ROYAL INSTITUTE
OF TECHNOLOGY

Random Testing of Code Generation in Compilers

BEVIN HANSSON

Master's Thesis
Supervisor: Roberto Castañeda Lozano
Examiner: Christian Schulte

TRITA-ICT-EX-2015:77

Abstract

Compilers are a necessary tool for all software development. As modern compilers are large and complex systems, ensuring that the code they produce is accurate and correct is a vital but arduous task. Correctness of the code generation stage is important. Maintaining full coverage of test cases in a compiler is virtually impossible due to the large input and output domains.

We propose that random testing is a highly viable method for testing a compiler. A method is presented to randomly generate a lower level code representation and use it to test the code generation stage of a compiler. This enables targeted testing of some of the most complex components of a modern compiler (register allocation, instruction scheduling) for the first time.

The design is implemented in a state-of-the-art optimizing compiler, LLVM, to determine the effectiveness and viability of the method. Three distinct failures are observed during the evaluation phase. We analyze the causes behind these failures and conclude that the methods described in this work have the potential to uncover compiler defects which are not observable with other testing approaches.

Referat

Kompilatorer är nödvändiga för all mjukvaruutveckling. Det är svårt att säkerställa att koden som produceras är korrekt, eftersom kompilatorer är mycket stora och komplexa system. Kodriktigheten inom kodgenereringsstadiet (registerallokering och instruktionsschemaläggning) är särskilt viktig. Att uppnå full täckning av testfall i en kompilator är praktiskt taget omöjligt på grund av de stora domänerna för in- och utdata.

Vi föreslår att slumpmässig testning är en mycket användbar metod för att testa en kompilator. En metod presenteras för att generera slumpmässig kod på en lägre representationsnivå och testa kodgenereringsstadiet i en kompilator. Detta möjliggör riktad testning av några av de mest komplexa delarna i en modern kompilator (registerallokering, instruktionsschemaläggning) för första gången.

Designen implementeras i en toppmodern optimerande kompilator, LLVM, för att avgöra metodens effektivitet. Tre olika misslyckanden observeras under utvärderingsfasen. Vi analyserar orsakerna bakom dessa misslyckanden och drar slutsatsen att de metoder som beskrivs har potential att finna kompilatordefekter som inte kan observeras med andra testmetoder.

Contents

Chapter 1

Introduction

Compilers form the backbone of all modern software development and are one of the most mature branches of computer science. Without compilers, programmers would be forced to write all of their code in assembly; a tedious endeavor for most. Compilers form an abstraction between the programmer and the computer, allowing the programmer to focus on the behavior of the code rather than optimization and the minutiae of computer architectures.

However, the great complexity of compilers is a liability. The chance of a compiler introducing a bug into a program increases as they perform more and more complex tasks. These program bugs can be difficult to find as they are a result of wrong-code compilation and not of user mistakes.

In order to discover these obscure compiler defects, compilers must have their code thoroughly verified and tested. However, with modern compilers consisting of hundreds of thousands of lines of code, this is a tedious job at best and a virtually impossible one at worst.

This thesis presents a method to generate random machine-level code in order to rapidly and automatically perform random testing on the code generation components of a modern compiler. Random testing of a compiler on this level has, as far as we are aware, not been attempted before. Similar types of testing on a higher abstraction level have been attempted with successful results [1, 2], but testing at a lower level of abstraction allows for more targeted, direct testing of the code generation components of the compiler.

In order to verify the validity of the methods presented in this report, a case study is performed on LLVM, a state-of-the-art optimizing compiler with Hexagon, an embedded digital signal processor architecture as the target architecture.

1.1 Background

Machine code for computer architectures is commonly encoded in the form of a sequence of instructions operating on registers in the CPU, constants and memory. The human-readable format of these instruction sets is called *assembly code*. Although there are many commonalities between them, these instruction sets are generally specific to a single computer architecture and are not cross-compatible; instructions for one architecture cannot be executed on another. This limitation, along with the fact that writing code in assembly is a tedious and time-consuming task, necessitates the existence of programming languages with a higher level of abstraction. However, these high-level programs cannot be executed directly on computer architectures.

Compilation is the process of converting a program from a high-level programming language to low-level machine code [3]. The processor architecture being compiled for is called the *compilation target* or simply the *target*. To facilitate this conversion, many different stages of transformation are required.

One of the final stages of compilation is code generation, in which the code is converted from a target-independent form into a target-dependent one. This stage encompasses several smaller stages: *instruction selection*, where instructions from the target instruction set are chosen to match the behavior of the target-independent code; *register allocation*, in which the architecture's register and memory resources are allocated to the program; and *instruction scheduling*, where the assembly instructions are reordered and/or grouped to optimize the use of the processor's resources and facilitate better performance on computer architectures that can take advantage of instruction-level parallelism.

As modern optimizing compilers must perform many different analyses and transformations on the code, the complexity of these compilers is large. As the complexity grows, so does the risk for defects in the code of the compiler [4]. Code generation is especially sensitive, as register allocation and instruction scheduling have a strong effect on the performance of the code [5].

Random testing is a testing technique where random, independent test data is generated and subsequently fed to the system under test to uncover defects in the software. As the data is completely random, it is difficult to determine whether the result of any given test case is correct or not. As a result, random testing must employ an *oracle*; a secondary test that can be used to verify the testing result. Oracles for random testing are commonly constructed from a simpler or alternative version of the system [6]. Since covering every possible test case in a compiler can be difficult due to its complexity, we believe random testing is a viable approach for testing compilers.

1.2 Research question

Can random generation of machine code uncover defects in the code generation stages of a modern optimizing compiler?

1.3 Goal

The goal of this work is to investigate whether is it possible to uncover defects in a modern compiler by performing random testing directly on the code generation components of the compiler. For this purpose, a method is devised to generate random, human-readable machine code and use it to test a modern compiler.

1.4 Methodology

Initially, we formulate the hypothesis that randomly testing a compiler on a lower level (post-instruction selection) of abstraction can potentially expose defects in a modern compiler. A method for generating random test cases in the form of code is designed, and subsequently implemented in a modern compiler as a case study. This implementation is then run for a long period of time to collect data on which cases succeed and fail. Success is determined by the successful termination of the compiler and the emission of a compiled assembly file. Failure is determined by unsuccessful termination. The failing cases are categorized and investigated to determine the cause of the failure.

1.5 Ethics and sustainability

Verifying that compilers are producing correct code is important from a sustainable point of view. If obscure defects caused during compilation rather than by programmer error are discovered in a software product, producing an update for this product can be very time and resource consuming. Furthermore, if the software is located in a system which is either very difficult to update or cannot be updated at all, companies may be forced to perform costly recalls. By improving testing of compilers, we can prevent wasting resources on this type of issue.

The notion of testing any system has ethical considerations in that engineers should strive to release products that do not cause harm to people. By testing compilers, we attempt to minimize any harm that could occur through miscompilation, hopefully making software safer for people to use.

