL header files have to downloaded separately from the Khronos OpenCL registry[83] and saved in the (newly created) `/usr/include/CL` directory.

A.2.5 Installing CLsmith, C-Reduce and dependencies

After setting up the Gentoo distribution the Chromebook it behaves much like a normal Linux system. Thus the steps to install CLsmith, C-Reduce and all dependencies are basically the same as in Section A.1 and only deviations from the general install routine are explained in detail.

Prerequisites

The tools described in the following paragraphs are not directly or not exclusively necessary for the reduction of test cases but still need to be installed.

Git Most of the tools have to be built from sources which are provided as Git repositories. The Git application can be installed through the Gentoo package manager via `emerge dev-vcs/git`.

Perl The main reduction loop of the C-Reduce algorithm is implemented in Perl. At least version 5.10.0 is required but Gentoo should provide a fairly newer version via `emerge dev-lang/perl`.

Flex *Flex* is program to build custom scanners which can be used as parts of a more complex lexer. The scanner is described in a specific format which is interpreted by Flex and converted into compilable C source files. During the installation of C-Reduce a tool named *clex* will be built from those Flex input files. It is used to support transformations based on the removal of single tokens or patterns. The Flex tool can be installed through the command `emerge sys-devel/flex`.

CMake The easiest and recommended way to configure the build of LLVM and Clang is to use CMake. It can be installed with the command `emerge dev-util/cmake`.

[83] See `https://www.khronos.org/registry/cl/`, visited on 05/08/2015.

Python Both, LLVM and the interestingness tests depend on Python. While Gentoo has Python already installed per default it has to be checked whether Python 3 is installed. If not this can be done with `emerge dev-lang/python`.

LLVM, Clang and Oclgrind

There is nothing special about installing LLVM, Clang and Oclgrind on the Chromebook which means it is sufficient to follow the guidelines for general Linux systems.

CLsmith and cl_launcher

Also for CLsmith the guidelines for any Linux systems apply with the only difference that a different branch of the repository has to be checked out. The "chromebook" branch contains small changes which make the timing mechanism of Csmith compatible with the ARM architecture (see Section 3.1) and set compile time options to explicitly enable the hardware FPU and to link against the Mali libraries.

C-Reduce

In addition to `clang` and `clang-format` which have already been installed with the Clang compiler C-Reduce depends three more code formatting tools. They are all available through the Gentoo package manager and can be installed with the `emerge` command. The required packages are `app-text/delta`, `dev-util/astyle` and `dev-util/indent`.

The Perl modules and the C-Reduce tool itself can be installed in the same way as on any general Linux distribution and are thus not explained in more detail at this point.

A.2.6 Configuration overview

The host CPU of the Chromebook has only been used for the concrete test case reduction. The kernels have been generated and preprocessed on a more powerful machine although it would have been possible on the Chromebook as well. Table A.3 contains further details about the underlying hardware of the Chromebook. In addition to the Samsung Exynos processor the Chromebook 2 comprises two ARM Mali GPUs which can be used as OpenCL computing devices. More detailed information is listed in Table A.4.

A.2.7 Using C-Reduce

Again the process of starting a reduction of a test case on the Chromebook is not different from other Linux systems. Before the reduction can be started

Operating System	Gentoo Base System release 2.2 (Linux 3.8.11)
Processor	SAMSUNG EXYNOS5 (Flattened Device Tree)
Model name	ARMv7 Processor rev 3 (v7l)
Clockrate	1.9/1.3 GHz
Virtual cores	4 + 4
RAM	3.6 GB

Table A.3 – Hardware information of the Chromebook 2. The presented information has been collected from the Linux system reports and the datasheet for the "Exynos 5 Octa (5420)".

Device	*Mali-T628*
Platform	ARM platform
Hardware version	OpenCL 1.1
Address bits	64
Parallel compute units	4
Device	*Mali-T628*
Platform	ARM platform
Hardware version	OpenCL 1.1
Address bits	64
Parallel compute units	2

Table A.4 – List of all OpenCL devices on the Chromebook 2. For the test only the faster first device with four compute units has been used. Both devices cannot be used in parallel as they seem to share the memory and running both leads to runtime crashes of the OpenCL kernels.

the *libclc* project and the interestingness test helper scripts have to be cloned. After the environment has been configured the reduction can be started as usual through the helper scripts or directly by invoking `creduce`.

The only quirk on the Chromebook is that the clock and hence the shelf must be hidden during the execution of OpenCL kernels. Otherwise the clock enforces an update of the display through the GPU and any running user kernels are aborted. Empirically this seems to happen every 5 to 20 seconds. Remote login via SSH and turning off the display do not solve the problem.

A.3 Windows

During the project one goal was to get all the tools running under Windows as well. First this section describes the steps that have to be made to set up the complete test environment for the reduction of OpenCL kernels. This is followed by an overview over the complete test setup. The section is concluded with an example how C-Reduce can be used to reduce a test case.

A.3.1 Prerequisites

The instructions to set up the environment have been tested on a fresh installation of *Windows Server 2012 R2 (64 bit)*. The following tools are not directly necessary to run the test case reduction but are needed during the setup.

Visual Studio The easiest way to compile the necessary tools is to use the provided Visual Studio Solutions. During the project *Visual Studio 2013* has been used which can be freely obtained from the Microsoft website.[84] Any newer version should also be sufficient.

Git Most of the tools have to be compiled from source. The simplest way to get the sources is to clone them as Git repositories. Visual Studio provides a Git plug-in which provides basic features of Git. To avoid problems during compilation and execution the options `core.autocrlf = false` and `core.eol = lf` should be set in the Git configuration file. As there seems to be no possibility of doing this for the Visual Studio Git plug-in a good alternative is *Git for Windows*.[85] Furthermore this makes Git usable from the command line and behaves as on Unix systems.

Perl The C-Reduce algorithms depend on Perl. In this project the *Strawberry Perl*[86] distribution has been used. It is a Perl environment for Windows

[84] See `https://www.visualstudio.com`, visited on 01/07/2015.
[85] See `https://github.com/git-for-windows/git`, visited on 01/07/2015.

which ships with all necessary tools to run Perl scripts from the command line and to install additional Perl modules. All following instruction assume that the Perl executable is contained in the path environment – which is the default for Strawberry Perl.

Flex Some of the dependencies of C-Reduce need to be built from source and require the *Flex* program to be installed. This program is used to generate C source files for scanners defined through a collection of rules. For Windows Flex is provided as GnuWin32 package.[87] It is recommended to download the complete package with setup routine as it includes all the necessary dependencies. Furthermore, the `GnuWin32\bin` directory which contains the binaries for all packages should be added to the system path. This simplifies the configuration of the dependent projects.

CMake The project files for the tools which have to be compiled from source can be created from *CMake*[88] configuration files. During the installation the CMake installer provides an option to add CMake to the environment path. This option should be selected as otherwise the command line usage gets more complicated. CMake can be used from the command line as well as with a graphical user interface. Both methods will be explained in the instructions. The version 3.3.0-rc3 which has been used in this project seems to fail to detect a valid 64 bit compiler from the GUI. In this case the command line interface should be used which does not have this problem.

Python The build system of LLVM and the interestingness tests depend on a Python interpreter.[89] To the run the interestingness tests Python 3 is required. After the installation is finished the install path has to be added manually to the environment `Path` variable.

OpenCL To execute the OpenCL kernels in the tests during the reduction a working OpenCL driver has to be installed. Further, the OpenCL header files and the OpenCL libraries are needed to compile and link the host program which launches the kernels (see Section A.3.4). The files are specific to the actual vendor of the device under test. Thus no general advice can be given on how to install and set up the OpenCL framework.

