Institut für Programmstrukturen
und Datenorganisation (IPD)

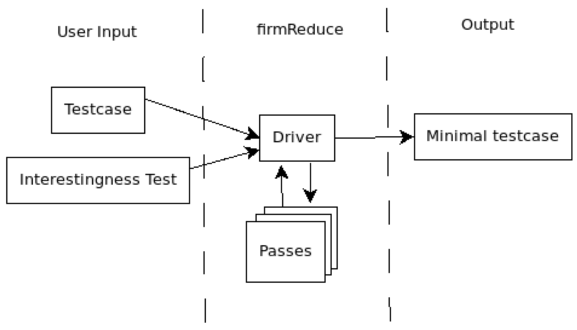
Lehrstuhl Prof. Dr.-Ing. Snelting

FirmReduce: Automated Test-Case Reduction for Graph-Based Compilers

Bachelorarbeit von

Tina Maria Strößner

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Sebastian Graf

Abgabedatum: 7. September 2018

Abstract

Testcase reduction is the process of isolating the failure-inducing parts of a given larger testcase, with the goal of obtaining a testcase, minimal in size, that reveals the same faulty behaviour in the system under test.

This thesis presents FirmReduce , a program to automate testcase reduction for testcases given in the Firm intermediate presentation that can be used to test and debug the libFirm compiler backend. It is based on the findings given in *Test-Case Reduction for C Compiler Bugs* [1]. We will show that the reducer FirmReduce has the capability to substantially reduce a given testcase in size and present reduction results using different bugs we identified in libFirm .

Zusammenfassung

Testfallreduzierung beschreibt die Isolation von versagensauslösenden Programmteilen in großen Testprogrammen mit dem Ziel, die Größe dieses Tesfalls zu minimieren, sodass dieser immernoch das Fehlverhalten des zu testenden Systems offenlegt.

Diese Arbeit präsentiert das Programm FIRMReduce zur automatischen Reduktion von Testfällen, die in der Compiler-Zwischensprache FIRM gegeben sind, das zum Testen und Debuggen des libFIRM Compiler-Backends verwendet werden kann. Wir werden zeigen, dass FIRMReduce in der Lage ist, die Größe von Testprogrammen erheblich zu reduzieren und präsentieren die Ergebnisse von Reduktionen mit Testfällen für in libFIRM vorhandenen Bugs.

Contents

1 Introduction

Compilers generate machine code from source code written in higher-level programming languages. The correctness of the produced executables therefore depends heavily on the compiler being able to translate the input to a semantically equivalent program. Bugs in compilers inhibit this ability and cause the compiler to produce wrong machine code or even to crash.

Bugs in compilers are often discovered by generating random test programs, using a fuzzer, or by chance, when using the compiler to build an executable. Either way, the resulting test programs, which cause the compiler to show faulty behaviour, are often quite large and therefore make it difficult to identify which part of the testcase provokes this behaviour. In order to make the debugging process easier, it is desirable to have a testcase of minimal size, triggering the bug in question.

A common approach to find a minimal testcase isolating the bug, is to start with a larger testcase and successively reduce its size, while maintaining its bug-triggering properties. Examples of such reduction frameworks include the Delta Debugging algorithm used in different applications or C-Reduce, a reducer that works on C programs developed at the University of Utah.

libFIRM is a library providing a compiler optimization suite and a backend used for code generation. It implements FIRM, a graph-based intermediate representation, used to represent programs in SSA-form as a basis for the subsequent optimizations and code generation. A flexible test suite is used to maintain libFIRM 's code quality and prevent software regression. The testcases used in the suite can be generated using FirmSmith [2], a fuzzer producing programs directly in FIRM. This makes the testcase generation independent of the compiler frontend and source language features and achieves a higher coverage of libFIRM features. However, this approach prevents the use of all common testcase reducers, as they do not operate on FIRM.

Based on the methods used in C-Reduce, this thesis aims to provide a testcase reduction framework, that works exclusively on FIRM graphs to improve the debugging process for the libFIRM library and make testcase reduction for FIRM graphs possible. Secondly, the reduction program shall be used to analyze and isolate bugs, currently found in libFIRM .

The main contributions of this thesis are as follows:

- We present a possible design for a modular reducer based on the graph-based intermediate representation Firm.
- We determine which graph transformations are most useful to minimize a given testcase. Furthermore we identify synergy effects between different graph transformations as an important part of the reduction process and present how they can best be exploited to increase efficiency of the reduction.
- We show that our reducer can match the results of existing tools, while allowing working directly on the intermediate representation. This makes it possible to omit the compiler's frontend from the whole debugging process if required, which is not possible with other testcase reducers.

2 Background

Compilers are used to translate programs from source code to architecture-specific machine code. The translation process is split into three different phases:

First, the compiler frontend analyzes the input and checks for syntactical and semantical correctness. Afterwards the program is transformed into an intermediate representation that is independent of the source and target language. The intermediate representation of a program is the result of the frontend's source code analysis and the basis for the middle-end's optimizations. The choice of data structure for the IR directly influences the efficiency and quality of a compiler's transformation [3]. Optimizations are applied to the program in an effort to improve the program's performance. It is essential for these optimizations to each output a program that is semantically equivalent to the input. Lastly, the compiler backend is responsible for the code generation [4].

This chapter will introduce the basics of the Firm intermediate representation, as well as give a more comprehensive explanation of the testcase reduction problem.

2.1 Firm

libFirm is a C library, providing optimizations and machine code generation, based on a graph-based intermediate representation Firm. Firm was developed in 1996 at the Karlsruhe Institute of Technology, as part of the Sather-K compiler Fiasco, but was later extracted to the separate library libFirm [5]. Firm is a completely graph-based representation of a program in SSA form. SSA form ensures, that every variable is defined before it is used and is assigned exactly once. A variable that is assigned more than once, is split into multiple variables. At points in the program where two control control flow paths join, a ϕ function is used, that chooses the right definition of the variable, depending on the control flow of the program [6].

2.1.1 Firm Graphs

A program in Firm is represented by mulitple graphs, each representing a function in the program. The nodes of a graph represent either a basic block, a transfer of the control flow or a data flow operation. Basic blocks are the vertices in a control flow graph and represent pieces of code, that are executed sequentially with exactly one entry and one exit point. Control flow is transferred to a different basic block on operations like (conditional) jumps. Data flow operations are operations that change the state of memory, such as Stores, Loads or Allocate operations.

The edges of a Firm graph represent control flow dependencies. A control flow dependency between two nodes exists, if the execution of the operation represented by node A is dependent on the result of the execution of node B. A dependency of this nature would be represented in a Firm graph by an outgoing edge from node A pointing to node B. The control dependency edges are reversed compared to a control flow graph. A special case are memory dependencies, where the value creating the dependency is in the memory and can therefore not be duplicated. A memory operation would have the whole memory state as input and produce a new memory state as output.

Firm Nodes

The transformations applied to a Firm graph by FirmReduce often target specific nodes in the graph. Below we provide a description of all nodes that are targeted.

`Alloc node` These nodes allocate a block of memory on the stack. Its inputs are a memory node and the size of the block in bytes. Output is the resulting memory state and a pointer to the newly allocated memory.

Arithmetic nodes Arithmetic nodes in the context of this thesis are `Add`, `Sub`, `Mul`, `Div`, `Mulh`, `Mod`, `Shr`, `Shrs`, `Shl nodes`. They all have two inputs (the operands of the arithmetic operation) and one output (the result of the arithmetic operation). The two operands and the result may have different types depending on the operation.

`Call node` This node represents a function call. It has the current memory state input as well as a pointer to the called code. Additionally, function parameters of the called functions may be predecessors to this node. Successors are the resulting memory state, a tuple value containing the result values of the call and the control flow successors.

`Cond node` A conditional change to the control flow. The input is the condition parameter ("selector").

`Load node` Loads a value either from heap or stack. The input parameters are the current memory state and a pointer to the address to load from. The output consists of the resulting memory state and the result of the load operation.

`Mux node` A `Mux node` has three predecessors: Two operands and one selector. One of the operands is returned based on the value of the selector.

`Proj node` In Firm every SSA value corresponds to a node. This can become a problem for operations that return multiple results. To avoid this, these operations instead return a single tuple value. Tuple values also appear after function calls that return both the call result, as well as the resulting memory state. From this tuple, the components can be projected using a `Proj node`. A `Proj node` has the tuple value as its input and outputs the value it projects.

`Return node` This node represents the *return* statement of a function. It takes the function's final memory state, as well as all return values as input.