1.6 Limitations

The method for generating random code is meant to be as general as possible; in other words, it is suitable for any reasonable machine code representation. However, the implementation is intended to only produce code for the Qualcomm Hexagon V4 architecture. The testing method is also general, but as with the generation implementation it is specifically targeted at the LLVM compiler framework.

Certain instructions are not used during generation to simplify the generation process. These instructions include intrinsic functions, memory barriers and other special instruction types. Section 6.1 goes into further detail on future work to eliminate these limitations.

1.7 Outline

Chapter 2 presents the background information of the relevant topics of the thesis. Chapter 3 details the theoretical design of the method developed in this thesis. Chapter 4 details a case study with a functional implementation of the method, along with methods for testing the validity of the method. Chapter 5 presents the results of the case study. Chapter 6 concludes the thesis work and covers future improvements and opportunities.

Chapter 2

Background

This chapter covers the necessary background knowledge for this report. Various topics pertaining to compilers, such as code generation are detailed. The subject of software testing is introduced. Abstract representations of computer programs and machine code are also described.

2.1 Compilers

At the most abstract level of design, a compiler consists of two parts: a front-end and a back-end. The task of the front-end is to take a program – commonly written in a high-level language – and convert it into a target-independent intermediate representation. This intermediate representation (IR) is subsequently fed to the backend, which performs target-specific optimizations, generates machine code and emits a textual or binary output for the target architecture [3, 7].

2.1.1 Code generation

The code generation stage is the largest part of the backend. It is during this stage that the IR is lowered from a target-independent representation to a target-dependent one, typically in the form of instructions of the target architecture. Code generation commonly consists of three smaller stages; instruction selection, register allocation, and instruction scheduling [3, 8].

Instruction selection is the process of selecting target-dependent machine instructions based on the operational semantics of the target-independent IR. This can be done with a tiling approach, where subtrees of IR are matched against known patterns to produce machine instructions [9, 10].

Register allocation is the process of assigning physical hardware register space to symbolic variables (typically called *temporaries* or *virtual registers*). As the number of physical registers in a processor is often limited, the compiler may be forced to store variables in memory instead of in registers. This is called *spilling* [11].

Instruction scheduling is the process of analyzing the data dependencies between instructions to determine an optimal ordering of operations. This is important for superscalar or VLIW processor architectures, where multiple instructions can be executed in parallel to increase performance [3, 5].

In order to ensure that the resulting code operates according to its specification and is as efficient as possible, it is important that each code generation stage is free of any defects that could cause the compiler to emit erroneous code.

2.2 Machine code representation

Representing machine code on a higher level requires models which are capable of encapsulating two properties of a program: *data flow* and *control flow*. Data flow is the representation of data or state as a procedure is executed. Higher level program representations (like C) use variables to model data flow. Control flow is the representation of the behavior of a program or procedure with respect to its state. In C, this is modeled with control flow statements such as `if` and `while`.

On a lower level, computer programs consist of a series of *instructions* which perform operations on values stored in either memory or in registers. Machine code procedures (and by extension, programs) consist of *basic blocks*; groups of instructions bounded by the control flow of the procedure. Each basic block consists of a series of instructions unbroken by control flow. By definition, execution of a basic block must begin at the first instruction in the block, and each instruction must be executed exactly once for that execution, though not necessarily in the given order. At the bottom of the basic block, control is passed to another program location, either through a jump or branch instruction, or through a return instruction (in which case control leaves the procedure completely).

2.2.1 Instructions and temporaries

Machine code instructions are typically constructed in the form of $D = op\ U$ where op is a mnemonic representing an instruction, U is a set consisting of inputs to the instruction and D is the set of resulting values. The inputs (called *uses*) are commonly registers, immediate values or memory locations. The outputs (called *defines* or *defs*) are stored in registers.

As all physical computer architectures have a limited number of registers, pre-register allocation machine code representations store and read values from a very large imaginary set of registers, commonly called *temporaries* or *virtual registers*. During register allocation, each temporary is assigned a physical register in the CPU. If there are not enough registers to store temporaries in, some temporaries may have to be stored in memory to free up space in registers which would otherwise be occupied.

2.2.2 Control flow graphs

The most common method of representing intra-procedural control flow is the *control flow graph*. A control flow graph (or CFG for short) is a directed graph where each node represents a basic block[1] and each edge represents control flow (branches or jumps) [3]. CFGs are a very common tool in program analysis and optimization.

The edges of a CFG model the branching of control flow between basic blocks. Nodes which control is transferred to are called *successors* and nodes which control is transferred from are called *predecessors*. One of the nodes in the CFG is designated as the *entry node*. This node is the entry point of the procedure, and therefore has zero predecessors [3].

Definition 1. *A control flow graph is a directed graph $G = (N, E, s)$ where N is the set of nodes in the graph, E is the set of edges in the form of tuples (x, y) where $x, y \in N$, and $s \in N$ is the entry node.*

Dominance is a property of nodes in a control flow graph. A node n in a graph *dominates* another node m if every path from the entry node to m must pass through n. By this definition, every node in a flow graph dominates itself, and the entry node dominates all nodes. Dominance is an important property for optimizing compilers and is used in several analysis methods [12, 13, 14].

Reducible control flow graphs

Control flow graphs can be divided into two classes depending on their structure: reducible graphs and irreducible graphs. Reducible graphs are susceptible to a number of analyses and optimizations, which makes them desirable in compilation. Common control flow structures like if-then-else and while-loops

[1]The terms node and basic block are used interchangeably in this report unless otherwise specified.

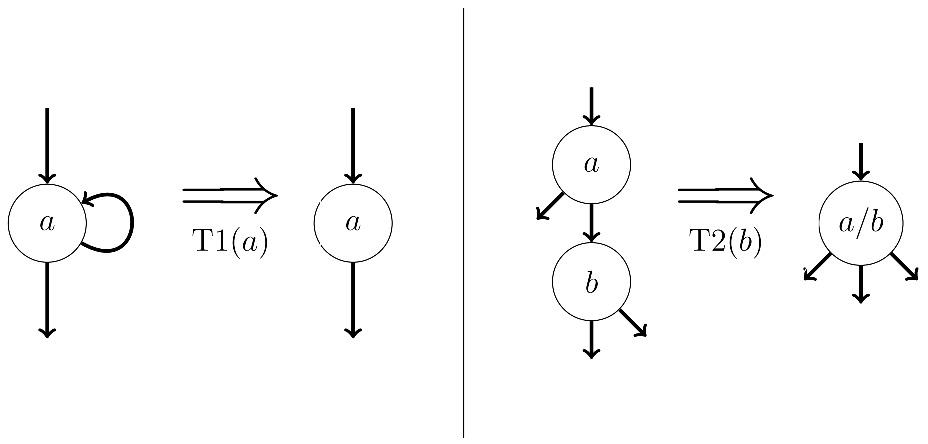

Figure 2.1. The T1 and T2 reducibility transforms. T1 eliminates self-loops and T2 eliminates nodes with single predecessors.

will always produce reducible control flow graphs [15]. Misusing language features such as `goto` in C, however, can produce irreducible flow graphs [16]. A definition of graph reducibility is given by [15, 14].