[86] See `http://strawberryperl.com`, visited on 01/07/2015.
[87] See `http://gnuwin32.sourceforge.net/packages/flex.htm`, visited on 08/07/2015.
[88] See `http://www.cmake.org`, visited on 01/07/2015.
[89] See `https://www.python.org`, visited on 01/07/2015.

```
source-dir\
└── llvm\ ........................................ LLVM source directory
    ├── tools\
    │   ├── clang\ ............................... Clang source directory
    │   └── ...
    ├── build\ ........... Manually created directory for build files
    └── ...
```

Figure A.2 – Directory structure of LLVM and depended projects. The clang tools should be placed in the `llvm/tools` directory. This allows the CMake build files of LLVM to automatically detect the compiler.

A.3.2 LLVM and Clang

First of all the interestingness tests directly depend on Clang as compiler and static analyser to reject inappropriate reductions. Furthermore, Oclgrind depends on LLVM and Clang to generate the LLVM Intermediate Representation from the OpenCL kernels. The simulation of the kernels is based on these instructions. To provide better reduction results C-Reduce also comprises a tool named "clang_delta" which performs source to source transformations based on the AST of the program under reduction. To generate the AST LLVM and Clang are used.

The sources of the latest stable release for both tools can be cloned as Git repositories. Detailed instructions are provided in the LLVM documentation.[90] To build against the latest release, the branch "release_37" has to be checked out in both repositories. In addition to the Git repositories the sources of released versions are also made available as archives from the official LLVM website.[91]

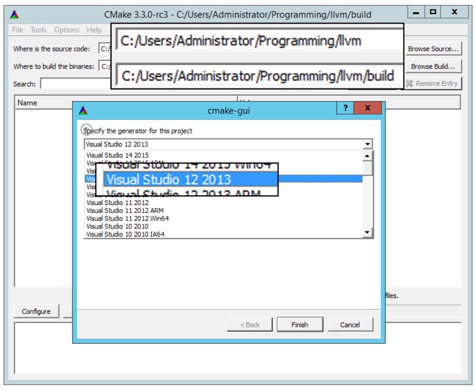
It is recommended to use version 3.7 of LLVM and Clang which is currently the latest stable release. Compared to earlier versions a bug in the code generation has been fixed (see Section 3.2) and – especially on Windows – during the project the tools clang_delta and Oclgrind were not always compatible with the latest contributions in the development branches.

The final directory structure should be similar to the example in Figure A.2. The source directory of the Clang compiler may has to be renamed to "`clang`" and should be copied into the "`tools`" subdirectory of LLVM. This ensures that the Clang compiler is automatically detected by the LLVM CMake build files. Moreover, a `build` directory has been created manually in which the configuration files will be created.

After changing into the newly created `build` directory CMake can be invoked from the command line with the following command:

[90] See `http://llvm.org/docs/GettingStarted.html#git-mirror`, visited on 01/07/2015.

[91] See `http://llvm.org/releases/download.html#3.7.0`, visited on 21/08/2015.


CMake 3.3.0-rc3 - C:/Users/Administrator/Programming/llvm/build

Where is the source code: C:/Users/Administrator/Programming/llvm

Where to build the binaries: C:/Users/Administrator/Programming/llvm/build

cmake-gui

Specify the generator for this project

Visual Studio 12 2013

Visual Studio 14 2015 Win64

Visual Studio 12 2013

Visual Studio 12 2013 ARM

Visual Studio 11 2012

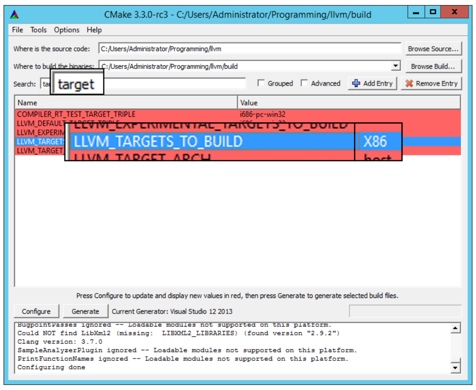
Visual Studio 11 2012 ARM

Visual Studio 11 2012 Win64

Visual Studio 10 2010

Visual Studio 10 2010 IA64

(a) Generator selection


CMake 3.3.0-rc3 - C:/Users/Administrator/Programming/llvm/build

Where is the source code: C:/Users/Administrator/Programming/llvm

Where to build the binaries: C:/Users/Administrator/Programming/llvm/build

Search: target

Name	Value
COMPILER_RT_TEST_TARGET_TRIPLE	i686-pc-win32
LLVM_EXPERIMENTAL_TARGETS_TO_BUILD	
LLVM_TARGETS_TO_BUILD	X86
LLVM_TARGET_ARCH	host

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 12 2013

```
BugpointPasses ignored -- Loadable modules not supported on this platform.
Could NOT find LibXml2 (missing:  LIBXML2_LIBRARIES) (found version "2.9.2")
Clang version: 3.7.0
SampleAnalyzerPlugin ignored -- Loadable modules not supported on this platform.
PrintFunctionNames ignored -- Loadable modules not supported on this platform.
Configuring done
```

(b) Target option

Figure A.3 – LLVM build settings in CMake GUI. In the top area the source and build path have to be set and in the options list the build target can be restricted to the x86 architecture.

```
> cmake -G "Visual␣Studio␣12␣2013" ^
        -DLLVM_TARGETS_TO_BUILD=X86 ..
```

It specifies to create a Visual Studio 2013 project for a 32 bit architecture.[92] Further it configures the `x86` architecture as the only build target to save time during the compilation and disk space. The project is configured for all build types and the desired type has to be selected later in Visual Studio. The same project files can be generated by starting the CMake program with its graphical user interface. In the top half of the window the paths to the LLVM source directory and the build directory have to be specified. Afterwards the project can be configured and on success the `LLVM_TARGETS_TO_BUILD` parameter can be changed. Finally the project files can be generated. An overview about the GUI is provided in Figure A.3.

In some cases the "MSBuild.exe" program crashed during the configuration phase of CMake. Although the exact reason could not be determined it is likely caused by testing for some unsupported features or by erroneous CMake configuration files in the development sources of LLVM. If a dialog box pop ups it is safe to confirm to close the crash executable. Afterwards the configuration proceeds normally. Even with a crash during the configuration no problems have been discovered in later stages of the build.

Once the generation of the Visual Studio project files finished they can be opened with the `LLVM.sln` solution file from the build directory. If the LLVM binaries should be installed later on it might necessary to run Visual Studio as Administrator depending on the install location. The solution file

[92] The identifier for the 64 bit version of Visual Studio is `Visual Studio 12 2013 Win64`.

```
clang version~3.7.0 (http://.../clang.git 40b68b4) (http://.../llvm.git dccade9)
Target: i686-pc-windows-msvc
Thread model: posix
```

Listing A.4 – Version information of the Clang compiler. The output has been emitted from the call to `clang --version`.

contains various projects for different parts and tools of the LLVM suite. They can either be built individually or all at once – which is recommended in the scope of this project.

The desired build configuration can be selected from the top toolbar in Visual Studio. As long as the compiler is only used as static analysis tool in the interestingness tests the "Release" configuration can be selected. On the other hand, if Clang is used for debugging or has to be debugged itself the "Debug" setting is much more useful.

Since the projects have to be built in a specific order it is not sufficient just to trigger a batch build for all projects from Visual Studio. Instead there are two special projects, namely "ALL_BUILD" and "INSTALL". The "ALL_BUILD" project just complies and links all binaries and places them into a directory matching the selected build configuration, e.g. `source-dir\llvm\build\Release\{bin,lib}`. This is also the project which is built when the complete solution is selected for building. The "INSTALL" project first invokes the "ALL_BUILD" target and installs the header files, libraries and binaries into the location to which the option `CMAKE_INSTALL_PREFIX` points to. The default location for programs is `C:\Program Files (x86)\LLVM` if the build has been configured for 32 bit or `C:\Program Files\LLVM` for a 64 bit target respectively. Finally, regardless of the selected method, the directory containing the binaries should be added to the `Path` environment variable.