`Store node` Stores a value either to heap or stack. It has three inputs: The current memory state, the address to store at and the value it is supposed to store. The output is the resulting memory state.

`Switch node` A node which changes the control flow depending on the value of its input selector. Successors to this node are all possible destinations of the control flow change.

2.1.2 Optimizations

Compilers not only translate programs from a source language into machine code, they also perform different analyzes and optimizations. Optimizations are transformations that maintain semantics, while improving the program in regard to execution time and memory usage. libFirm implements 17 different compiler optimizations that are used in FirmReduce to advance the reduction process. [7]

2.2 Testcase Reduction

While a general description of the problem of testcase reduction was given in the introduction, this chapter will provide a more formal definition.

2.2.1 The Testcase Reduction Problem

Bugs in compilers are often discovered by compiling program that triggers said bug and causes the compiler to crash or miscompile said program. Finding out which part of the program is responsible for triggering the bug is vital for understanding and ultimately fixing the bug, but can be a tedious task if the program is large. Testcase reduction is trying to solve said problem. In the following chapter, we adapt and expand the definitions given in [1] in order to create a formal foundation upon which we can base our work in this thesis.

Let $\mathcal{T}$ be the set of all valid inputs to some system under test. An element $t \in \mathcal{T}$ is called a testcase. Let $\mathcal{I}$ be the set of predicate $i : \mathcal{T} \to \{True, False\}$, that map a testcase to a boolean value. A testcase $t \in \mathcal{T}$ is considered an *interesting testcase* (or *bug reproducer*), iff it fulfills the given predicate $i \in \mathcal{I}$, i.e. $i(t) = True$. Otherwise it is considered *uninteresting.*

Let $\mathcal{T}_i \subseteq \mathcal{T}$ be the set of all interesting testcases for a given predicate I and for all $t \in \mathcal{T}_I$ let $|t|$ be the testcase's size according to some appropriate metric (for the metric used in this thesis, refer to section 4.1.1). The testcase reduction problem for a given compiler bug is to find $t_{min} \in \mathcal{T}_I$, where $\forall t \in \mathcal{T}_I : |t_{min}| \leq |t|$. t_{min} might not be unique. We call t_{min} the *minimal interesting testcase.*

The process of finding a *minimal interesting testcase* is based on the production of variants of a given interesting input testcase t. Let $\mathcal{P} \subseteq \mathcal{T} \to \mathcal{T}$ be a set of transformations, where each transformation $p \in \mathcal{P}$ manipulates a given input testcase t in a predefined way, and outputs a different testcase t'. We write $t \Rightarrow_p t'$. Additionally, we write $t \Rightarrow_{\mathcal{P}} t'$, iff $t \Rightarrow_p t'$ for an arbitrary $p \in \mathcal{P}$. We assume $id \in \mathcal{P}$.

Definition 1. t' is called *variant of* t iff t' lies within the reflexive-transitive hull of t w.r.t. $\Rightarrow_{\mathcal{P}}$.

Depending on the initial interesting testcase t used to find variants, it is possible, that the *minimal interesting testcase* r_{min} we are looking is not a *variant of* t. Instead of finding an unrelated interesting testcase t', where r_{min} is a *variant of* t', we will focus on finding the *minimal interesting variant:*

Definition 2. For a given initial testcase t and a compiler bug predicate $i \in \mathcal{I}$, let $\mathcal{V}_{t,i} := \{t' \,|\, t' \text{ is variant of } t \wedge t' \text{ is interesting w.r.t } i\}$. We call t' a *minimal interesting variant* iff $t' \in \mathcal{V}_{t,i} \wedge \forall t'' \in \mathcal{V} : |t'| \leq |t''|$.

The *minimal interesting variant* might not be unique.

In order to check if an interesting variant is also minimal, we use **??** 1. How it is used is explained in section 4.1.2.

Theorem 1. *Let t' be an interesting variant for a given initial testcase t and w.r.t a given bug i in the compiler. t is a minimal interesting variant iff $\forall$ variants t'' of t with $|t''| < t'$: t'' is not interesting.*

Proof. The theorem results from the contraposition of the definition of the minimum for a given partial order. In this case, the partial order in question is given by the chosen metric $| \cdot |$. $\square$

2.2.2 Testcase Minimization vs Testcase Reduction

The terms 'testcase reduction' and 'testcase minimization' are not always used uniformly. We use the definitions given in [1], where 'testcase reduction' refers to the transformation of a single given testcase (i.e. finding the minimal interesting variant), whereas 'testcase minimization' means finding the absolute minimal interesting testcase.

Other definitions, such as 'testcase reduction' being the minimization of number of testcases, while preserving the same code coverage, are not applicable in this thesis.

3 Related Work

Existing reduction tools use different approaches to minimize the size of a testcase. One of the first tools to minimize testcases automatically is the delta debugging algorithm. Another commonly referred to method is the algorithm used in C-Reduce, which is also the basis for the work done in this thesis. Both approaches are described in the following chapter.

3.1 Delta Debugging

The Delta Debugging algorithm was first described in 1999 at the university of Saarland. The motivation behind the development of the methodology stems from the *Mozilla Gecko BugAThon: A call for volunteers*, where Mozilla was asking volunteers to help to simplify the many open bug reports listed in the Mozilla bug database [8].

ddmin The minimizing Delta Debugging algorithm *ddmin* minimizes a testcase by removing characters from the original test case and testing whether Mozilla crashes on the given test case. The result is a minimal failing test case, where every character is relevant to reproducing the Mozilla crash.

dd The general Delta Debugging algorithm *dd* in contrast tries to isolate the differences between a failing and a passing test case. Additionally to minimizing the failing input, *dd* also tries to maximize the passing input until a minimal failure-inducing difference is obtained.

While Delta Debugging was first developed for and tested with HTML input, it is not limited to such and the general concept can be applied to "all circumstances that in any way affect the program execution" [9].

3.2 C-Reduce

C-Reduce is a test case reduction tool for testing C compilers that was developed at the university of Utah. It is a modular reducer, that uses a series of transformations, that are iteratively applied to the program until a fixpoint in the reduction is found. These transformations, in contrast to the Delta Debugging methodology are not hard-coded, but implemented as plug-ins to the main program driver. *C-Reduce* uses five types of transformations [1]:

Peephole optimizations A peephole optimization is an optimization that is performed over a small contiguous chunk of the code. The optimizations used in *C-Reduce* include changing identifiers and integer constants to `0` or `1`, removing a basic block or constant folding.

Localized, but non-contiguous changes These transformations mostly affect the control flow of the program by merging or separating basic blocks. As an example, this can be achieved by removing an if statement plus the following pair of curly braces without altering the block of code between them.

Line Removal Instead of applying transformations to specific source code elements, these transformations remove lines from the source code, without taking into account their semantic value to the program. The number of lines removed initially is the number of lines in the test case. This number is successively reduced until it reaches the minimum of one line removed.

External Pretty-Printing Reformatting can turn a failure-inducing test case into a non-failure-inducing testcase, which is why this step is included in the reduction itself and not post-processing.

Compiler-like transformations Using LLVM's Clang front end, a set of compiler-like transformations were added to the reducer, including dead-code elimination, function inlining or the scalar replacement of aggregates.

C-Reduce uses the number of lines in a program as its metric to identify the minimal testcase. This metric may not always deliver the best testcase for easy bug analysis, since the number of lines doesn't necessarily correlate to the complexity of a program.

4 Design and Implementation

The reduction strategy used by FirmReduce is based on iteratively producing and evaluating different variants of the initial testcase given as input. A variant is a testcase derived from the input program by applying a transformation, such as substituting constant values or omitting parts of control flow where possible. Variants are evaluated according to their size and bug reproducibility and either kept for further reduction or discarded.

FirmReduce has a modular structure, consisting of the driver, numerous passes, as well as a user-supplied interestingness test (see figure 4.1).

The driver manages the different parts of the program. Its responsibility is invoking the different passes and the interestingness test script, as well as evaluating the newly found variants according to their size and determining if a fixpoint is found and FirmReduce should terminate. The passes' responsibility is the reduction itself, by each creating new variants from the input given to them. Checking whether a variant is still a reproducer of the bug is done by a interestingness test. The criteria that a variant must fulfill in order to pass the test depends on the bug we want to reproduce. The test can therefore not be hardcoded into the program, but is supplied by the user in form of a shell script that is executed repeatedly during the reduction process.

The algorithm used by the driver to apply passes and the interestingness test is shown in line 31, a more detailed explanation of the responsibilities and functionalities of FirmReduce 's different modules is given in the following sections.