In Definitions 2 and 3, the head of an edge is the node which the edge points *to* and the tail of an edge is the node which the edge points *from*.

Definition 2. *The set of back edges in a flow graph consists of all edges whose heads dominate their tails. The set of forward edges consists of all edges which are not back edges.*

Definition 3. *A flow graph is reducible if the subgraph consisting of its forward edges is acyclic and there is a path from the entry node s to every other node in the graph.*

An analysis method for determining graph reducibility is given by Ullman and Hecht [15]. They introduce two transforms, T1 and T2, that can be iteratively applied to the nodes of a control flow graph. The transforms are exemplified in figure 2.1. By applying these transforms to a flow graph, it will be successively reduced until either a single node remains or the transforms can no longer be applied to any nodes. If the former occurs, the graph is reducible. Otherwise, it is irreducible.

Definition 4. *Consider a graph $G = (N, E, s)$ with node $n \in N$. If there is an edge $(n, n) \in E$, the result of* T1(n) *on graph G is a subgraph where the edge (n, n) is removed from E. In other words, the self-loop on n is eliminated.*

Definition 5. *Consider a graph $G = (N, E, s)$ with nodes $n, m \in N$ and $n \neq m$. If n is the only predecessor of m, the result of* T2(m) *on graph G is* *where m is removed from N, all successor edges of m are folded into n and the edge (n, m) is removed from E. Any duplicate edges are removed.*

2.2.3 Static single assignment form

Static single assignment (or SSA) is a form of machine code which combines aspects of control and data flow. Having a procedure in SSA form simplifies analysis and transformation and allows for many efficient optimization techniques [17].

The primary requirement of SSA form is that *each temporary in a procedure be defined only once* and *each temporary be defined before it is used.* If an instruction produces a result, it must define a new temporary instead of reusing an old one. In a procedure with no control flow, fulfilling this requirement is trivial; all that is needed is a value numbering algorithm [18]. However, a procedure with branching or looping control flow cannot define each temporary only once, as a single value in a basic block may be sourced from several different predecessor blocks; see figure 2.2 for an example.

```
int function(int a) {
  int b;
  if (a < 5) {
    b = a + 1;
  }
  else {
    b = a - 1;
  }
  return b;
}
```

```
int function(int a) {
entry:
  t1 = a;
  if (t1 >= 5) jump bb2;
bb1:
  t2 = add t1, 1;
  jump bb3;
bb2:
  t3 = sub t1, 1;
bb3:
  t4 = φ(t2, t3);
  return t4;
}
```

Figure 2.2. A code snippet where the value in a variable can be assigned from multiple locations based on program state. The left shows a C-like code snippet and the right a pseudo-machine code example of the snippet. In the example, variable `b` (`t2`, `t3` and `t4`) will have a different value depending on the value of `a` (`t1`).

This problem is solved by the use of *ϕ-functions*: special, virtual instructions that encode congruences between temporaries in different basic blocks. If the value of a temporary is taken from different basic blocks depending on control flow, a ϕ-function is added in order to determine which temporary to copy the value from based on which basic block control was passed from. In figure 2.2, temporary `t4` is congruent with `t2` and `t3` and will take its value from either depending on the evaluation of the conditional jump.

2.3 Software testing

Testing is the process of uncovering defects in a system or product. These defects can be a result of incorrect system requirements or mistakes made in the design and implementation phases of the system. Implementation defects in a software system are often referred to as bugs and manifest in the form of mistakes in the source code of the system. If a system contains an excess of defects, the chance of experiencing a failure – a state where the system fails to perform its task correctly – increases [19].

There are many ways of testing a software system, but in most cases it involves providing some test data as input to the *system under test* (or SUT), receiving a resultant output data and then verifying that the output is correct for the given input. This verification step is performed with an *oracle*; an alternate system capable of telling whether or not the SUT has performed its task correctly [20].

2.3.1 Oracles

In testing, oracles are used to validate the results of tests performed on a system. The quality of the oracle used to test a system weighs heavily on the test results. Depending on the system, an oracle can be run before, during or after execution of the system under test. Oracles can also be split into distinct design types, including (but not limited to) true, heuristic, and assertion-based oracles [20].

A *true oracle* is a reproduction or alternate implementation of the SUT. For a SUT to pass a test, it should produce the same output as the true oracle for a certain input. True oracles are powerful as they attempt to model the exact behavior of the system, but constructing a true oracle is difficult for any system beyond a simple algorithm. A true oracle for a complex software system must often be of equal complexity to the system, which increases the probability of introducing defects into the oracle itself. An example of a true oracle for a sorting algorithm can be another, independent implementation of a sorting algorithm [20].

A *heuristic oracle* is an oracle that attempts to model the correlation between input data and output data to determine correct system behavior. To accomplish this, heuristic oracles use a simpler algorithm than the system under test to provide a reasonable guess at the validity of the results. Heuristic oracles are not as reliable as true oracles, but they are easier to implement. An example of a heuristic oracle for a sorting algorithm can be a routine that verifies that the output is ordered and contains the values given in the input [20].

Assertion-based or *specification* oracles are a type of oracle that utilizes assertions, preconditions and/or postconditions to determine if a system is operating correctly. Assertions are a software mechanism used to assert certain conditions at specific points during program execution. The conditions represent the required system state at the specified points. If a condition does not hold, the program will abort its execution as it has reached a state which will result in an eventual failure. An assertion-based oracle for a sorting algorithm is the same as the heuristic one; given an input and an output, the oracle would examine if the output data matches the required postcondition [21].

2.3.2 Random testing

As compilers consist of a large number of interconnected stages that perform complex analyses and optimizations, constructing tests and providing an oracle for a compiler can be difficult. The domain of both input (source code) and output data (machine code) is virtually infinite in size, making it difficult to construct test cases for all possible system states and determining that the compiler has produced correct machine code for the given input.

For this reason, *random testing* (or fuzzing) is a viable approach for testing compilers. By constructing vast amounts of random test cases and using them as input to the system under test, the system can be tested very quickly and with less human input than a manual testing approach [6]. The testing approach can be nearly fully automated with the addition of test minimization and delta debugging [22, 23].

2.4 Random testing of compilers

As generation of random test data (and in this specific case, code) is an important part of random testing, it is a widely studied topic [24, 25, 26]. A technique worthy of note is property-based testing [27]. It employs an approach parallel to assertion-based oracles; instead of using preconditions and postconditions to verify testing results, the assertions are used to generate test cases and suites. The tool QuickCheck is an implementation of a property-based tester [28].

In their paper on graph reducibility, Ullman and Hecht present a simple graph grammar [29] to express reducible subgraphs for generating random control flow [15]. Yang *et al.* [2] use conditional filters and probability tables to produce random C programs. The compiler framework LLVM includes a tool called *llvm-stress*, designed to generate random intermediate representation code for LLVM [30].

Using random testing on compilers has been attempted in the past, with excellent results [31, 2]. One notable example of this is Csmith, a generator of random, valid (according to the C99 specification) C programs [2]. Csmith has found hundreds of bugs in mainstream compilers like GCC and LLVM by employing random differential testing. The generated programs produce an output on the standard output stream. The code is compiled with multiple, independent compilers and the results of executing the programs are compared. If there is a discrepancy, there is a potential bug.