If the installation has been successful the Clang compiler should now be available from the command line. For instance, the command

```
> clang --version
```

should produce an output similar to the one in Listing A.4. The version number should either be 3.7.0 for a build of the latest stable release or 3.8.0 if the development version has been built.

A.3.3 Oclgrind

The tool *Oclgrind* is used as dynamic analysis tool in the interestingness tests to check for undefined behaviour that has not been detected through the static analysers.

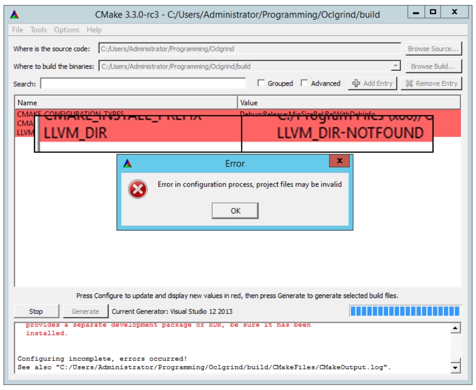
Oclgrind has to be built from source and can be obtained as a Git repository. The original repository is located at `https://github.com/jrprice/Oclgrind`. But, as of the 01/09/2015 the new plugin which has been developed during this project and the better control over emitted warning messages have not been merged. Therefore the forked version should be cloned from `https://github.com/mpflanzer/Oclgrind`. The development branch "dev" is kept up-to-date with the upstream master branch and additionally contains both improvements.

Like LLVM (see Section A.3.2) the project can be configured with the CMake environment to generate the Visual Studio project files. The selected architecture must match the one which had been selected for LLVM, i.e. a 32 bit Oclgrind application requires a 32 bit LLVM build and equally for 64 bit. Again a `build` subdirectory is manually created to keep the source tree clean.

In order to be able to build Oclgrind it must know about the LLVM source locations. Therefore the LLVM install location has to be included in the `Path` environment variable or the "LLVM_DIR" option has to be specified during the CMake configuration. This parameter must point to the directory which contains the CMake configuration file `LLVMConfig.cmake` for LLVM. If LLVM previously has been installed to the standard location the path will be `C:\Program Files (x86)\LLVM\share\llvm\cmake`. When CMake is invoked from the graphical user interface and the path has not been specified before the configuration is started it aborts with an error message (Figure A.4). After the path has then been set as value for the "LLVM_DIR" option the configuration can be restarted and the generation of the project files should succeed. Alternatively CMake can be invoked from the command line. In this case and if the LLVM installation has been added to the `Path` environment variable the "LLVM_DIR" parameter is automatically inferred. Otherwise it can be specified with the `-DLLVM_DIR="C:\Program␣Files␣(x86)\LLVM\share\llvm\cmake"` flag.

The solution file `Oclgrind.sln` for the project is created in the `build` directory. After opening it in Visual Studio – possibly as Administrator if Oclgrind will be installed to the default location (recommended) – the build configuration can be selected. To achieve a good performance with Oclgrind the "Release" build should be selected. Afterwards the application can be built with the "ALL_BUILD" project or built and installed with the "INSTALL" project. If the installation succeeds Oclgrind will be installed into `C:\Program Files (x86)\Oclgrind`. As Oclgrind will be invoked through the OpenCL ICD it is not necessary to add the directory containing the binaries to the `Path` environment variable.

On Windows the most convenient way is to register Oclgrind as an OpenCL platform on its own. This is done by adding keys to the Windows Registry. The name of the key is always the absolute path to the `oclgrind-rt-icd.dll` library, e.g. `C:\Program Files (x86)\Oclgrind\`



Where is the source code: C:/Users/Administrator/Programming/Oclgrind
Where to build the binaries: C:/Users/Administrator/Programming/Oclgrind/build

Name	Value
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/
LLVM_DIR	LLVM_DIR-NOTFOUND

Error in configuration process, project files may be invalid

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 12 2013

```
  provides a separate development package or SDK, be sure it has been
  installed.


Configuring incomplete, errors occurred!
See also "C:/Users/Administrator/Programming/Oclgrind/build/CMakeFiles/CMakeOutput.log".
```

Figure A.4 – Oclgrind build settings in CMake GUI. The parameter "LLVM_DIR" has to be set to the path containing the CMake configuration files for LLVM.

Application	Platform	Location
	32 bit	`*\Khronos\OpenCL\Vendors`
32 bit	64 bit	`*\Wow6432Node\Khronos\OpenCL\Vendors`
64 bit	64 bit	`*\Khronos\OpenCL\Vendors`

Table A.5 – Registry key locations for Oclgrind. The * has to be replaced with `HKEY_LOCAL_MACHINE\SOFTWARE`.

`lib\oclgrind-rt-icd.dll`. For 32 bit architectures in general and for 64 bit applications on 64 bit architectures a key of type `REG_DWORD` has to be created at `HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors`. For 32 bit applications on 64 bit architectures the location for the key is `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Khronos\OpenCL\Vendors`. Table A.5 summarises all possible combinations.

With these modifications Oclgrind is reported like any another OpenCL platform-device combination and can be used together with the general OpenCL API functions. Configuration options for Oclgrind have to be specified as environment variables.

A.3.4 cl_clauncher

The *cl_launcher* application is needed to run the automatically generated kernels. While in principle the kernels could be launched with any host program the cl_launcher tool automatically extracts metainformation from the kernel to launch it with the correct arguments. The source code for the application is included in the CLSmith Git repository.[93]

[93] See `https://github.com/ChrisLidbury/CLSmith`, visited on 07/07/2015.

The original version of cl_launcher does not initialise the global buffer before running the OpenCL kernel. This has led to wrong interestingness tests in cases where every write access to the output array has been removed in the reduction. In such situations the tests compare uninitialised values which results in false positives for wrong-code bugs. Instead of adding an extra check to the interestingness test to make sure at least one write is performed an alternative version of cl_launcher is available from the *init* branch of a forked Git repository.[94] There a patch has been applied through which the global buffer is initialised with zero values before it is used in the kernel.

The cl_launcher tool consists just of a single C-file and does only depend on the OpenCL headers files and library. Therefore it can be easily compiled and linked from a *Developer Command Prompt*[95] without the need of a large project setup. The actual command needed is the following:

```
> cl /I"%AMDAPPSDKROOT%\include" /FI stdbool.h cl_launcher.c ^
     /link /LIBPATH:"%AMDAPPSDKROOT%\lib\x86" OpenCL.lib
```

It has to be invoked from inside the `CLSmith\src\CLSmith` directory and produces the executable `cl_launcher.exe`. Depending on which OpenCL framework is used the include path and the library path have to be adapted. To eliminate the need to copy the application into every test directory or to specify always the absolute path it is useful to copy the executable into a location that is contained in the `Path` environment variable.

A.3.5 C-Reduce

Before the actual C-Reduce tool can be configured and built some other helper tools which are used during the reduction have to be installed. Some of them are distributed as pre-compiled binaries while a few have to be built from source.

Dependencies

The following paragraphs describe shortly how to obtain the necessary dependencies. For pre-built binaries it is explained how to install them properly. Additionally, for custom builds the required steps are outlined.

Perl modules To run the `creduce` script the following Perl modules are required. They can be installed from the *CPAN Client* by invoking `install`

[94] See `https://github.com/mpflanzer/CLSmith`, visited on 03/08/2015.
[95] Invoking the compiler and linker from the normal Command Prompt will likely not work. The Developer Command Prompt can be started from the `Visual Studio Tools` directory which has been installed alongside Visual Studio.

followed by the module name. Alternatively, the installation is possible from the command line by calling `cpan -i MODULE` where `MODULE` must be replaced with the actual module name.