FirmReduce is programmed in C and uses libFirm version 1.22.1. Additionally, the interestingness tests used for the reductions use `cparser` (version 1.22.1), a C99 parser and frontend for libFirm .

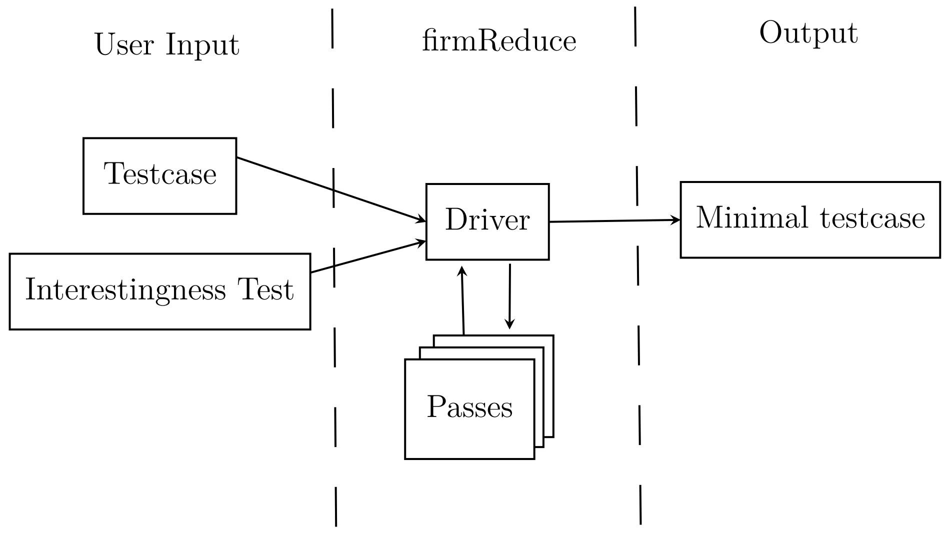


Figure 4.1: Structure of FirmReduce
The user inputs the testcase and Interestingness Test. The driver then loops through the passes that produce new variants, while making sure the new variants fulfill the Interestingness Test. The final output is a minimal testcase.

4.1 Driver

The driver manages the interaction of the passes and the interestingness test. Within its responsibilities lies the decision of which pass to apply to which part of the program and when to test the current variant for reproducibility. The passes are plugged into the program dynamically and are reloaded at every program start to allow for easy addition / removal of passes. A path to the reproducer script needs to be passed to the driver when starting the reduction.

4.1.1 Optimization strategy

FirmReduce uses a greedy optimization strategy meaning the variant given to the passes to reduce is always the best one we have found so far. A variant is considered 'better' than another variant, if it either

- has fewer nodes,
- has fewer graphs,

- has reduced control flow complexity, measured by the number of nodes in the program that influence the control flow,
- has less data dependencies, measured by the number of nodes in the program, performing memory dependent operations,
- or has less types.

Choosing these criteria may lead to one criterion favouring variant A, whereas another criterion favours variant B. If from the criteria it is not clear, which variant is the better one, we choose the newer variant, as we assume that the passes used in the reduction are implemented to transform the input variants in a useful way. Additionally, we allow for these metrics to become temporarily worse, as many compiler optimizations require node insertions to be able to simplify the program.

Because the FIRM intermediate representation is a low-level representation of a program compared to C source code, the correlation between the number of nodes and the complexity of the program is quite high compared to the metric used in *C-Reduce*, which is the number of lines in the C source code.

4.1.2 Reduction Loops

FIRMReduce uses two separate reduction loops, that work with two different granualities on the input file. The first loop operates on graph-level, meaning it advises the passes to apply as many transformations to a graph as possible. The advantage of being this aggressive with the reduction is a significant reduction in execution time. This is due to less system overhead, since every pass invocation requires creating a new process, as well as less overhead in FIRMReduce itself. If after a graph-level application of a pass the interestingness test fails, the reduction approach may have been too aggressive. The graph-level changes are discarded. In a second reduction loop, all graph-pass combinations that yielded uninteresting outputs during the first reduction loop are executed again. This time the pass is restricted to do only a single transformation and return the result. The order in which passes are applied is important. Applying pass A to a graph can lead to pass B being able to make more transformations than before the application of pass A. An example of this is shown in figure 4.3. The order which is best depends on the characteristics of the test-case. Therefore it cannot be determined in advance. Instead, FIRMReduce uses a randomized order in which it applies the passes to the test-case. In most cases, passes will be executed more than once, as the reductions that were applied to the variant in between these two executions, allow the pass to find more places in the program, where it can apply its own transformations. To exploit synergy effects between the passes as much as possible, all passes have to applied at least once, before one pass will be applied a second time. The randomization in FirmReduce is therefore based on iterating over a permutation of the list of loaded passes, before creating a new permutation and repeating the process.

The reduction is complete and FirmReduce exits, if a fixpoint in the reduction is found. A fixpoint is a variant that cannot be further reduced by any of the passes in FirmReduce . Here, we use 1: we stop the reduction if every variant smaller than the current one is either not a valid Firm graph (thus, it is not a testcase, since we defined a testcase to be a valid input) or it is not interesting w.r.t. the predicate used for reduction.

Listing 4.1: Algorithm used by the driver to reduce a testcase

```

while !fixpoint {
    passes ← random permutation of passes
    for pass in passes {
        while testcase contains reduction opportunities {
            irg ← random irg
            new_variant = apply_pass_aggressively(pass, irg,
                current_variant)

            if new_variant is reproducer {
                current_variant ← new_variant
            } else {
                mark((pass, irg))
                discard(new_variant)
            }
        }
    }
}

for (pass, irg) in passes × irgs {
    if marked {
        new_variant = apply_pass_conservatively(pass, irg,
            current_variant)

        if new_variant is reproducer {
            current_variant ← new_variant
        } else {
            mark((pass, irg))
            discard(new_variant)
        }
    }
}

```

4.2 Passes

The responsibility of the passes is the reduction of the input program. Each pass performs one specific transformation to the program. The modular architecture of FirmReduce means, passes can be added and removed easily and can be implemented in different programming languages. The driver communicates with each pass over the command line: It decides for the pass, which input file it should transform, which graph of that input and how aggressive the reduction should be. Internally, a pass does not need to adhere to a certain structure. Each pass needs to make sure it produces a valid program, i.e. a program that the compiler should be able to handle without faults.

4.2.1 Destructive Passes

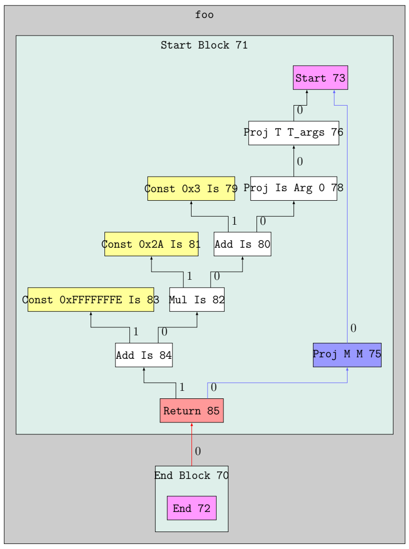
Destructive passes do not uphold semantic equivalence between the input and the output program. This yields more possibilities for transformations. The destructive passes and their transformations that are used in FirmReduce are:

Garbage Collection of IRGs The pass tries to remove a graph from the program. This transformation only succeeds, if no other graph has a dependency to the deleted one and we are not trying to remove the main function. This pass works best, if, by removing function calls from the program, we have already removed dependencies between graphs.

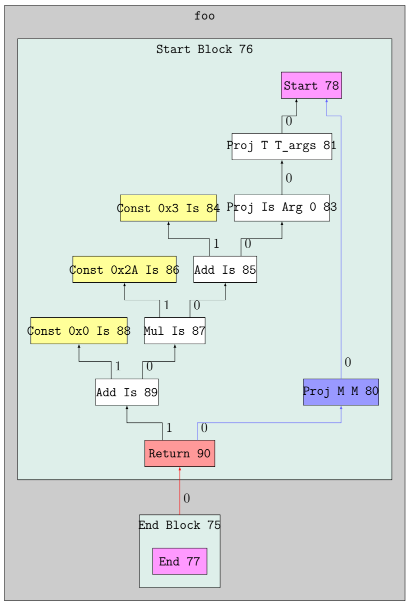
Removal of `Alloc nodes` `Alloc nodes` that only have a data dependency in-edge and no control flow in-edge are removed. The data dependency is re-routed to the previous memory state. `Alloc nodes` without successors except the output memory state, usually don't appear in a program naturally, but after the application of other passes, that eliminate the need to save a variable, because it is unused in the rest of the program.