Chapter 3

Design

This chapter details the general design of the random code generation methods presented in this report.

3.1 Random code generation

The random code generation method in this paper is divided into three primary steps: architecture analysis, control flow generation and data flow generation.

Architecture analysis is the process of analyzing the properties of the computer architecture being targeted by the random code generator. This analysis determines which instructions are going to be used in generation, the properties of the instructions as well as the different types of registers that are available.

Control flow generation is the process of creating a random control flow graph. We establish a number of rules which the generated CFG must adhere to in order to simplify the generation algorithm.

Data flow generation is the final step in constructing the random machine code program. In this step, instructions and temporaries are randomized and inserted into each basic block to produce the data flow of the generated procedure.

3.1.1 Architecture analysis

Architecture analysis is the first step of random code generation. In this step, the entire instruction set of the target architecture is enumerated and sorted into various categories to simplify the later stage of data flow generation. This step is required to prevent the generator from producing invalid code - code which is not in SSA form nor follows the rules of CFG generation. While the purpose of the random code generator is to produce random code, code that is *too* random (instructions with incorrect operands, temporaries that are defined multiple times or not at all) would make little sense to both the compiler and the tester.

Instruction categories

The instructions of the generation target is split up into the five following categories:

Operators are the most common instruction type. These are instructions that take at least one temporary as input and provide a result in another temporary, and have no other side effects such as reading or writing to memory. This group is where instructions such as `add` (addition) and `mul` (multiplication) belong. However, it is not required that operators actually modify the inputs they are given; a copy from one temporary to another still belongs in the operator category.

Branches are instructions that alter the control flow of the procedure. These instructions are placed at the bottom of each basic block depending on the layout of the control flow graph. Branch instructions are further split into *conditional* and *unconditional* branches. The meaning of these categories can be inferred from the name; conditional branches only branch depending on some computed condition, while unconditional branches will always branch to their specified target.

Returns are instructions that terminate the procedure. These instructions are placed at the end of a basic block that has no successor blocks.

Immediate moves are instructions that introduce constants to the data flow. These are instructions that take a single immediate value and store it in a temporary.

Memory instructions are instructions that either read from or write to memory. The memory location can be on the stack, a global variable or a pointer.

Instructions that do not fall into any of these categories are not considered for random code generation, as they might have effects that are either not modeled by the compiler's intermediate representation or could result in undefined behavior. However, we believe that most common instructions fall into these categories.

Operand analysis

Each instruction has a number of operands. These operands can take a number of different types of values, the most common of which are temporaries/registers and immediates. However, in order to produce valid code, it is important that the operands are only assigned values that correspond to their operand type, and that the values are in the correct range.

Architectures commonly have sets of registers that can be used for different purposes or differently sized values. For example, the Hexagon architecture possesses 3 primary general purpose register sets: 32-bit integer, 64-bit integer and 1-bit predicate registers. Temporary operands must only be assigned temporaries which belong to the register set specified in the architecture definition, or invalid code may be produced.

Immediate values are also restricted by the architecture. For every immediate operand in an instruction, three properties are considered: size, signedness and alignment. For example, an instruction could specify that it can only take an immediate that is unsigned, 8 bits large and aligned on the two least significant bits[1]. If these constraints are not considered during random code generation, the generator might output code in which the immediate values do not fit in their instructions.

3.1.2 Control flow generation

Control flow generation is the process of constructing an arbitrary control flow graph. The algorithm must be capable of generating valid graphs that can model any generic procedure, but the graphs cannot be *too* random, as overly random graphs may lack properties that would make them susceptible to analysis and optimization. To ensure that these conditions hold, we establish a number of rules for the control flow graph generation algorithm:

A) There must always be at least one node with zero successors;
B) no node in the graph may have more than two successors; and
C) the CFG must be reducible.

Rule A guarantees the existence of at least one exit node. In some approaches, rule A is more restrictive and only allows exactly one exit node [32, 33, 13]. It is commonly regarded as good practice to have a single exit location in a procedure [34], but multiple exit locations do occur in real world code. Therefore, we use the laxer requirement.

[1]By *aligned*, we mean that the value must be a multiple of the magnitude of the next bit. From a binary perspective, this means that the two least significant bits must be 0.

It is also possible to have procedures without any exit nodes at all, as a result of an unterminated infinite loop. However, these constructs rarely occur in real code except in very simple programs and examples. Certain types of programs, such as servers, employ a type of loop colloquially referred to as *pseudo-infinite loops*. These resemble infinite loops in the sense that they do not have a standard termination condition, but they can be terminated through other means (exceptions, break statements).

Rule B is not a strict requirement for control flow graphs. Certain control flow structures allow for more than two successors from a single node. In C, this can be accomplished with a `switch` statement, resulting in an indirect jump or a computed goto. However, this construct is not considered in this project for simplicity's sake. Multiple successors are not explicitly required to model this type of branching; chains of binary conditional jumps will suffice on architectures that do not support the required jump operations.

Inverse reducibility transformation

Inverse reducibility transformation is presented as a method for generating arbitrary reducible control flow graphs by inverting the Ullman-Hecht T1-T2 analysis described in Section 2.2.2. These transforms are capable of reducing a reducible CFG to a single node. It stands to reason that if applied in the other direction, they can expand a single node into an arbitrary reducible CFG.

As Ullman and Hecht point out in their paper, similar inverses of these transforms have been used in other, earlier works for generating random flow graphs [15]. However, our report defines a constrained and simple version of the inverse which can easily be applied to random code generation.

The definition of inverse T1 is simple. The operation of T1 on a node in a graph eliminates self-loops on that node. Therefore, the $T1'$ on a node would instead *introduce* a self-loop. As a graph does not allow for duplicate edges, a node which already has a self-loop is not a candidate for transformation. The rules for control flow graphs described above must also be respected. To ensure that the rules are not broken, the following restrictions are placed on which nodes $T1'$ can be applied to:

A) $T1'$ cannot be applied to the entry node, as this would add a predecessor to the node, resulting in a graph where no node has zero predecessors.

B) $T1'$ cannot be applied to an exit node. If there is only one exit node in the graph, the resulting graph would have no exit nodes.

C) $T1'$ can only be applied to nodes with a single successor. Then, the largest number of successors a node can have after applying $T1'$ will always be two.

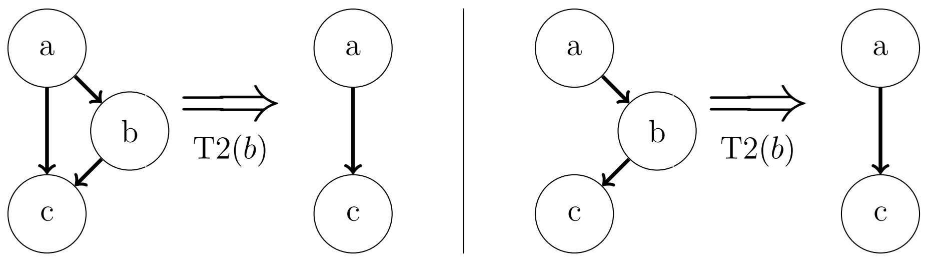

Figure 3.1. This example demonstrates how the result of T2 on two different graphs is the same. Thus, T2 is not injective and is harder to invert than T1.