- Benchmark::Timer
- Exporter::Lite
- File::Which
- Getopt::Tabular
- Regexp::Common
- Sys::CPU[96]

Artistic Style The intention behind *Artistic Style* is to "beautify" source code files to make them better readable for programmers. In the C-Reduce algorithms it is used to simply change the shape of the program to improve the simple delta reduction methods. A zip archive containing a pre-built binary for Windows can be downloaded from the Sourceforge project website.[97] The binary should be extracted into a directory which is included in the system path such that it can automatically be detected by other tools.

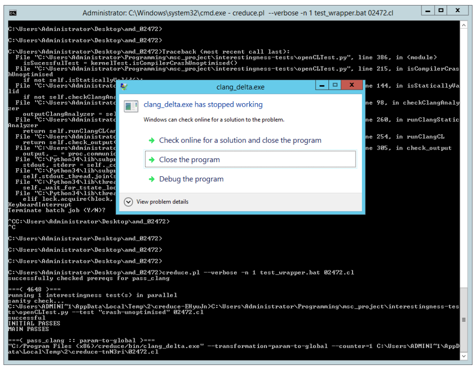
Indent Also *Indent* is used to reformat the test case during the reduction. It can be obtained as GnuWin32 package in the form of a pre-compiled binary.[98] It is recommended to download the complete installer as it also contains potential dependencies of this tool. After the installation the `GnuWin32\bin` directory which contains the binaries for all packages should be added to the system path. This simplifies the configuration of the dependent projects.

Delta From the Delta project only the *topformflat* tool is used by C-Reduce. It is the third tool that reformats source code. In contrast to the previous two tools it does not enforce a specific code style but collapses the contents of the source file depending on the nesting level of the code. The official website does not provide the tool as binary but for the purpose of this project a release version can be downloaded from the created GitHub project.[99] As for the other tools the binary should be included in the system path environment. Instead of using the pre-built release the source code can be cloned from the repository. The project contains CMake configuration file that can be used to build and install the files of the Delta project.

C-Reduce

Finally, *C-Reduce* can be installed as the actual reducer for the generated test cases. It has to be built from source and is available as GitHub project.[100]

[96] The "Sys::CPU" module is an optional dependency and it is likely that it will not work on Windows.
[97] See `http://astyle.sourceforge.net`, visited on 08/07/2015.
[98] See `http://gnuwin32.sourceforge.net/packages/indent.htm`, visited on 08/07/2015.
[99] See `https://github.com/mpflanzer/delta`, visited on 08/07/2015.
[100] See `https://github.com/mpflanzer/creduce`, visited on 08/07/2015.



Administrator: C:\Windows\system32\cmd.exe - creduce.pl --verbose -n 1 test_wrapper.bat 02472.cl

```
C:\Users\Administrator\Desktop\amd_02472>

C:\Users\Administrator\Desktop\amd_02472>

C:\Users\Administrator\Desktop\amd_02472>Traceback (most recent call last):
  File "C:\Users\Administrator\Programming\msc_project\interestingness-tests\openCLTest.py", line 386, in <module>
    isSucessfulTest = kernelTest.isCompilerCrashUnoptimised()
  File "C:\Users\Administrator\Programming\msc_project\interestingness-tests\openCLTest.py", line 215, in isCompilerCras
hUnoptimised
    if not self.isStatically
  File "C:\Users\Administrato                                                             ne 144, in isStaticallyVa
lid
    if not self.checkClangAna
  File "C:\Users\Administrato                                                             ne 98, in checkClangAnaly
zer
    outputClangAnalyzer = sel
  File "C:\Users\Administrato                                                             ne 260, in runClangStatic
Analyzer
    return self.runClangCL(ar
  File "C:\Users\Administrato                                                             ne 254, in runClangCL
    return self.check_output(
  File "C:\Users\Administrato                                                             ne 305, in check_output
    output, _ = proc.communic
  File "C:\Python34\lib\subpr
    stdout, stderr = self._co
  File "C:\Python34\lib\subpr
    self.stdout_thread.join(s
  File "C:\Python34\lib\threa
    self._wait_for_tstate_loc
  File "C:\Python34\lib\threa
    elif lock.acquire(block,
KeyboardInterrupt
Terminate batch job (Y/N)?

^CC:\Users\Administrator\Desktop\amd_02472>
^C

C:\Users\Administrator\Desktop\amd_02472>

C:\Users\Administrator\Desktop\amd_02472>

C:\Users\Administrator\Desktop\amd_02472>

C:\Users\Administrator\Desktop\amd_02472>creduce.pl --verbose -n 1 test_wrapper.bat 02472.cl
successfully checked prereqs for pass_clang

===< 4648 >===
running 1 interestingness test(s) in parallel
sanity check...
C:\Users\ADMINI~1\AppData\Local\Temp\2\creduce-EHyuJn>C:\Users\Administrator\Programming\msc_project\interestingness-tes
ts\openCLTest.py --test "crash-unoptimised" 02472.cl
successful
INITIAL PASSES
MAIN PASSES

===< pass_clang :: param-to-global >===
"C:/Program Files (x86)/creduce/bin/clang_delta.exe" --transformation=param-to-global --counter=1 C:\Users\ADMINI~1\AppD
ata\Local\Temp\2\creduce-tnN3ri\02472.cl
```

clang_delta.exe

clang_delta.exe has stopped working

Windows can check online for a solution to the problem.

Figure A.5 – Example of a clang_delta crash dialog box. This example shows the blocking dialog box that Windows presents if a program crashes. In this case clang_delta was not able to handle its input due to a bad reduction.

For LLVM and Clang version 3.7.0 the "dev" branch should be checked out since it contains some OpenCL related patches which have not been applied to the main repository.[101]

After the repository has been cloned the project files can be generated with CMake as usual. If everything has been set up correctly no manual interaction is needed to generate the Visual Studio project files. Otherwise CMake warns about missing dependencies and how to fix them. The "creduce" project must be built with the same configuration which had been selected for LLVM, e.g. Release Win32. It is recommended to use the "INSTALL" target since it distributes the various files such they find each other. To make the `creduce.pl` script available from everywhere the install location should be added to the `Path` environment variable.

A.3.6 Windows configuration

During the execution of C-Reduce some programs might crash either because a bug has been triggered or because the reduction produced invalid input. In the default configuration Windows reacts to a program crash with presenting a dialog box to the user to ask how to handle the problem (Figure A.5). Especially in an automated test setting this can be annoying since the test execution will stop until an option has been selected.

Thus it is recommended to disable the dialog box completely. The necessary steps to deactivate the pop-up dialog are described and illustrated in an article on *Raymond.cc*.[102] In summary, the tool *gpedit.msc* allows

[101] This branch is not compatible with the current development sources of LLVM and Clang. If they have to be used the branch "llvm-svn-compatible" of C-Reduce has to be merged with the "opencl-extra" branch to create a new development branch of C-Reduce.

[102] See `https://goo.gl/bgUhHS`, visited on 10/07/2015.

Operating System	Microsoft Windows Server 2012 R2 Datacenter
Hypervisor	Xen HVM domU
Processor	Intel® Xeon® E5-2690 v2
Clockrate	3.00 GHz
Virtual cores	6
RAM	10 GB

Table A.6 – Exemplary hardware information of the Windows VMs. All updates have been installed for the operating system.

to edit the "Windows Group Policies". After starting the tool one has to navigate through the panels "Computer Configuration", "Administrative Templates" and "Windows Components" to "Windows Error Reporting". There the option "Prevent display of the user interface for critical errors" can be enabled.

A.3.7 Configuration overview

The following section provides an overview over the Windows machines which where available for developing the Windows variant of C-Reduce and the actual reduction of generated and miscompiled kernels.

Windows VM

The main development and testing of the Windows variant of C-Reduce itself has been done on several virtual machines running with Windows Server 2012 R2. Table A.6 contains further details about the underlying hardware. Testing the correct setup and interaction of all programs on a virtual machine is beneficial as the VM can be reset to an earlier configuration or to a fresh install. This allows to validate setup routines and configurations without side effects from earlier trials.