Removal of `Store nodes` All `Store nodes` in the graph are removed. All memory successors are re-routed to the `Store node`'s memory predecessors. This not only voids the single store operation, but also all calculations done to the value that was to be saved at this point in the program.

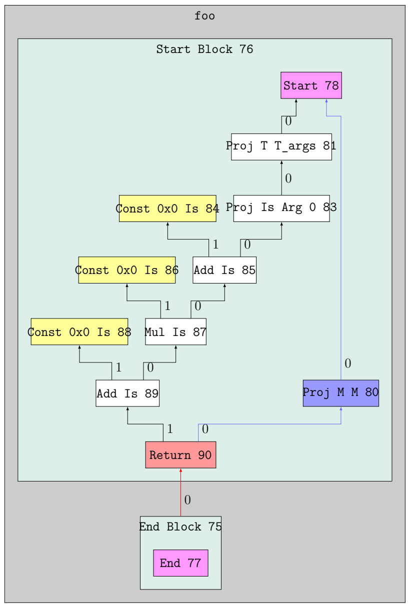
Removal of Void Calls The pass removes all calls to void functions and non-void functions, the return values of which are not used. Function calls that do have a return value that is used in the initial program cannot be removed by this pass. However testing has shown, that the data and memory dependencies of a function's return value are reliably removed by the other passes. The return



(a) Original graph The arithmetic operations all have a non-zero constant operand.



(b) Conservative Reduction The transformation was applied to only one of the constant values. The last addition becomes superfluous and will be removed, the other two remain unchanged.



(c) Aggressive Reduction The transformation was applied to all `Const` nodes. The result of the function will always be 0, hence all operations done in the function will be removed.

Figure 4.2: Aggressive vs Conservative Reduction The pass used in this example replaces constants with the value 0.

value becomes an unused variable, hence this pass is now able to remove the function call completely. In all (so far...) testcases we reduced, function calls were removed, except the ones that are important to a testcases reproducibility properties.

Replacement of Arithmetic nodes The pass removes nodes that perform arithmetic operations, including `Add`, `Sub`, `Mult`, `Multh`, `Div`, `Mod`, `Shl` and `Shr` nodes. The operation is replaced with a `Const node`, having the same mode as the operation's result and containing the value 0.

Replacement of Cond Selectors In libFIRM every `Cond node` has a Selector, that selects one of the outgoing control-flow edges as the next instruction, based on the evaluation of the condition given to the node. The transformation done in this pass replaces this condition randomly by true or false. Making the result of the condition a constant, means always the same branch is chosen for execution. The other one is dead and can be garbage collected.

Replacement of `Load nodes` `Load nodes`, that load values stored on the heap or stack, are replaced by `Const nodes` that hold values of the same mode and the value 0. If all instances, where a value in memory is loaded, are removed from the program, we assume the storing of the value itself to become obsolete. The corresponding `Store node` will be removed by subsequent passes.

Replacement of Mux Selectors Similar to `Cond nodes`, `Mux nodes` have a selector that chooses one of two operands depending on the evaluation of a boolean value. This boolean value is replaced by a randomly chosen constant. The not-chosen operand and all its control-flow dependencies become unreachable in the program and will be removed by other passes in the course of the reduction.

Replacement of `Proj nodes` `Proj nodes` are used to project single values from a tuple. Operations that return tuple values include function calls, load and store operations and reading function arguments. If the value projected is of a primitive type, we substitute this `Proj node` for a `Const node`, containing a constant value of the same type. Doing this also removes the data dependency to the `Proj node`'s predecessor.

Replacement of Return Values The return value of a function is replaced by a constant value. A function that always returns the same constant value will be inlined by the subsequent optimizations, therefore removing all data dependencies to the function. This makes the removal of the corresponding graph possible.

Replacement of Switch Selectors The selector value, that indicates which control-flow branch will be executed, is replaced by a constant integer value. This leads to the same effects described in the passes section 4.2.1 and section 4.2.1.

Substitution of Constant Values The value of a variable represented by a `Const node` is set to the value 0.

The nature of the transformations used in FIRMReduce 's destructive passes allows them to be used either on the whole program (i.e. each node that the transformations may be applied to), all nodes in a single graph or even only a single node. For each possible reduction granularity, there is a trade-off between execution time and reduction success: Being more aggressive in a single reduction iteration means that the interesingness and other validity tests on the variant have to be executed less often, therefore reducing overall execution time. However, the more a variant is changed before it is tested for reproducibility, the higher is the probability that this test fails and the variant has to be discarded. Reducing the whole program at once has proven to be impractical as the interestingness test would fail too often. The most aggressive reduction FIRMReduce uses is a graph-level reduction. If this granularity proves to be to low, the driver remembers the tuple of pass and graph that failed and in a later reduction cycle will retry the reduction with a finer granularity (node-level).

4.2.2 Non-destructive Passes

Non-destructive passes do not change the semantics of the input file. This is needed when we reduce a testcase, the interestingness of which can only be determined by evaluating the output of an execution of the compiled testcase. For the transformations, FIRMReduce uses compiler optimizations that are implemented in libFIRM . This raises the question if the reduction may be compromised if there are bugs in the optimizations, especially since the input for the passes are programs that are supposed to trigger these bugs. This is not the case, as changes in the semantic validity of the program is either not important to its quality as a bug reproducer or are discovered by the interestingness test and immediately discarded. In cases where a pass fails to produce a syntactically valid program from a syntactically valid input, we may have found a variant that invokes a bug in the pass. This is especially interesting if the pass solely consists of an optimization used in libFIRM . The transformations of the pass are discarded, however the input variant is saved as a potentially interesting, bug-triggering testcase to examine at a later time.

4.2.3 Pass Synergies

The transformation used by the passes, especially the ones described in section 4.2.1 are very simple. Despite this, they manage to reduce the input program significantly, because we can exploit synergy effects between the passes. Applying each pass once may not result in a minimal program, but the application of passes can produce new opportunities for other passes to further transform the program. An example for synergy effects occurring between the replacement of `Proj nodes` and the removal of `Call nodes` can be seen in figure 4.3.

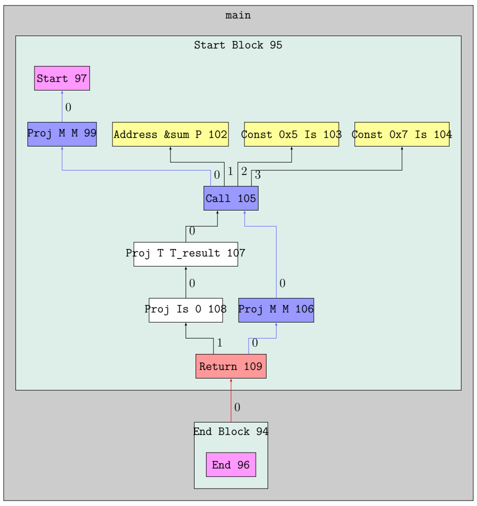
Other synergy effects taking place are:

Passes producing unreachable code Some passes, such as the replacement of `Cond`, `Mux` or `Switch` selectors, produce unreachable code, i.e. code that will never be executed. The compiler optimizations will remove the dead branch from the program graph.

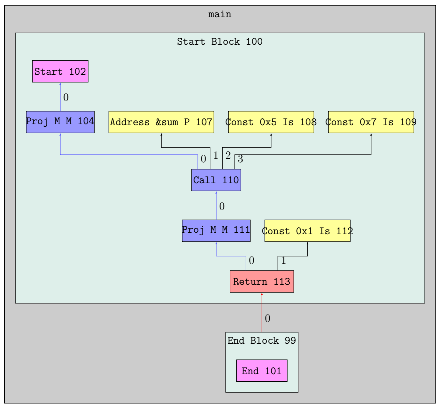
Passes producing dead code Because the passes remove calculations, function calls or store operations, the variables used in these become unused. The data dependencies to the part where these variables are calculated no longer exist, therefore these calculations can also be removed from the program, as they are unnecessary.

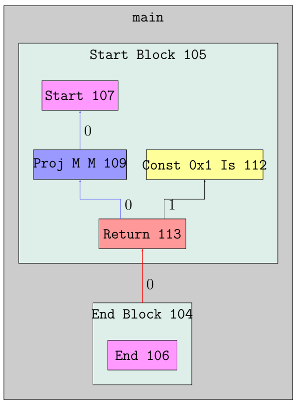
Passes replacing variables by constants The value of variables is usually unknown until run-time. FIRMReduce replaces these variables with constant values, therefore making especially local optimizations more effective.