As given by the definition of reducible graphs in Section 2.2.2, a flow graph is reducible if the subgraph consisting of its forward edges is acyclic and all nodes in the subgraph are reachable from the entry node. Since all nodes in a flow graph dominate themselves, a self-loop is a back edge and does not affect the subgraph of forward edges. Therefore, adding a self-loop to a reducible graph will not affect its reducibility.

Defining the inverse of T2 is slightly more difficult. For the purpose of this explanation, x and n are graph nodes, where n is the only predecessor of x. x does not necessarily have to be in the graph. Nodes to which a transform can be applied to are called *candidate nodes.* x is a candidate node for T2, and n is a candidate node for T2′. Performing T2(x) will result in the elimination of x according to the rules of T2. However, performing T2′(n) will instead result in x being added to the graph.

Performing T2 on node x in a graph will eliminate x from the graph. Therefore, it is not possible to define the inverse of T2 as an operation on node x, as it has yet to be added to the graph. Since the node being added must have a single predecessor, the inverse of T2 on a given node n is instead defined as an operation that creates the node x and appends it to n.

Furthermore, transform T2 is not injective (as shown in Figure 3.1); the result of performing T2′ on a graph is not unique. This means that for graphs G and G' where G and G' are CFGs, and $\text{T2}(n \in G) = G'$, there can exist multiple graphs G. As such, there is no single function T2′ so that $\text{T2}(\text{T2}'(n \in G)) = G$.

However, by analyzing the possible combinations of node elimination that can occur based on the previously established rules of CFG generation, it is possible to construct an inverse of T2 with multiple resulting graphs. Which combinations are possible depends on the number of successors of the selected node. The combinations are summarized in figure 3.2.

If a candidate node n has zero successors, the transformation is trivial. T2 would normally have folded x into n, but since n would have no successors

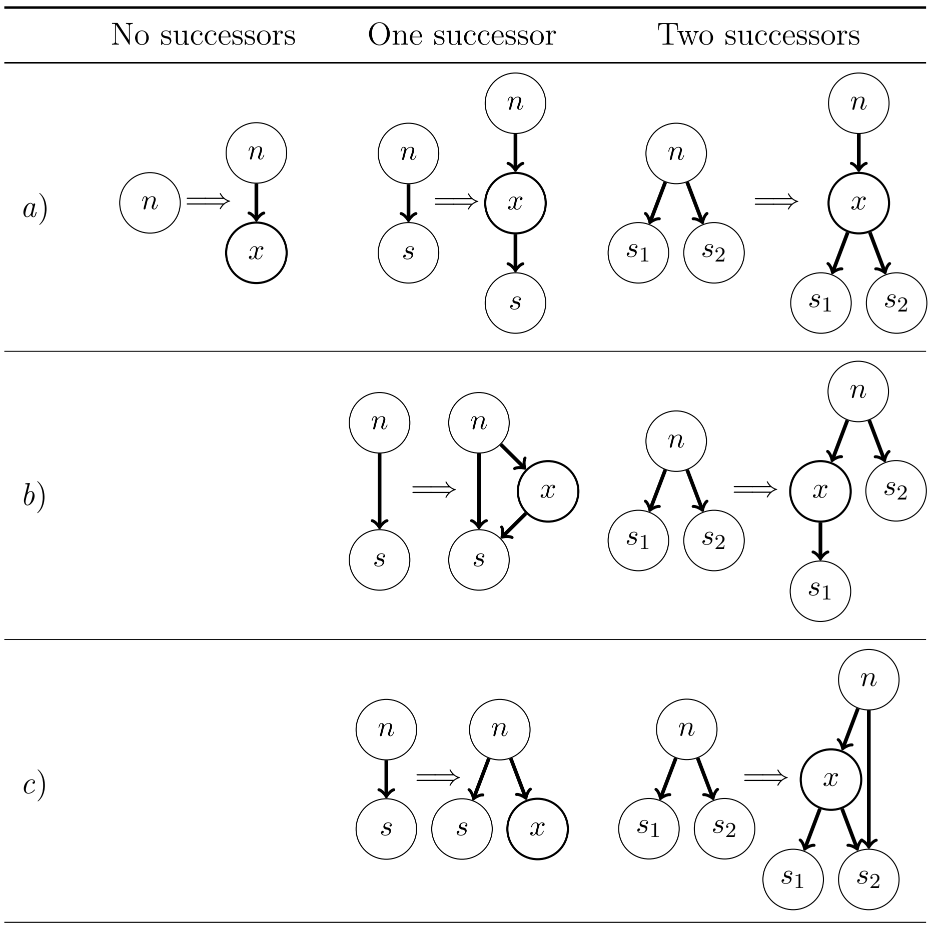

Figure 3.2. The possible T2′ transforms for zero, one and two successors. n is the node which the transform is being applied to, x is the node being added to the graph and s, s_1 and s_2 are the successors of n. Note that if any of the edges to the successor nodes are self-loops or back edges, these examples will no longer apply visually.

after performing T2, it follows that x would have no successors as well. Thus, the operation is to add x to the graph and create an edge $n \rightarrow x$.

If a candidate node n has one successor s, there are three possibilities for transformation. In all three cases, x is added to the graph and the edge $n \rightarrow x$ is added. The possibilities are T2$'a$, T2$'b$ and T2$'c$ for one successor:

a) The edge $n \rightarrow s$ is removed and the edge $x \rightarrow s$ is added. This produces a chain $n \rightarrow x \rightarrow s$.
b) The edge $n \rightarrow s$ is kept. This produces a conditional branch from n to x or s.
c) x gets no edges other than $n \rightarrow x$. This produces a new exit node.

If a candidate node n has two successors s_1 and s_2, there are also three possibilities. As with the case of one successor, x is added to the graph and the edge $n \rightarrow x$ is added. The possibilities are T2$'a$, T2$'b$ and T2$'c$ for two successors:

a) The edges $x \rightarrow s_1$ and $x \rightarrow s_2$ are added. Edges $n \rightarrow s_1$ and $n \rightarrow s_2$ are removed. This produces a similar structure as before, but with x between n and its former successors.
b) The edge $x \rightarrow s_1$ is added. The corresponding edge $n \rightarrow s_1$ is removed. This produces a chain with x between n and s_1. Note that which successor is assigned to which node is arbitrary; either n precedes s_1 and x precedes s_2, or vice versa.
c) Edges $x \rightarrow s_1$ and $x \rightarrow s_2$ are added. The edge $n \rightarrow s_2$ is removed. This produces two conditional branches, one from n to x or s_1 and another from x to s_1 or s_2. As in the case of *b*, the assignment of successor is arbitrary.