The Intel® Xeon® processor itself has been used as OpenCL device. While it is not as powerful as dedicated GPUs it has been sufficient to test all settings. The OpenCL drivers for the Intel® Xeon® processor are provided through Intel's *OpenCL™ Runtime 15.1*.[103] The header files and the library can be obtained by installing *Intel® INDE*.[104] Alternatively AMD's *AMD APP SDK 3.0 Beta*[105] does also provide OpenCL drivers and the needed files. Lastly, if only the header files and the library are needed to compile and

[103] See `https://software.intel.com/en-us/articles/opencl-drivers`, visited on 06/07/2015.

[104] More information about Intel® Integrated Native Developer Experience (Intel® INDE) is provided at `https://software.intel.com/en-us/intel-inde`, visited on 07/07/2015.

[105] See `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`, visited on 06/07/2015.

Device	*Oclgrind Simulator*
Platform	Oclgrind
Vendor	University of Bristol
Hardware version	OpenCL 1.2 (Oclgrind 15.5)
Device	*Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz*
Platform	Intel(R) OpenCL
Vendor	Intel(R) Corporation
Hardware version	OpenCL 1.2 (Build 57)
Platform	Experimental OpenCL 2.0 CPU Only Platform
Vendor	Intel(R) Corporation
Hardware version	OpenCL 2.0 (Build 162)
Platform	AMD Accelerated Parallel Processing
Vendor	Advanced Micro Devices, Inc.
Hardware version	OpenCL 1.2 AMD-APP (1642.5)

Table A.7 – List of all OpenCL platforms on the Windows VMs. The Oclgrind platform has been used to validate the generated kernels itself whereas the actual test case reduction has been performed for the other platforms.

link the host application the files can also be downloaded from the *Khronos OpenCL Registry*.[106] All available OpenCL platforms are listed in Table A.7.

A.3.8 Using C-Reduce

Before a reduction can be started two more dependencies have to be obtained. Because the Clang compiler itself does not know about the types and functions introduced through OpenCL additional headers file have to be provided. The *libclc* project[107] is compatible with OpenCL 1.1 and provides declarations for the newly defined types and functions. Further, interestingness tests for reductions of OpenCL kernels are required. During this project a collection of interestingness tests for OpenCL kernels has been developed which can be used on Linux and Windows since they are written in Python. The test script as well as some helper scripts to start the reduction and to prepare the kernels are available as Git repository.[108] The repositories of both dependencies can be cloned into any convenient location and no further interaction is required.

[106] See `https://www.khronos.org/registry/cl/`, visited on 07/07/2015. The header files are directly available from the website and the OpenCL library has to be built from the ICD Loader project.
[107] See `http://libclc.llvm.org`, visited on 13/07/2015.
[108] See `https://github.com/mpflanzer/interestingness-tests`, visited on 14/07/2015.

Starting a reduction of a test case with C-Reduce does not need more than a few commands if everything has been set up as describes in the previous sections. Some of the necessary configuration parameters have to be specified through environment variables. The example script `set_test_env.bat` which is provided together with the interestingness tests does not only contain these variables but also those that can optionally be specified through the environment. The values have to be adapted to the actual ones for the given machine that is used. Afterwards the script can be invoked from the command line to make the values available to the following steps.

After everything has been set up the best way to start a reduction is to use the `findMiscompilations.py` script as described in Section 3.6.2. Alternatively the `creduce.pl` script can be invoked manually. The advantage of using the Python wrapper is that not all parameters have to be specified as environment variables but can be passed as additional flags.[109]

[109] For more detailed information about the available flags please see the usage information produced by `findMiscompilations.py --help`.

B

Tool overview

This chapter describes the basic features of the programs which have been used in the course of this project. The focus is thereby on the usage of the tools and the configuration options they provide. For details about the design of the tools it is referred to Chapter 2. Further, only those options are presented which had already been implemented at the beginning of the project. Information about options that have been added in the scope of this project are provided alongside the implementation details in Chapter 3.

B.1 CLsmith and cl_launcher

The CLsmith tool is the generator for random kernels whereas cl_launcher is the host application which can be used to launch the kernels. Both tools can be configured though various command line options [Lid+15a].

B.1.1 CLsmith

The CLsmith generator provides several *modes* which can be used to control which features should be included in the generated kernels (Table B.1). They are simply passed as parameters on the command line. The options which have been used to generate the kernel and a few other parameters are also written into the source files of the kernels. This allows the cl_launcher application to configure the execution environment according to the selected modes. Further, the seed of the random number generator used by CLsmith can be specified. This allows to reproduce the exact same kernels of a previous run.

The generated kernels depend on the header files `safe_math_macros.h`, `cl_safe_math_macros.h` and `CLSmith.h`. These files comprise helper functions to compute thread and work group ids and define macros around

Mode switch	Description
`--atomic_reductions`	Includes atomic operations that involve an element in a buffer and a local variable in the kernels
`--atomics`	Includes critical sections in the kernels which are guarded by an atomic expression such that the threads enter the section deterministically
`--barriers`	Includes memory barriers in the generated kernels. Each barrier is surrounded by a read from one element in a buffer and a write to a different element
`--divergence`	Makes the control flow dependent on the *linear global id* of a thread such that not all threads execute the same instructions
`--fake_divergence`	Includes expressions in the kernel that seem to diverge but evaluate to the same expression during the runtime
`--group_divergence`	Makes the control flow dependent on the *linear group id* of a thread such that not all work groups execute the same instructions
`--inter_thread_comm`	Includes instructions in the generated kernels through which the threads exchange their computed values during the runtime
`--vector`	Includes vector typed expressions and vector instructions in the generated kernels

Table B.1 – Kernel mode switches of CLsmith. The different modes can be passed as command line options to the CLsmith tool to include different features in the generated kernels. The options `--barriers` and `--divergence` are mutually exclusive.

certain arithmetic functions to prevent undefined behaviour in the kernels. To execute the kernels the header files have to be in the same directory as the kernel source code file.

B.1.2 cl_launcher

While in principle it would be possible to run the kernels from any host application the cl_launcher tool automatically reads configuration parameters from the source code file of the kernels and sets up the execution environment as well as the data structures required by the kernels. For example are the global and the local size coupled with the size of the output buffer passed to the kernel and some modes require additional kernel arguments.

In addition to the parameters which are already included in the kernel source code file itself cl_launcher provides the command line options `-p` and `-d` to customise the platform and the device respectively on which the kernel file specified with the option `-f` should be executed. Moreover, the flag `-n` can be used to verify that the name of the selected device matches a specific string.

Further optional parameters are the `---debug` option to enable a more verbose output on failures during the preparation or execution of the kernel and the `---disable_opts` option to deactivate the optimisation passes of the OpenCL compiler of the selected device.

B.2 Oclgrind

The usage of Oclgrind differs between Unix like operating systems and Windows. On the former systems dynamic library preloading is used to reroute the OpenCL library calls to the Oclgrind implementations whereas on Windows Oclgrind has to be registered as regular OpenCL device.

B.2.1 Linux

To run a kernel with Oclgrind instead of the real OpenCL device it is sufficient to prepend the call to the host program with `oclgrind` or the absolute path to the `oclgrind` script respectively. Additional configuration parameters can be passed to the `oclgrind` script via the command line or by setting environment variables directly. A possible invocation could look like the following:

```
$ oclgrind --data-races ./cl_launcher -p 0 -d 0 -f CLProg.c
```

In addition to launching kernels via the original host program Oclgrind provides the `oclgrind-kernel` application which acts like a host program and does not depend on the library preloading. It expects an Oclgrind

```
Bug_14.cl
entry
1 1 1
1 1 1

<size=8 fill=0 dump>
```

Listing B.1 – Example for an Oclgrinf simulation file. The first line is interpreted as the path to the actual kernel file and the second line has to consist of the name of the top level kernel function that should be called by the host application. The next two lines describe the global and local size for the kernel execution and all following lines specify which arguments have to passed to the kernel function.

specific file which describes the simulation parameters as input and accepts (mostly) the same command line options as the `oclgrind` script.