The occurrence of synergy effects between specific passes was analyzed in the course of the evaluations done in this thesis. The results are illustrated in section 5.2.2 and especially in table 1, where we examined which passes where most beneficial to call after the application of a specific pass.



(a) Original graph The `Call node` cannot be removed, because the Call result is needed for later operations.





(b) The `Proj node` was replaced by a `Const node`. This also removed the data dependency to the function call. Now the pass are able to the function call

(c) The resulting graph without the function call.

Figure 4.3: Synergy Effect between the `Call node` removal and `Proj node` replacement The pass used in this example replaces constants with the value 0.

4.3 Interestingness Test

The criterion that marks a testcase as a bug reproducer is dependent on the bug itself, therefore the interestingness test cannot be hard-coded into the program. It is separated into a user-supplied script that is invoked by the driver at the appropriate times. The user can specify a set of command line arguments that will be passed to the reproducer script by the driver to increase reusability of the script.

5 Results and Evaluation

In this chapter we will evaluate our implementation of FIRMReduce . We will compare its output to the *minimal interesting testcase* as well as C-Reduce's output for the same testcase. Furthermore we examine the impact of randomization during the reduction process.[1]

5.1 Input Data

The testcases used for this evaluation are produced by FIRMSmith [2] and *CSmith* [10] (version 2.3.0). FIRMSmith generates programs directly in the FIRM intermediate representation, while *CSmith* delivers C programs that need to be translated into FIRM representation before FIRMReduce can use it. The translation was done using *cparser* (version 1.22.1). FIRMSmith and *CSmith* were both developed to find programs where the tested compiler shows incorrect behaviour. For the purpose of this evaluation, we modified the fuzzers in order to also produce programs where libFIRM behaves correctly. This produces testcases that compile successfully.

For the interestingness tests we used two different kinds of predicates: First, we used the compilability of the input testcase: A testcase is interesting iff it compiles successfully. Using a predicate that looks for a certain pattern in the testcase might inhibit a specific pass from exploiting its full reduction potential and therefore distort the analysis to the disadvantage of said pass. Choosing this method of evaluation also allowed for a bigger sample size, as bug-triggering testcases are significantly harder to find.

Secondly, we used testcases that trigger existing bugs in the compiler. These were harder to compute for the fuzzers and therefore have a smaller sample size for each predicate used. However, these cases give a clearer indication of the potential reduction that can be achieved for a predicate that likely depends on a more complex structure in the input testcase than the ones that were tested in the first step.

[1] All testcases and scripts used in this chapter can be found in: `https://github.com/tnstrssnr/firmreduce`

5.2 Results

5.2.1 Reduction

For this part of the analysis, we used the compilability of the program as the reduction predicate. The interestingness test used is shown in figure 5.1.

```
#!/usr/bin/env bash
! cparser $@ >/dev/null 2>&1
```

Figure 5.1: Interestingness Test that checks if the input is compilable

For this predicate, the minimal interesting testcase, which is the smallest possible compilable C program is shown in figure 5.2. It consists of seven nodes. Additionally, the figure shows the smallest FIRM graph any function in a C program can have.

```
int main() {
    return 0;
}
```

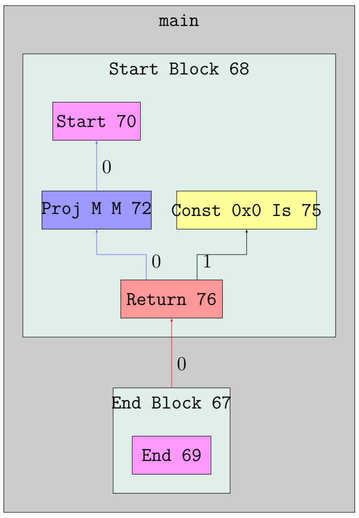


Figure 5.2: Smallest compilable C program as source code and in FIRM intermediate representation

To test the reduction as a whole, we used 107 different input testcases[2] and ran FIRMReduce once for each. The average size of the input testcases was just over 8000 nodes, with the values ranging between 912 and 18585 nodes per testcase. The size of most input testcases either range from ~900 to ~1500 or between ~12.000 and ~15.000.

In 94.4% FIRMReduce was able to produce the minimal interesting testcase. For the remaining testcases, the result either consisted of 14 or 17 nodes. The output

[2]see directory `{firmReduce_home}/examples/Working`

testcases containing 14 nodes consist of two instances of the graph shown in figure 5.2. As there are no dependencies between the two graphs, FirmReduce should have been able to remove one of them. One output testcase contained 17 nodes, with the graph of the main function being a minimal 7-node graph and the graph of the second function containing a recursive call to itself. The function call is void, so ideally it should have been removed by FirmReduce . For all testcases where the maximum reduction was not achieved on the first run of FirmReduce , a second run was performed on the output of the first reduction. This time all testcases were reduced to the minimal interesting testcase. Running FirmReduce a second time on the original input with a different seed also yields the minimal interesting testcase.

FirmReduce failed to fully reduce these testcases on the first try, because the program prematurely assumed it has found a fixpoint. This can happen due to the randomized nature of the reduction process, as FirmReduce does not guarantee to apply a pass to every graph in the testcase.

We also conducted a second run of FirmReduce , where the order of passes was completely randomized, instead of using a permutation of passes, where immediate repetitions of passes are not possible. This reduced the share of testcases reduced to 7 nodes to 84.4%. Guaranteeing, instead of expecting, that each pass will be run periodically increases the quality of the reduction due to an improved pass success rate, caused by stronger synergy effects, and a more reliable fixpoint identification.

How the program size develops during the reduction process can be seen in figure 1. The diagram shows the number of nodes over the number of passes applied for a selection of the testcases. To keep the diagram easy to read we did nott include all testcases, a second graph containing all data can be seen in appendix 1. All reductions seen in the graph show the same pattern: The number of nodes stays stationary for a number of passes, until it drops by a significant amount with only one pass. Mostly, these drops are the result of a destructive pass immediately following a number of non-destructive passes. A more detailed of pass effectiveness under different circumstances is given in section 5.2.2.

Also apparent in the diagram is the strong variance in the number of passes needed during a reduction to reach the fixpoint. This variance can also be seen in the execution time FirmReduce needs. Not only is the variance great between testcases of different sizes, but also between testcases of similar sizes. This suggests that the order of passes in the reduction has a great impact on the reduction result. Reducing the same testcase with different seeds confirms this assumption, as the ratio of successful/total number of passes ranges between 14.6% and 3.2%.

table 5.4 summarizes the results of the analysis.