These different combinations of possible transformations all conform to the rules for CFG generation. Invalidating any possible exit nodes is impossible, as the transformations only modify the predecessor edges of the successor nodes s, s_1 and s_2; the successor edges are never modified. The number of successors of these nodes will never be altered. The exception is when n has no successors, in which case x will become the new exit node. As shown in figure 3.2, none of the transformations will result in a node having more than two successors.

Flow graph generation is briefly discussed in Ullman and Hecht's paper [15]. They present a graph grammar with which to represent simple control flow concepts (while and if-else), and then prove that all graphs constructed with this grammar are reducible. This approach is intuitive but not exhaustive. It only covers a limited subset of reducible graphs and does not consider many common control flow structures (if, do-while, break, continue).

3.1.3 Data flow generation

Data flow generation is the process of constructing the contents of basic blocks of the procedure being generated. This process involves generating random instructions, temporaries/registers, immediate values and memory locations and inserting these into the basic blocks. The method described in this paper is a *bottom-up* approach to generating machine code in SSA form.

We choose SSA form as the code representation for two reasons. First, it is very simple to generate code in SSA form as there is no need to manage and allocate register usage. Second, as mentioned in Section 2.2.3, many optimizations and analyses require the code to be in SSA form. As the purpose of the random generation is to test the compiler, it is appropriate that the code be in a form that will maximize the coverage of the generated test cases.

The method works by iteratively adding defining instructions for temporaries that are being used, but have yet to be defined. Each temporary can only be used and defined in a single basic block, with the exception of ϕ-instructions which may use temporaries across block boundaries. For every temporary in a basic block that is being used but has not been defined, we attempt to resolve this use with different types of instruction.

There are some differences between the SSA form described here and what is considered standard SSA. The only restriction standard SSA places on using temporaries is that they must be defined before they are used. In other words, the block in which a temporary is defined must dominate all blocks where it is used. This means, for example, that temporaries defined in the entry block can be used anywhere else in the procedure. In our SSA, use of temporaries is restricted to the block in which they are defined, unless it is being used for a ϕ-function.

This approach to SSA is similar to the stricter LSSA form described in [35]. In LSSA, temporaries cannot be used across basic block boundaries as they can in SSA. Instead, congruences must be established between all temporaries in all basic blocks. The difference between LSSA and the form used in this report is that temporary uses across single-predecessor edges do not require a ϕ-function, as described in Section 3.1.3.

The data flow generation method is broken up into three steps: branch generation, where branching instructions are added to the basic blocks; instruction generation, in which the basic blocks are filled with random instructions; and cleanup, where the procedure is cleaned to remove extraneous instructions.

Branch generation

The first step in data flow generation is to finalize the earlier control flow generation stage by adding the necessary branching instructions to each basic block. Section 3.1.1 describes three categories of branching instructions; conditional branches, unconditional branches and return instructions. Random instructions from these categories are selected according to the CFG and inserted into the basic blocks.

There are three cases to consider: basic blocks with zero, one and two successors. Basic blocks with one successor are trivial. As the branch taken will always be to that single successor, it is enough to add an unconditional branch at the bottom of the basic block. No temporary uses will be added as unconditional branches do not depend on program state.

A basic block with zero successors is considered to be an exit node of the procedure. A return instruction is inserted at the end of the basic block and a register is added as a use operand according to the calling convention of the target architecture. This will ensure that the procedure returns a value.

If a basic block has two successors, both an unconditional and a conditional branch must be added. As the generated CFG does not encode any information regarding which successor is the conditional, one of the successors is chosen at random to be the conditional target and the other is chosen as the unconditional one. Any necessary temporary or immediate operands are added to the conditional branch according to the instruction definition. These added temporaries form the foundation of the data flow in the basic block, as they will be the first temporaries to be defined in the next step.

Instruction generation

The second step of data flow generation fills the basic blocks with random instructions. As mentioned in Section 2.2.1, temporaries in SSA form can be defined only once and defined before they are used. There is no restriction on the number of times a temporary can be used. As a result of the bottom-up construction of the basic blocks, all blocks will contain a number of temporaries that are being used by instructions, but have not been defined by any instruction. These temporaries will be *resolved* with a number of techniques in order to build the data dependency graphs of the blocks.

Figure 3.3 shows an example of the instruction generation method for one basic block. For each iteration of the generation algorithm, each basic block is visited and the used-but-not-defined temporaries are enumerated. Then, a method to resolve the temporary is chosen. Temporaries can be resolved with an operator, an immediate move, a memory load, a ϕ-function, or they can be

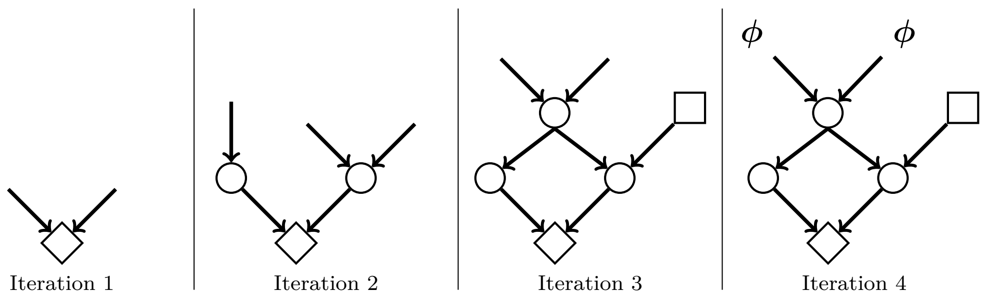

Figure 3.3. The data dependence graph of a basic block being built from the bottom up. Circles are operators, squares are immediate moves and diamonds are conditional branch instructions. $\boldsymbol{\phi}$ represents temporaries defined by ϕ-functions. Every edge tail represents a temporary def and every edge head a use.

replaced by another temporary altogether. Resolution of a temporary can also be postponed to a later iteration to produce a more convoluted dependence graph. If an operator, an immediate move or a memory load is used to resolve a temporary, only instructions which define a temporary of the type being resolved may be chosen. When all temporaries in all basic blocks have been resolved, the instruction generation stage is complete.

Resolution methods

Resolving a temporary with an operator is straightforward. An operator capable of defining a temporary of the correct type is randomly selected and inserted at the top of the basic block. As all operators must have at least one temporary use, resolving a def with an operator will always expand the dependence graph by producing more temporaries to be defined.

Resolving a temporary with an immediate move introduces a constant into the data flow. An immediate move instruction does not use any temporaries, and will therefore reduce the expansion of the dependence graph as shown in iteration 3 of figure 3.3.

Memory load resolutions are capable of both expanding and reducing the expansion of the dependence graph. This depends on whether or not the load instructions take temporaries as input.

A temporary can be defined by adding a ϕ-function to the top of the basic block. Using a ϕ-function to resolve a temporary has multiple effects. A ϕ resolution closes the dependence graph locally, but expands it in the predecessor basic blocks. A temporary of the correct type is taken from each predecessor block to be the use temporary in the ϕ-function. In order to prevent explosive growth of the dependence graph, it is preferable to select a pre-existing temporary in the predecessor rather than creating a new one (which would require further resolution). In case a ϕ resolution is performed in the entry block (which has no predecessors to source a temporary from), an empty ϕ-function is added. These empty functions are used in the clean up stage as arguments to the procedure.