Since the randomly generated kernels in this project have to be started via the cl_launcher application anyway, the format of the simulation file is not further described in this report. Instead is it referred to an article[110] in the Wiki of the Oclgrind GitHub website which explains everything in detail. Here only a short example of such a simulator file is presented in Listing B.1.

B.2.2 Windows

Since the library preloading mechanism is not available on Windows systems Oclgrind has to be registered as regular OpenCL platform-device combination. The host application must than use the device number of Oclgrind to launch the kernel. Since there is no direct way to pass arguments to configure the Oclgrind implementation they have to be specified through environment variables before the kernel is started. An exemplary invocation could be the following:

```
> SET OCLGRIND_DATA_RACES=1
> .\cl_launcher -p 1 -d 0 -f CLProg.c
```

Configuration options

Most of the options that Oclgrind provides are used to activate specific plugins or to control the output during the execution. An overview over all options available in the original version of Oclgrind can be found in the Wiki of the GitHub project.[111] The subset of the standard parameters that have

[110] See `https://github.com/jrprice/Oclgrind/wiki/Running-Kernels-in-Isolation`, visited on 20/08/2015.

Parameter	Description
`--build-options`	This parameter can be used to pass options through to the internal invocation of the Clang compiler. In the scope of this project it has been used to disable optimisations via the `-cl-opt-disable` switch
`--data-races`	Activates the plugin that checks whether data races would occur during the execution of the kernel
`--uninitialized`	Activates checks for invalid uses of undefined values. This option has been used before the new plugin had been implemented

Table B.2 – Used standard configuration options of Oclgrind. The table contains only those options which are provided by the original version of Oclgrind. Additional options are described in their corresponding sections.

been used during this project is summarised in Table B.2. Further options have been added through the contributions to the Oclgrind project. Those are described in the corresponding sections of Chapter 3.

B.3 C-Reduce

As mentioned earlier C-Reduce is not a single application but comprises different tools and components. This section describes possible configuration options for all modules that are required during a test case reduction.

B.3.1 creduce

The `creduce` script itself does not provide many customisable options. The two mandatory command line parameters are first the path to a script which is used as interestingness test and second the path to the file which will be reduced. The optional flag `-n` can be used to specify the maximal amount of transformations that should be executed in parallel (see Section 2.1.4). It defaults to the number of (virtual) cores in the processor if the `Sys::CPU` Perl module is available. Otherwise only one transformation is run at a time. Further, the switch `--verbose` enables more detailed output during the reduction. For instance, in verbose mode the result of every transformation is emitted.

To configure which transformations should actually be performed by C-Reduce one has to edit the `creduce` script by hand. All transformations are listed in a hash table and can easily be removed. There exist also command

[111] See `https://github.com/jrprice/Oclgrind/wiki/Using-Oclgrind#options`, visited on 06/07/2015.

line options to disable a few specific groups of transformations but they do not offer a fine-grained control. Another advantage of modifying the script directly is that the order of the transformations during the reduction can be altered by changing the priorities of the transformations.

B.3.2 clang_delta

The clang_delta tool need not to be used explicitly if one is only interested in the reduction of source files. However, during the project it has become necessary to test specific transformations in isolation. For instance, to patch the discovered bugs with out running the complete C-Reduce algorithm every time the transformation result had to be checked.

The tool operates in two modes and depending on the mode either the number of possible applications of the given transformation is computed or the transformation is actually performed. The first mode is activated by passing the option `--query-instances=TRANSFORMATION` through the command line. For the second mode the desired transformation has to be selected via the `--transformation=TRANSFORMATION` flag and additionally the option `--counter=NUM` specifies which instance of the transformation should be applied. Specifying a number other than one is equivalent to advancing the state of the transformation as described in Section 2.1.4. In both modes the file name of source file on which the transformation should be applied is mandatory. A common invocation of clang_delta could be the following:

```
$ ./clang_delta --transformation=remove-unused-var \
                --counter=1 CLProg.c
```

B.4 Interestingness tests

The *interestingness* of a particular variant is determined by a user defined (shell) script which has to return zero if the transformed source is still valid and interesting, e.g. free from undefined behaviour but still triggers the bug. Any other return value indicates a failed transformation.

The name of the file under reduction has to be hard-coded in the test script[112] and all references to programs or files outside of the script have to be specified as absolute paths since the file containing the source code and the test script are copied over to a temporary location prior to each transformation.

[112] The documentation of C-Reduce states that the name of the test case is not provided for the test script (`https://embed.cs.utah.edu/creduce/`, visited on 20/08/2015). However, it turned out that in the actual implementation the path to the test case is the first (and only) argument passed to the test script.

```
clang_ocl CLProg.cl > out_clang.txt 2>&1 &&\
! grep 'warning: zero size arrays are an extension'\
       out_clang.txt > /dev/null 2>&1 &&\
! grep "excess elements in " out_clang.txt > /dev/null 2>&1 &&\
! grep "address of stack memory associated with local variable"\
       out_clang.txt > /dev/null 2>&1 &&\
clang_static_analysis CLProg.cl > out_clang_static.txt 2>&1 &&\
! grep 'warning: Assigned value is garbage or undefined'\
       out_clang_static.txt > /dev/null 2>&1 &&\
! grep 'warning: Dereference of null pointer'\
       out_clang_static.txt > /dev/null 2>&1 &&\
timeout 60 gpuverify --local_size=1 --global_size=1 --stop-at-opt\
                     CLProg.cl > out_verifier.txt 2>&1 &&\
! grep 'warning: control reaches end of non-void function'\
       out_verifier.txt > /dev/null 2>&1 &&\
! grep "uninitialized" out_verifier.txt > /dev/null 2>&1 &&\
timeout 60 ./cl_launcher -p 0 -d 0 -f CLProg.cl\
           > out_opt.txt 2> /dev/null &&\
timeout 60 ./cl_launcher -p 0 -d 0 -f CLProg.cl ---disable_opts\
           > out_noopt.txt 2> /dev/null &&\
diff out_opt.txt out_noopt.txt > /dev/null 2>&1
```

Listing B.2 – Example for an interestingness test. First the warnings emitted from the Clang compiler are checked for indicators of undefined behaviour. Then the tools "Clang Static Analyzer" and "gpuverify" are utilised. In the end the kernel is run with and without optimisations and the results are compared. As soon as one of the tools exits with a non-zero status the test fails immediately. For a successful test the results between optimised and unoptimised execution have to be different.

Typical interestingness tests first validate the syntactical correctness of the new source code by checking compiler warnings and the output of static analysers to achieve a fast fail on bad transformations. More time expensive analysis steps and the actual execution of the new program to check for the desired (mis-)behaviour are postponed towards the end of the test (Listing B.2). Furthermore, the test script should contain a timeout after which a failed test is reported. Otherwise the reduction could get stuck if for instance the transformed test case contains an endless loop and hence its execution would never finish.