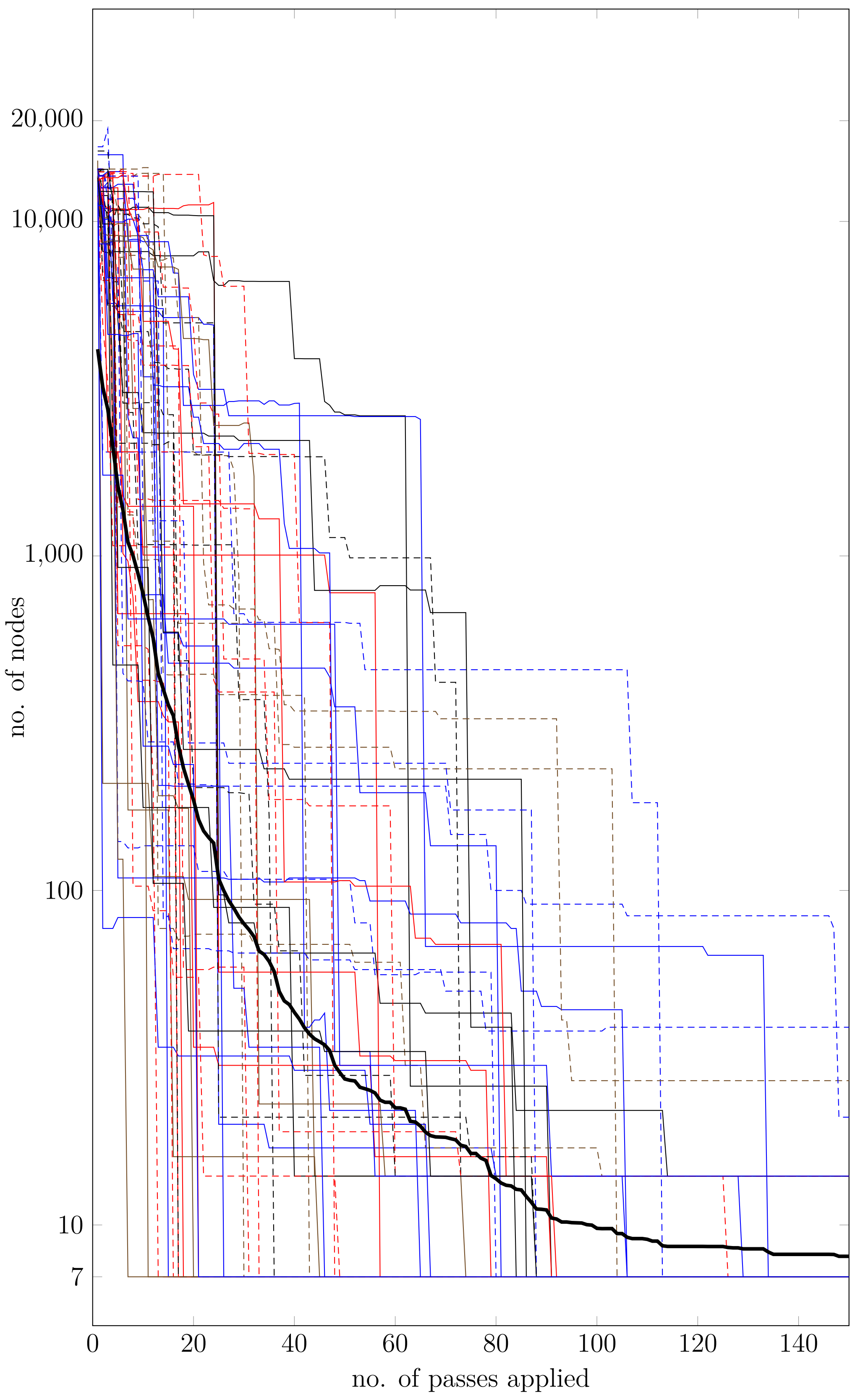


Figure 5.3: Size of graphs over no. of passes applied Each line shows the total number of nodes after x passes have been applied for a different input file. The thick line shows the geometric mean of the data plots.

Category	Max.	Min.	Median	Standard Deviation
Size of input	18585	912	11030	6358
Size of output	17	7	7	1.75
output size / input size	0.111 %	2.2 %	0.043%	0.46 %
no. of passes applied	279	25	125	42.7
no. of passes applied successfully	18	3	8	4
applied / success	32.1 %	1 %	6.4 %	4.7%
execution time (hh:mm:ss)	02:25:58	00:00:11	00:08:52	00:21:14
% of testcases, where max. reduction was achieved	94.4%			

Table 5.1: Results of reducing 107 different testcases

5.2.2 Pass Efficiency

This section details the efficiency of the single passes in the reduction process. For this purpose, we analyzed each pass individually and during the reduction process. For this section we will call a pass *successful*, if it managed to remove a part of the testcase or simplified the testcase in a way that is better according to the metric defined in section 4.1.1.

First we applied each pass to the same testcases used in section 5.2.1 separately in order to see, which passes achieve a reduction and which don't. The result is shown in table 5.2. Most passes are never successful, giving the impression that they are superfluous and don't need to be included at all. A smaller subset of passes is either successful for some testcases, but not all. Contrasting this, table 5.3 shows the pass evaluation during a complete run of FIRMReduce . After running the testcases, the results show, that almost every pass had successful invocations. This difference in pass effectiveness can be attributed to the synergy effects occurring between the passes during the reduction process. Overall, the sucess rate of the passes is acceptable, but with an average of 10.01%, there is still room for improvement. Unsuccessful pass invocations happen predominantly towards the end of the reduction. The smaller the testcase has already become, the smaller is the subset of passes that still have the ability to reduce the testcase further. The probability of randomly choosing one of these passes declines over time. This can also be seen in figure 1, where the slope of the curve initially is quite high and flattens towards the end of the reduction. Apart from choosing the wrong passes, we also need a series of unsuccessful passes to identify a fixpoint. On average, 70 passes are applied without any progress, until FIRMReduce recognizes a fixpoint.

The second and third column in table 5.3 shows the average size delta for each pass, i.e. the percentage by which each pass could reduce the testcase compared to the

Successful every time	**Never Successful**	**Sometimes Successful**
remove void calls	replace loads	opt blocks (38.74%)
replace cond selectors	remove allocs	normalize n returns (3.61%)
remove stores	replace mux selectors	opt jumpthreading (1.81%)
replace switch selectors	normalize one return	opt tail rec (80.19%)
replace return values	remove critical cf edges	optimize reassociation (31.54%)
combo	remove unreachable	opt if conv (36.04%)
conv opt	opt funccalls	opt ldst (72.08%)
optimize graph df	opt frame	replace proj (72.08%)
optimize cf	gc irgs	
simplify consts	gc entities	
replace arithmetic	do loop inversion	
	remove bads	
	opt bool	
	construct confirms	
	scalar replacement opt	
	place code	
	optimize load store	
	opt parallelize mem	
	gvn pre	
	occult consts	
	dead code elimination	

Table 5.2: Evaluation of Pass Efficiency for passes that were applied to different testcases by itself. The passes can be put in the three categories seen above, with unsuccessful passes being the majority.

state before pass invocation and the initial testcase size respectively. The data shows the destructive passes being far more effective than the non-destructive passes. Some non-destructive passes even on average increase the size of the testcase, however this will not greatly affect our result, since these increases are small. More likely, these create opportunities for other passes to reduce the testcase further. Additionally, since changing parts of the program without changing its interestingness further isolates the failure-inducing elements, even a slightly bigger graph may help in identifying the underlying issue if we compare different variants throughout the reduction process.

To be able to increase the success rate of FirmReduce 's passes, we analyzed the sequences of successful passes that occurred in the testruns and scanned them for recurring patterns. For each pass, the list of passes that were successful immediately afterwards is no longer than 22 (out of 47) passes, showing a clear inclination towards certain passes being better choices than others. The detailed results of this analysis can be seen in table 1. This knowledge can be used to define a heuristic for choosing

Pass	**% successful**	**Δ size (%)**	**Δ size compared to initial size**
Destructive Passes			
replace arithmetic	14.02%	29.74%	25.00%
replace loads	0%	26.74%	-
remove allocs	28.68%	27.56%	2.92%
simplify consts	27.93%	15.41%	9.56%
replace mux selectors	0%	-	-
replace proj	24.63%	33.62%	16.64%
remove void calls	12.84%	7.16%	1.70%
replace cond selectors	20.08%	42.90%	26.95%
remove stores	26.52%	37.38%	13.42%
replace switch selectors	0%	-	-
replace return values	12.88%	23.77%	18.36%
gc irgs primitive	3.38%	67.6%	8.06%
Non-Destructive Passes			
combo	6.82%	24.26%	21.59%
normalize one return	7.96%	-1.27%	-1.11%
opt blocks	9.1%	-0.69%	-0.87%
remove critical cf edges	6.8%	-2.23%	-1.86%
remove tuples	0%	-	-
remove unreachable	0.0%	-	0.13%
opt funccalls	1.51%	2.50%	0.27%
normalize n returns	12.46%	0.92%	0.46%
opt frame	0.0%	-	-
gc irgs	29.55%	86.19%	28.96%
do loop inversion	0%	-	-
conv opt	8.34%	9.95%	8.45%
remove bads	2.27%	1.08%	0.40%
opt bool	0.0%	-	-
construct confirms	11.52%	1.59%	0.36%
scalar replacement opt	1.14%	0.05%	0.04%
opt jumpthreading	4.53%	0.21%	0.72%
optimize graph df	11.75%	35.12%	30.83%
place code	9.44%	-2.73%	-1.61%
opt tail rec	11.37%	3.09%	1.49%
optimize load store	25.76%	12.74%	1.99%
optimize reassociation	7.2%	-0.07%	-0.05%
opt if conv	9.47%	2.08%	2.09%
opt parallelize mem	17.36%	–0.86%	-0.23%
gvn pre	8.72%	-1.11%	-0.97%
optimize cf	14.34%	3.31%	1.66%
occult consts	2.32%	1.08%	1.00%
opt ldst	0%	-	-
dead code elimination	0%	-	-
gc entitites	8.34%	24.73%	0.00%
Total	10.01 %		

Table 5.3: Evaluation of Pass Efficiency during the reduction process. The table shows the % of total invocations that each pass has been successful as well as the difference in size each pass achieved compared to the programs size before pass invocation, as well as initial program size.

passes in the reduction process, where each pass is given a weight dependent on the previously chosen passes and passes with higher weights are more likely to be chosen.