Temporaries can also be defined by replacing them with an existing temporary. An example is shown in figure 3.3; the two operators in iteration 2 end up sharing a temporary in iteration 3. The temporary being replaced is the *replacement target* and the temporary being used for replacement is the *replacement candidate*. The candidate can either be a temporary which has yet to be defined or one which already has been. In the latter case, care must be taken to avoid selecting candidates whose defining instruction comes *after* any uses of the replacement target. If this condition is not upheld, the candidate will (after replacing the target) have a use before def and the SSA form of the procedure will be broken.

The final type of instruction that can be added to a basic block are memory stores. However, stores cannot be used to resolve temporaries, as they generally do not have any defining operands. In order to ensure that the generated code contains memory stores, we insert these after every temporary resolution iteration. If a basic block has new, unresolved temporaries after all outstanding temporaries in an iteration have been resolved, there is a small random chance of adding a memory store to the block. This ensures that the blocks are seeded with memory stores.

Clean up

Once the instruction generation is finished and all basic blocks are filled with instructions, a basic cleanup is performed to remove redundant instructions. The existence of the cleanup stage is justified, as generating the data flow without considering the cleanup is simpler due to fewer assumptions and rules regarding the format of the code.

For our SSA form, we state that ϕ-functions are only required for basic blocks which have more than one predecessor. In the previous stage, ϕ-functions may have been added to blocks with only one predecessor. These ϕ-functions are redundant, as there is no reason to disambiguate between a single temporary. We remove these ϕ-functions and replace the temporaries that they define with the single source temporary.

As mentioned in Section 3.1.3, when resolving a temporary with a ϕ-function in the entry basic block, an empty ϕ-function is added. These ϕ-functions are meaningless in the entry block, so we replace them with instructions and register uses that match the calling convention of the target architecture. This lets us create functions that take input parameters.

Chapter 4

Implementation

This chapter discusses an implementation of the random code generation methods described in Chapter 3. The compiler framework LLVM (and its intermediate languages) is chosen as the code generator to work against. Although the methods are general and target-independent, we have chosen Hexagon, a general purpose digital signal processor (DSP) from Qualcomm [36] as the generation and testing target.

Section 4.1 describes the compiler, LLVM, chosen for the evaluation implementation as well as the relevant internal representations of LLVM. Section 4.3 describes the textual format of LLVM's Machine IR developed for the implementation. Section 4.4 describes the implementation of the random code generation algorithms and the parameters used for generation.

4.1 LLVM

LLVM is an optimizing compiler with industrial-strength code generation capabilities. LLVM's versatility stems from its modular code processing pipeline. Different analyses, optimizations and transformations in LLVM are encapsulated in *passes*. Each pass performs some contained task, such as global value numbering or dominator tree analysis. These passes are strung together to produce the desired final result [37].

LLVM's intermediate representation, LLVM IR (simply IR from here on), is a RISC-like instruction set built to model a general computer architecture at a high level of abstraction. The IR uses a set of high-level instructions to operate on data stored in an infinite set of temporaries[1]. As the IR has both a

[1]Technically, the "temporaries" in LLVM IR are named instructions which form a data dependency graph. However, for all intents and purposes they can be thought of as temporaries.

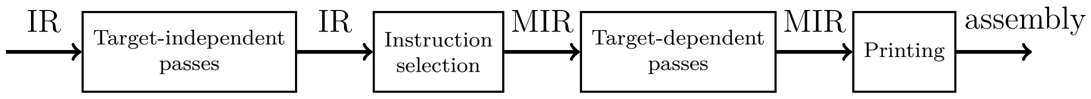

Figure 4.1. A simple flowchart describing the flow of IR and Machine IR through LLVM. Some steps and representations have been omitted for simplicity. The target-independent passes include optimizations such as linear invariant code motion, strength reduction and dead code removal. The target-dependent passes include register allocation, instruction scheduling and target-specific optimizations.

textual and binary representation, it can be freely generated and manipulated by any external program and fed into and out of LLVM [38].

As the IR is mostly target-independent, it must be converted into a format that is closer to the assembly language of the compilation target during code generation. After instruction selection, the IR is lowered into a more target-dependent (but still largely general) representation, called Machine IR. Figure 4.1 shows the structure of the LLVM pipeline and which representations are used between which stages.

Machine IR is closer in appearance to machine code than the IR. Each Machine IR instruction either belongs to the instruction set of the compilation target or is a pseudo-instruction which must be resolved before code emission. Machine IR can be in both SSA and non-SSA form, unlike the IR. Instruction operands can be immediates, basic blocks and memory locations, as well as both physical and virtual registers. This representation is wholly internal (unlike the IR) and has no textual representation outside of LLVM, other than in the form of debugging printouts, which cannot be read back into LLVM.

4.2 Hexagon

Hexagon is a general purpose very long instruction word (VLIW) digital signal processor architecture from Qualcomm. Its primary application is as a supporting DSP in Qualcomm's Snapdragon system-on-chip, located in numerous smartphones and tablets. As a VLIW architecture, Hexagon instructions are collected into groups called *bundles* during compilation. Each bundle can contain up to four instructions, and the instructions in a bundle are executed in parallel. The architecture possesses a 32-bit and 64-bit register set, as well as a set of 1-bit predicate registers for conditional jumps and predicated instructions [36].

Hexagon was chosen as the target for the experimental evaluation for several reasons. It is a well established embedded architecture, having shipped 1.2 billion Hexagon cores in 2012 [39]. As Hexagon's parallelism depends virtually entirely on compile-time optimization, correctness of the code generation is especially important for Hexagon. The architecture possesses features that exercise many different parts of the compiler (predication, instruction bundling) which lets us maximize the coverage of our testing.

4.3 MLL

To make testing of the post-instruction selection stages viable, a textual format of Machine IR must be constructed. This format must be readable and writable from LLVM at any stage of code generation in order to generate file deltas, write contained tests and emit randomly generated test cases. As of this writing, LLVM has no such facility and only supports IR for this purpose.

We have constructed a textual format of Machine IR called MLL[2] which encapsulates the functionality needed to serialize Machine IR. An example of MLL is shown in figure 4.2. This format can be emitted at any point after instruction selection and then read back into LLVM to resume compilation.

Functions are preceded by a declaration containing the name of the function, along with a set of function-specific properties such as stack frame objects and register classes of temporaries. The properties are in the form of tuples, with the left item denoting the property name and the right item being the property contents. This declaration is followed by series of basic block labels and instructions.

Each instruction is defined by three parts: an instruction name as given in LLVM's target description, a series of operand tuples, and an optional series of instruction-specific properties similar to the function properties. The operand tuples consist of an operand type (such as register, temporary, immediate, etc.) and its associated operand value.

4.4 Random code generator

The implementation of the random code generator has several parameters which affect the generated code.