C

Additional Listings

```
void MemCheck::checkArrayAccess(const WorkItem *workItem,
                                const llvm::GetElementPtrInst *GEPI) const
{
    // Iterate through GEPI indices
    const llvm::Type *ptrType = GEPI->getPointerOperandType();

    for(auto opIndex = GEPI->idx_begin(); opIndex != GEPI->idx_end(); opIndex++)
    {
        int64_t index = workItem->getOperand(opIndex->get()).getSInt();

        if(ptrType->isArrayTy())
        {
            // Check index doesn't exceed size of array
            uint64_t size = ptrType->getArrayNumElements();

            if((uint64_t)index >= size)
            {
                ostringstream info;
                info << "Index␣("
                    << index << ")␣exceeds␣static␣array␣size␣("
                    << size << ")";
                m_context->logError(info.str().c_str());
            }

            ptrType = ptrType->getArrayElementType();
        }
        else if(ptrType->isPointerTy())
        {
            ptrType = ptrType->getPointerElementType();
        }
        else if(ptrType->isVectorTy())
        {
            ptrType = ptrType->getVectorElementType();
        }
        else if(ptrType->isStructTy())
        {
            ptrType = ptrType->getStructElementType(index);
        }
    }
}
```

Listing C.1 – Checking `getelementptr` for out-of-bounds conditions. The function sequentially walks through all indices of the instruction. Everytime the index accesses an object of array type the number of elements of the array is queried and compared to the index. If the index is larger or equal to the number of elements a warning is emitted.

```
TypedValue ShadowFrame::getValue(const llvm::Value *V) const
{
    if (llvm::isa<llvm::Instruction>(V)) {
        // For instructions the shadow is already stored in the map.
        assert(m_values->count(V) && "No␣shadow␣for␣instruction␣value");
        return m_values->at(V);
    }
    else if (llvm::isa<llvm::UndefValue>(V)) {
        return ShadowContext::getPoisonedValue(V);
    }
    else if (llvm::isa<llvm::Argument>(V)) {
        // For arguments the shadow is already stored in the map.
        assert(m_values->count(V) && "No␣shadow␣for␣argument␣value");
        return m_values->at(V);
    }
    else if(auto *VC = llvm::dyn_cast<llvm::ConstantVector>(V))
    {
        TypedValue vecShadow = ShadowContext::getCleanValue(V);
        TypedValue elemShadow;

        for(unsigned i = 0; i < vecShadow.num; ++i)
        {
            elemShadow = getValue(VC->getAggregateElement(i));
            size_t offset = i*vecShadow.size;
            memcpy(vecShadow.data + offset, elemShadow.data, vecShadow.size);
        }

        return vecShadow;
    }
    else
    {
        // For everything else the shadow is zero.
        return ShadowContext::getCleanValue(V);
    }
}
```

Listing C.2 – Mapping shadow values to data values. The shadows of interemediate results and function arguments are simply looked up in the map containing all shadow values. The special `undef` value is mapped to a poisoned shadowed and constant vectors are scanned componentwise for `undef` values to obtain are precise shadow representation. Finally, all other values, i.e. constants, are mapped to a clean shadow.

```
case llvm::Instruction::Call:
{
    const llvm::CallInst *callInst = ((const llvm::CallInst*)instruction);
    const llvm::Function *function = callInst->getCalledFunction();

    // Check for indirect function calls
    if (!function)
    {
        // Resolve indirect function pointer
        const llvm::Value *func = callInst->getCalledValue();
        const llvm::Value *funcPtr = ((const llvm::User*)func)->getOperand(0);
        function = (const llvm::Function*)funcPtr;
    }

    assert(!function->isVarArg() && "Variadic␣functions␣are␣not␣supported!");

    // For inline asm, do the usual thing: check argument shadow and mark all
    // outputs as clean. Note that any side effects of the inline asm that are
    // not immediately visible in its constraints are not handled.
    if (callInst->isInlineAsm())
    {
        checkAllOperandsDefined(workItem, instruction);
        shadowValues->setValue(instruction,
                               ShadowContext::getCleanValue(instruction));
        break;
    }

    if(auto *II = llvm::dyn_cast<const llvm::IntrinsicInst>(instruction))
    {
        handleIntrinsicInstruction(workItem, II);
        break;
    }

    if(function->isDeclaration())
    {
        if(!handleBuiltinFunction(workItem, function->getName().str(),
                                  callInst, result))
        {
            // Handle external function calls
            checkAllOperandsDefined(workItem, instruction);

            if(callInst->getType()->isSized())
            {
                // Set return value only if function is non-void
                shadowValues->setValue(instruction,
                                       ShadowContext::getCleanValue(instruction));
            }
        }
        break;
    }

    assert(!llvm::isa<const llvm::IntrinsicInst>(instruction) &&
           "intrinsics␣are␣handled␣elsewhere");

    // Fresh values for function
    ShadowFrame *values = shadowValues->createCleanShadowFrame();

    llvm::Function::const_arg_iterator argItr;
    for (argItr = function->arg_begin(); argItr != function->arg_end(); argItr++)
    {
        const llvm::Value *Val = callInst->getArgOperand(argItr->getArgNo());
        if (!Val->getType()->isSized())
        {
            continue;
        }

        if(argItr->hasByValAttr())
        {
            assert(Val->getType()->isPointerTy() &&
                   "ByVal␣argument␣is␣not␣a␣pointer!");
            // Make new copy of shadow in private memory
            size_t origShadowAddress = workItem->getOperand(Val).getPointer();
            size_t newShadowAddress = workItem->getOperand(argItr).getPointer();
            ShadowMemory *mem = shadowWorkItem->getPrivateMemory();
            unsigned char *origShadowData =
              (unsigned char*)mem->getPointer(origShadowAddress);
            size_t size = getTypeSize(argItr->getType()->getPointerElementType());

            // Set new shadow memory
            TypedValue v = ShadowContext::getCleanValue(size);
            memcpy(v.data, origShadowData, size);
            allocAndStoreShadowMemory(AddrSpacePrivate, newShadowAddress,
                                      v, workItem);
            values->setValue(argItr, ShadowContext::getCleanValue(argItr));
        }
        else
        {
            TypedValue newShadow = shadowContext.getMemoryPool()->clone(
              shadowContext.getValue(workItem, Val));
            values->setValue(argItr, newShadow);
        }
    }

    // Now, get the shadow for the RetVal.
    if(callInst->getType()->isSized())
    {
        values->setCall(callInst);
    }

    shadowValues->pushFrame(values);

    break;
}
```

Listing C.3 – Handling of `call` instructions in the ShadowKepper plugin. First indirect function calls are resolved and it is asserted that the function is not a variadic function. Afterwards inline assembler is handled in a general way by checking the shadow values of all arguments. Intrinsic functions and built-in functions are handled separately from other external functions as Oclgrind knows their side effects and can perform a precise shadow propagation. Finally, for internally defined functions a new shadow frame is created, initilised with the shadows of all function arguments and pushed onto the stack of shadow values. Further, the call instruction is stored in the dedicated slot in the shadow frame if the function has a non-void return type.

```
void ShadowMemory::allocate(size_t address, size_t size)
{
    size_t index = extractBuffer(address);

    if(m_map.count(index))
    {
        deallocate(address);
    }

    Buffer *buffer = new Buffer();
    buffer->size   = size;
    buffer->flags  = 0;
    buffer->data   = new unsigned char[size];

    m_map[index] = buffer;
}
```

Listing C.4 – Lazy deallocations in the ShadowKeeper plugin. Since there are no explicit instructions to deallocate memory the deallocations are instead performed on demand. If a new allocation is issued for an address which is already associated with a buffer object it is deallocated first. Since Oclgrind itself ensures the consistency of the data memory system this strategy cannot cause problems.

```
bool ShadowContext::isCleanStruct(ShadowMemory *shadowMemory, size_t address,
                                  const llvm::StructType *structTy)
{
    if(structTy->isPacked())
    {
        unsigned size = getTypeSize(structTy);
        TypedValue v = {
            size,
            1,
            m_workSpace.memoryPool->alloc(size)
        };

        shadowMemory->load(v.data, address, size);

        return isCleanValue(v);
    }
    else
    {
        for(unsigned i = 0; i < structTy->getStructNumElements(); ++i)
        {
            size_t offset = getStructMemberOffset(structTy, i);
            unsigned size = getTypeSize(structTy->getElementType(i));

            if(const llvm::StructType *elemTy =
                 llvm::dyn_cast<llvm::StructType>(structTy->getElementType(i)))
            {
                if(!isCleanStruct(shadowMemory, address + offset, elemTy))
                {
                    return false;
                }
            }
            else
            {
                TypedValue v = {
                    size,
                    1,
                    m_workSpace.memoryPool->alloc(size)
                };

                shadowMemory->load(v.data, address + offset, size);

                if(!isCleanValue(v))
                {
                    return false;
                }
            }
        }

        return true;
    }
}
```