5.2.3 libFirm Bugs

To test how FirmReduce deals with real use cases, we identified several bugs in libFirm , constructed testcases for them using FirmSmith and variants produced by FirmReduce and examined the reduction results according to the same aspects used in the previous chapters.

The testcases we used showed the following faulty behaviours:

Assertion failure on call to function 'gen_Proj_Load' Compiling a testcase from `{firmReduce_home}/examples/pn_load` using *cparser* and the options `'-fbool -O0'` results in an assertion failure with the following output to stderr:

```
cparser: ir/be/amd64/amd64_transform.c:2785: gen_Proj_Load:
    Assertion ‘pn == pn_Load_M’ failed.
Aborted (core dumped)
```

Assertion failure in Load Store Optimization When applying the Firm optimization `opt_ldst` to a testcase found in `{firmReduce_home}/examples/ldst`, libFirm aborts with the following error message:

```
pass_libfirm_opt_ldst: ./ir/ir/irnode_t.h:650: set_Block_phis_:
    Assertion ‘ir_resources_reserved(get_irn_irg(block)) &
    IR_RESOURCE_PHI_LIST’ failed.
Aborted (core dumped)
```

Assertion failure in IRG garbage collection When calling the garbage collection function for graphs `gc_irgs`, libFirm aborts with the following error message:

```
pass_gc_irgs: ir/ana/cgana.c:52: cg_get_call_n_callees: Assertion
    ‘is_Call(node) && node->attr.call.callee_arr’ failed.
Aborted (core dumped)
```

Testcases for this bug can be found in `{firmReduce_home}/examples/gc`

Assertion failure on edge exchange On re-routing edges from one node to another, using the `exchange` function, libFirm aborts with the following error message:

```
pass_replace_loads: ir/ir/irgmod.c:44: exchange: Assertion ‘irg ==
    get_irn_irg(nw)’ failed.
Aborted (core dumped)
```

Testcases for this bug can be found in `{firmReduce_home}/examples/exchange`

Segmentation fault during Compilation When compiling a testcase from `{firmReduce_home}/examples/exit_code_11` using *cparser*, the compiler crashes due to a Segmentation Fault.

Pass success rates, execution time and pass applications needed for the reduction on average do not differ from the results obtained in section 5.2.1, where we reduced testcases without a specific predicate other than the validity of the testcase. Therefore we will concentrate on the output testcases for each bug.

For the bug in the Load-Store-Optimization, the testcases were mostly reduced to a minimal 7-node graph. A small minority consisted of 14-17 nodes, however this is due to a premature termination of the reduction, as is explained in section section 5.2.1. The graphs being reduced to the smallest outcome possible, suggests that the cause for the compiler failure does not lie in the structure of the program itself. Other programs, with the same program graph do not invoke the error. The difference between these testcases may lie in the types that are included in the testcase. If we compare the minimized testcases to a program with the same graph, that didn't go through a reduction process, we see big difference in the number of types included. The reduced testcases still include old, unused types, since no pass or compiler optimization removes them. FirmReduce is only designed to isolate structures in the program graph and not the type system, which means that a further analysis of the bug will have to be done by hand.

The bug in the exchange function, a function that re-routes edges from one node of the graph to another, was discovered while using the function in one of the passes in FirmReduce . The reduction of the affected testcases results in graphs, that contain one memory operation plus the necessary successors and predecessors of this operation, leaving a single opportunity for the pass to apply the exchange function.

A similar result was obtained from the reduction of testcases for the garbage collection bug. Every part of the program could be removed, except for the parts needed for the pass to be applied. Interestingly, all reduced testcases included a function call to a function that was previously removed from the program. This suggests that the error is not in the subsequent call to the garbage collection function, but happened beforehand, when a function, contained in the call graph of the main function, was removed from the program.

The testcases for the assertion failure during the `Proj node` generation all had similar sizes coming out of the reduction. Furthermore they all had a similar structure, consisting of only one main graph, that contained exactly two `Cond nodes`, many load and store operations, but not many other statements. The similarities between the reduced testcases let us conclude that theresults are close to the minimal interesting

Category	**Max.**	**Min.**	**Median**	**standard deviation**
Assertion failure on call to function 'gen_Proj_Load'				
Size of input	13469	12380	12380	266.3
Size of output	222	145	146	22.1
Assertion failure in Load Store Optimization				
Size of input	14819	16	3624	5322.8
Size of output	16	7	7	3.6
Assertion failure in IRG garbage collection				
Size of input	15827	94	5097	5751.6
Size of output	8122	24	24	1433.6
Assertion failure on edge exchange				
Size of input	14819	17	947	4218.1
Size of output	34	17	22.5	5.5
Segmentation fault during Compilation				
Size of input	14287	9226	12967	1896.5
Size of output	5582	108	1781.5	2323.2

Table 5.4: Results of reducing testcases for the different bugs in libFirm

testcase for this bug.

The results of the segmentation fault testcases contained just over 100 nodes on average. Running FIRMReduce multiple times on these testcases didn't improve the outcome. The resulting graphs contain many structures that FIRMReduce should be able to reduce, if it won't cause the interestingness test to fail. Since it is yet unknown what the cause for the segmentation fault is, we cannot assess how close the reduction results are to the minimal interesting variant of each testcase. Comparing the testcases amongst each other does not deliver much insight, since we only have two different testcases for this bug. The other two testcases we found for the segmentation fault bug, as well as two of the testcases belonging to the failed garbage collection, contained over 5000 nodes after the reduction and were clear outliers compared to the other testcases we reduced. All of them were composed of a small main function and a large second function, that would fail the interestingness test if a pass was applied.

5.2.4 Comparison to C-Reduce

C-Reduce uses C source code as its testcases and the number of lines in the source code as the reduction metric. To be able to compare the results of *C-Reduce* and FIRMReduce , we translated the original C file to FIRM intermediate representation using *cparser*, used this as input for FIRMReduce . Then we also translated *C-Reduce*'s output to FIRM representation and compared the graph to FIRMReduce 's output. To be able to compare the two reducers, we need to use *cparser* twice, which might distort the results. The same applies to the evaluation of variants that *C-Reduce* and FIRMReduce do. They use different metrices that don't always rate the same variant as the smaller one.

Because we needed testcases in C source code, using FIRMsmith is not an option. *CSmith* was not able to produce testcases with compiler errors. Therefore we decided to manipulate libFIRM in different ways, that would cause the compiler to crash, as long as either a multiplication (explicit or implicit in for example a pointer dereference) or a comparison is part of the testcase. A comparison between the output sizes of the two reducers is presented in table 5.5. Then we used small C programs from the libFIRM test suite[3]

Regarding execution time, *C-Reduce* is clearly the faster reducer. The causes for FIRMReduce 's lacking efficiency were already analyzed earlier in this chapter. Ways to improve FIRMReduce 's efficiency will be outlined in chapter section 5.3.

Testcase No.	initial size	C-Reduce Result	FirmReduce Result
1	159	16	16
2	287	22	9
3	184	16	30
4	372	22	138
5	401	17	25

Table 5.5: Comparison between *C-Reduce and* Firm*Reduce* The size of the testcases is measured in # of nodes in their respective FIRM graph.

The output graphs of both reducers for most testcases are a similar size. However, we also found outliers, where FIRMReduce 's output was significantly bigger than that of *C-Reduce*. One reason for the large graphs is that they contain program structures that FIRMReduce 's passes can't handle. Adding more passes to FIRMReduce to deal with those will likely improve the result. Another reason is that many passes, including all compiler optimizations, affect large parts of the graph. This often causes the interestingness to fail. *C-Reduce*'s passes on the other hand often only affect a small part of the program, whereby the interestingness test will fail less often.

[3]`https://github.com/libfirm/firm-testsuite/blob/master/opt/HeapSort.c`, `https://github.com/libfirm/firm-testsuite/blob/master/opt/MergeSort.c`, `https://github.com/libfirm/firm-testsuite/blob/master/opt/Queens.c`, `https://github.com/libfirm/firm-testsuite/blob/master/opt/Hanoi.c`, `https://github.com/libfirm/firm-testsuite/blob/master/opt/QuickSort.c`

5.3 Future Work

Based on the results outlined in this chapter so far, the following improvements and extensions can be added to FirmReduce to improve the results:

Increasing Pass-Scope

So far FirmReduce uses 47 different passes for the reduction. Especially the destructive passes that remove a certain type of node are responsible for pushing the reduction forward. Here we only have 12 passes that each target a specific type of node. As the results have shown, this is enough to reach a satisfying result for most input cases. Widening the scope of the passes, by adding more destructive passes for so far unconsidered nodes, especially `Member nodes` and `Jmp nodes`, will certainly increase the reduction capabilities of FirmReduce .