Listing C.5 – Special check for undefined values in structs. Structs can legally contain undefined padding values. Therefore the simple check if all bits are clean produces false positives in the general case and can only be applied if the struct is marked as "packed", i.e. it is not padded. Otherwise padding areas are skipped by iterating over all members of the struct and performing the check recursively. If any of the members contains undefined values other than padding values of potentially nested structs the entire top-level struct is considered as poisoned.

```
if(addrSpace == AddrSpaceGlobal)
{
    shadowContext.getGlobalMemory()->lock(address);
}

loadShadowMemory(addrSpace, address, oldShadow, workItem);

if(!ShadowContext::isCleanValue(argShadow) ||
   !ShadowContext::isCleanValue(oldShadow))
{
    newShadow = ShadowContext::getPoisonedValue(4);
}
else
{
    newShadow = ShadowContext::getCleanValue(4);
}

storeShadowMemory(addrSpace, address, newShadow, workItem);

if(addrSpace == AddrSpaceGlobal)
{
    shadowContext.getGlobalMemory()->unlock(address);
}
```

Listing C.6 – Locking mechanism for general atomic operations. Only when the atomic operations affect the global address space explicit locking is required. Then the lock is acquired before the old shadow value is read and released after the new value has been written. In between the new shadow value is computed by checking the argument of the atomic operation and the old shadow value for their definedness.

```
// -g 1,1,1 -l 1,1,1
void transparent_crc_no_string(ulong *p1, ulong p2) { *p1 += p2; }

uint get_linear_global_id() {
  return (get_global_id(2) * get_global_size(1) + get_global_id(1)) *
           get_global_size(0) +
         get_global_id(0);
}

struct U0 {
  int f0;
};
struct S1 {
  volatile ushort g_300[2][4][7];
  short g_953[8];
};
void func_1() { struct U0 a, b = a; }

__kernel void entry(__global ulong *p1) {
  int i, j, k;
  struct S1 c, e = {{{{0xC55AL}}}};
  struct S1 *d = &c;
  c = e;
  ulong f = i = 0;
  for (; i < 2; i++) {
    j = 0;
    for (; j < 4; j++) {
      k = 0;
      for (; k < 7; k++) {
        transparent_crc_no_string(&f, d->g_300[i][j][k]);
      }
    }
  }
  i = 0;
  for (; i < 8; i++) {
    transparent_crc_no_string(&f, d->g_953[i]);
  }
  p1[get_linear_global_id()] = f ^ 0xFFFFFFFFFFFFFFFFUL;
}
```

Listing C.7 – Average sized reduced test case. The reduced test case has a size of 844 bytes which is close to the average of the evaluated reductions. It is free from undefined behaviour and triggers a wrong-code bug on the device "dev-a". The correct result is "0xffffffffffff3aa5" but with optimisations enabled the computed output is "0xffffffffffff3aa4".

Bibliography

[C++11] ISO/IEC JTC 1/SC 22. *ISO/IEC 14882:2011. Programming languages – C++*. Standard. Geneva, Switzerland: International Organization for Standardization, 2011.

[Bar+14] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew and Shaz Qadeer. 'Engineering a Static Verification Tool for GPU Kernels'. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 226–242. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_15.

[Bet+15] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson and John Wickerson. 'The Design and Implementation of a Verification Technique for GPU Kernels'. In: *ACM Trans. Program. Lang. Syst.* 37.3 (2015), p. 10. DOI: 10.1145/2743017.

[Bon+07] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer and Kathryn S. McKinley. 'Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors'. In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 405–422. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297057.

[C11] ISO/IEC JTC 1/SC 22. *ISO/IEC 9899:2011. Programming languages – C*. Standard. Geneva, Switzerland: International Organization for Standardization, 2011.

[C99] ISO/IEC JTC 1/SC 22. *ISO/IEC 9899:1999. Programming languages – C*. Standard. Geneva, Switzerland: International Organization for Standardization, 1999.

[Che+13] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide and John Regehr. 'Taming Compiler Fuzzers'. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 197–208. ISBN: 978-1-4503-2014-6. DOI: `10.1145/2491956.2462173`.

[Gov10] Darryl Gove. *Multicore Application Programming: For Windows, Linux, and Oracle Solaris.* Developer's Library. Addison-Wesley, 2010. ISBN: 9780321711373.

[LAS14] Vu Le, Mehrdad Afshari and Zhendong Su. 'Compiler Validation via Equivalence Modulo Inputs'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 216–226. ISBN: 978-1-4503-2784-8. DOI: `10.1145/2594291.2594334`.

[Lib+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken and Michael I. Jordan. 'Scalable Statistical Bug Isolation'. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 15–26. ISBN: 1-59593-056-6. DOI: `10.1145/1065010.1065014`.

[Lid+15a] Christopher Lidbury, Andrei Lascu, Nathan Chong and Alastair F. Donaldson. *CLSmith tool description.* 2015. URL: `http://multicore.doc.ic.ac.uk/tools/CLsmith/PLDI15/CLSMITH_USAGE.pdf` (visited on 18/08/2015).

[Lid+15b] Christopher Lidbury, Andrei Lascu, Nathan Chong and Alastair F. Donaldson. 'Many-core Compiler Fuzzing'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2015. Portland, OR, USA: ACM, 2015, pp. 65–76. ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737986`.

[LLVM15] *LLVM Language Reference Manual.* Version 3.8. 2015. URL: `http://llvm.org/docs/LangRef.html` (visited on 08/08/2015).

[NS07a] Nicholas Nethercote and Julian Seward. 'How to Shadow Every Byte of Memory Used by a Program'. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments.* VEE '07. San Diego, California, USA: ACM, 2007, pp. 65–74. ISBN: 978-1-59593-630-1. DOI: `10.1145/1254810.1254820`.

[NS07b] Nicholas Nethercote and Julian Seward. ‘Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation’. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: `10.1145/1250734.1250746`.

[OCL12] Khronos OpenCL Working Group. *The OpenCL Specification*. Specification. Version 1.2. Khronos Group, 2012.

[Reg+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison and Xuejun Yang. ‘Test-case Reduction for C Compiler Bugs’. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 335–346. ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254104`.

[Reg12] John Regehr. *Parallelizing Delta Debugging*. 11th July 2012. URL: `http://blog.regehr.org/archives/749` (visited on 19/08/2015).

[Sar95] Gurusamy Sarathy. *perlfork – Perl’s fork() emulation*. 1995. URL: `http://perldoc.perl.org/perlfork.html` (visited on 04/08/2015).

[SN05] Julian Seward and Nicholas Nethercote. ‘Using Valgrind to Detect Undefined Value Errors with Bit-precision’. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA: USENIX Association, 2005.

[SS13] Evgeniy Stepanov and Konstantin Serebryany. ‘MemorySanitizer’. In: European LLVM Conference. (29th Apr. 2013). Paris, France, 2013. URL: `http://llvm.org/devmtg/2013-04/stepanov-slides.pdf` (visited on 18/08/2015).

[SS15] Evgeniy Stepanov and Konstantin Serebryany. ‘MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++’. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’15. San Francisco, California: IEEE Computer Society, 2015, pp. 46–55. ISBN: 978-1-4799-8161-8.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide and John Regehr. ‘Finding and Understanding Bugs in C Compilers’. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993532`.