Heuristically choose the next pass

Instead of choosing the next pass randomly over a uniform distribution, using a heuristic may increase the success rate of passes over the course of a reduction. Based on the sequences of successful passes seen during the testing phase, for each pass we can define a probability measure that is used to pick the next pass. A higher weight will be given to passes that proved to be more successful in the past, meaning they are more likely to be chosen.

Parallelization

Depending on the size of the initial input and the order of the passes, FirmReduce took between 1 and 60 minutes to complete a reduction. The execution time could be improved by exploiting the Parallelization potential that the program offers. One possibility of parallelizing the reduction was suggested by the developers of C-Reduce in [11]. For FirmReduce 's purposes it would be useful to outsource the pass application to multiple child processes until one of them has found a new smaller variant. All other child processes are interrupted and restarted with the new smaller variant as their input.

Reducing Failure-Inducing Differences

Apart from reducing a single testcase that contains the for the failure critical section, it may be useful for debugging to have two similar testcases, one that fails and another that does not fail. Using the same reduction on both of them while ensuring that still only one of them fulfills the reduction predicate results in two smaller testcases whose differences highlight the failure-inducing parts of the program. Such two testcases could be obtained during a normal reduction, if a pass application causes the interestingness test to fail.

6 Conclusion

The aim of this thesis was to provide an implementation for an automatic testcase reducer that operates on programs in the Firm intermediate representation and subsequently evaluate its performance, using testcases that trigger bugs in the libFirm compiler backend.

After evaluating the data we collected about FirmReduce , the following conclusions can be made: FirmReduce has a respectable ability to reduce a testcase, having reduced the majority of tested inputs to or close to its minimum. Improvements can be made to FirmReduce 's efficiency however: The execution time and pass success rate are highly dependent on the order of passes during the reduction. Introducing a heuristic for achieving a favourable order of execution will most likely improve the program's performance.

In the future, we would like to add more transformations to FirmReduce , to increase the amount of language constructs that FirmReduce can utilize in its reduction process. Furthermore, improving FirmReduce 's execution times through an improved pass heuristic and parallelization will increase its usability.

As a final conclusion we can say that FirmReduce , while still offering room for improvement, provides a helpful tool to use in the debugging of libFirm , as it is able to create small and easy to understand testcases for a given compiler error.

Bibliography

[1] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-Case Reduction for C Compiler Bugs,” p. 11, 2012.

[2] J. Wagner, “Firmsmith test generation for compiler optimizations,” p. 44.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[4] R. Wilhelm and D. Maurer, *Übersetzerbau: Theorie, Konstruktion, Generierung ; mit 70 Tabellen.* Springer-Lehrbuch, Berlin: Springer, 2., überarb. und erw. aufl ed., 1997. OCLC: 75899693.

[5] M. B. S. Buchwald and A. Zwinkau, “FIRM—A Graph-Based Intermediate Representation,” p. 8, 2011.

[6] S. Buchwald, D. Lohner, and S. Ullrich, “Verified construction of static single assignment form,” pp. 67–76, ACM Press, 2016.

[7] “Firm - Optimization and Machine Code Generation.” `https://pp.ipd.kit.edu/firm/`. Accessed: 2018-06-28.

[8] “The Gecko BugAThon.” `https://www-archive.mozilla.org/newlayout/bugathon.html`. Accessed: 2018-07-30.

[9] A. Zeller, “Simplifying and Isolating Failure-Inducing Input,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 28, no. 2, p. 17, 2002.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *SIGPLAN Not.*, vol. 46, pp. 283–294, June 2011.

[11] J. Regehr, “Parallelizing Delta Debugging – Embedded in Academia,” Nov. 2012.

Erklärung

Hiermit erkläre ich, Tina Maria Strößner, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Appendices

1 Size over Time - Full data

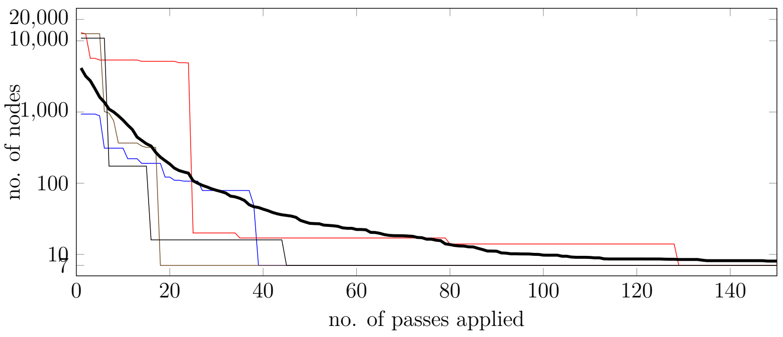


Figure 1: Size of graphs over no. of passes applied Each line shows the total number of nodes after x passes have been applied for a different input file. The thick line shows the geometric mean of the data plots.

2 Analysis of successful sequences

combo	**occult consts**	**optimize reassociation**	**remove critical cf edges**
replace arithmetic (4)	simplify consts (3)	conv opt (3)	opt blocks (3)
gvn pre	**optimize graph df**	**construct confirms**	**conv opt**
opt blocks (3) replace return values (3)	replace return values (4) optimize load store (4)	gc irgs (3) normalize n returns (4) combo (3)	replace cond selectors (4) opt tail rec (3) opt if conv (3)
place code	**opt blocks**	**opt tail rec**	**optimize cf**
opt if conv (3) remove stores (3) replace return values (3)	opt if conv (3) opt tail rec (3) optimize load store (3) remove void calls (3)	opt if conv (3) simplify consts (4) replace proj primitive (3) replace arithmetic (3)	replace proj primitive (4) opt jumpthreading (4) optimize load store (4) gvn pre (4) place code (4)
opt parallelize mem	**remove void calls**	**replace proj primitive**	**gc irgs primitive**
gvn pre (3) gc irgs primitive (4) simplify consts (3) optimize load store (3) opt blocks (3)	optimize load store (5) simplify consts (7) replace proj primitive (3) gc irgs (3) gc irgs primitive (3)	remove stores (10) remove allocs (14) gc irgs (8) gc irgs primitive (8) simplify consts (9)	simplify consts (5) replace return values (3) gc irgs primitive (7) remove allocs (3) gc irgs (4) optimize load store (3)
opt if conv	**gc irgs**	**replace arithmetic**	**replace cond selectors**
opt blocks (5) simplify consts (3) replace cond selectors (3) opt tail rec (3) normalize n returns (3) gc irgs (5)	simplify consts (6) gc irgs primitive (17) opt parallelize mem (3) optimize cf (3) gc entitites (6) remove allocs (6) remove stores (6)	remove allocs (5) gc irgs (4) replace proj primitive (4) simplify consts (4) optimize load store (3) normalize n returns (4) remove stores (3)	remove allocs (3) remove void calls (5) gc irgs (4) replace proj primitive (6) gc irgs primitive (4) optimize load store (3) remove stores (4)

Table 1: Analysis of successful sequences The table shows which passes are most likely to succeed following a successful invocation of the pass in the left column. We only show passes that have succeeded at least 3 times

normalize n returns	**remove allocs**	**replace return values**	**optimize load store**
remove allocs (4)	gc irgs primitive (20)	replace cond selectors (3)	gc irgs (4)
combo (3)	replace return values (4)	replace proj primitive (5)	gc irgs primitive (8)
replace cond selectors (3)	replace proj primitive (6)	gc irgs primitive (3)	replace return values (3)
construct confirms (4)	simplify consts (6)	remove allocs (5)	replace proj primitive (3)
opt if conv (4)	remove void calls (3)	opt blocks (3)	remove allocs (7)
gvn pre (3)	opt parallelize mem (3)	normalize n returns (4)	opt parallelize mem (3)
gc irgs (3)	gc irgs (10)	simplify consts (4)	optimize cf (3)
remove stores (3)	remove stores (4)	opt tail rec (3)	opt tail rec (3)
			replace arithmetic (3)

remove stores	**simplify consts**		
optimize cf (3)	remove void calls (5)		
gc irgs (5)	remove allocs (12)		
gc irgs primitive (7)	conv opt (3)		
remove allocs (9)	opt tail rec (3)		
simplify consts (5)	optimize load store (6)		
normalize n returns (3)	normalize n returns (5)		
optimize load store (5)	replace return values (5)		
replace proj primitive (4)	gc irgs (11)		
replace return values (4)	replace proj primitive (4)		
	gc irgs primitive (12)		