The flow graph generation algorithm has a parameter to control the size of the graph. The algorithm performs a certain number of iterations consisting of transforms $T1'$ and $T2'$. The number of iterations is dependent on a parameter provided upon execution of the generator, called *graph scale*, *GS* for short.

[2]The file extension of LLVM IR files is `.ll`, so the Machine IR format becomes `.mll`.

```
function: fac ({triple, hexagon-unknown-linux-gnu},
{regclasses, 0-0|1-0|2-0|3-0|4-0|5-0|6-0|7-2|8-2|})
b0:
  ENTRY {reg, R0}
  COPY {temp, 5}, {reg, R0}
  TFRI {temp, 6}, {imm, 1}
  CMPEQri {temp, 7}, {temp, 5}, {imm, 0}
  JMP_t {temp, 7}, {mbb, 2}
  JMP {mbb, 1}
b1:
  PHI {temp, 0}, {temp, 6}, {mbb, 1}, {temp, 2}, {mbb, 2}
  PHI {temp, 1}, {temp, 5}, {mbb, 1}, {temp, 3}, {mbb, 2}
  ADD_ri {temp, 3}, {temp, 1}, {imm, -1}
  MPYI {temp, 2}, {temp, 0}, {temp, 1}
  CMPEQri {temp, 8}, {temp, 3}, {imm, 0}
  JMP_f {temp, 8}, {mbb, 1}
  JMP {mbb, 2}
b2:
  PHI {temp, 4}, {temp, 6}, {mbb, 0}, {temp, 2}, {mbb, 2}
  COPY {reg, R0}, {temp, 4}
  JMPret {reg, R31}
  RETURN {reg, R0}
```

Figure 4.2. An MLL output example for the factorial function. The instruction set and compilation target is Hexagon.

Given the graph scale parameter, the number of iterations will be a random number uniformly distributed in the interval $[GS/3, GS]$.

Figure 4.3 shows the probability parameters for the T1$'$ and T2$'$ transforms. As shown in the table, there is a 15% and 85% chance that a T1$'$ and T2$'$ transform will be performed, respectively. These probabilities stem from the observation that approximately 10% of intraprocedural branches are back edges. This number is supported by scanning the source code of LLVM for the words `if`, `case`, `while` and `for`. The scan shows that 88% of these occurrences are `if` and `case`, and the remaining 12% are `while` and `for`. This makes 85-15 a reasonable assumption.

The probabilities for the variants of T2$'$ are also given. These are relative to the base probability of 85%. Given that T2$'$ has been selected and the chosen transform node has one successor, there is then a 40%, 50% and 10% chance for T2$'a$, b and c, respectively. The low probability for c is to prevent

			0 succ.	1 succ.	2 succ.
T1′	15%		–	100%	–
		a)	100%	40%	33%
T2′	85%	*b*)	–	50%	33%
		c)	–	10%	34%

Figure 4.3. The probabilities for control flow graph generation. The T1′ and T2′ probabilities are absolute; for example, in an iteration there is an 85% chance that T2′ will be performed. The subsequent probabilities are relative; for example, given that T2′ was chosen and the node chosen for transformation has one successor, there is a 40% chance that T2′a will be performed.

the graph from becoming too tree-like and limit the number of exit nodes to a reasonable size.

Pseudocode for the graph generation algorithm is given in algorithm 1. As T1′ can only be performed on nodes with one successor and no self-loop, T2′ is performed instead if there are no nodes that fulfill this criteria. All nodes in a graph are candidates for T2′. The probability for selecting a candidate node is uniform; all nodes (except those which are not valid candidates) have the same probability of being selected.

Algorithm 1 CFG generation algorithm

```
procedure GenerateCFG
    $Iter \leftarrow \text{uniform}(GS/3, GS)$
    for every $Iter$ do
        $Prob \leftarrow \text{uniform}(0, 1)$
        if $Prob \in [0, 0.85)$ then
            perform T2′
        else
            $Success \leftarrow$ perform T1′
            if ! $Success$ then
                perform T2′
            end if
        end if
    end for
end procedure
```

For the data flow generation algorithm, we must define the different probabilities for resolving temporaries in basic blocks. We use a sliding scale dependent on an externally defined parameter called *basic block scale* (or BBS).

When combined with the number of instructions in a basic block, this scale is used to randomly pick a method for resolution. The probability of adding a memory store to a basic block after resolving its temporaries is fixed at 10%. Algorithm 2 shows the primary loop of the data flow generation algorithm.

Algorithm 2 Data flow generation algorithm

```
1: procedure GenerateData
2:     while !$Finished$ do
3:         $Finished \leftarrow True$
4:         for all $BB \in BasicBlocks$ do
5:             $Temps \leftarrow$ undefined temporaries in $BB$
6:             for all $T \in Temps$ do
7:                 $Finished \leftarrow False$
8:                 $Method \leftarrow$ select random method
9:                 resolve $T$ with $Method$
10:            end for
11:            $Prob \leftarrow$ uniform$(0, 1)$
12:            if $Temps$ is not empty & $Prob \in [0, 0.1)$ then
13:                add memory store
14:            end if
15:        end for
16:    end while
17: end procedure
```

Figure 4.4 shows the probability scale used in the implementation. P_{start} is the low end of the scale and P_{end} is the high end. The actual probability for each method is given by linearly interpolating between P_{start} and P_{end}. Given the number of instructions in a basic block n, the basic block scale parameter BBS and the P-values, the following formula is used to calculate the actual probability P:

$$P = P_{start} + (P_{end} - P_{start}) \cdot \frac{\min(n, BBS)}{BBS}$$

As an example, presume that we are resolving a temporary in a basic block which has six instructions, and our basic block scale is set to 10. This provides us with $BBS = 10, n = 6$. The scaling factor will then be $\frac{6}{10} = 0.6$. The probability of resolving this temporary with an operator is $0.65 + (0.25 - 0.65) \cdot 0.6 = 0.41$. The scaling factor saturates when $n = BBS$, so for a basic block with 10 instructions or more, the probability would be 0.25.

Method	P_{start}	P_{end}
Operator	65%	25%
Memory read	5%	5%
Immediate move	5%	15%
ϕ-function	5%	20%
Existing temporary	0%	25%
Postpone	20%	10%

Figure 4.4. The minimum and maximum probabilities for determining which method to use for resolving a temporary. The actual probability slides linearly along the scale depending on how many instructions are in the basic block.

Chapter 5

Evaluation

This chapter describes the experimental evaluation of the code generation and testing methods detailed in Chapters 3 and 4.

5.1 Experimental setup

This section contains a description of the testing method and a summary of the results. The following sections 5.2, 5.3 and 5.4 discuss the individual failures and their causes.

The testing was performed by generating MLL code functions of varying size. These sizes (small, medium and large) were determined by varying the parameter values for graph and basic block scale, as shown in figure 5.1. 4000 test files were generated for each size category.

This test data was then fed into a modified version of llc, LLVM's IR compilation tool. The program was modified to accept MLL files and initiate compilation directly after the instruction selection stage. Compilation of the MLL files was performed at three optimization levels: O0 (no optimiza-

	small	medium	large
graph scale	15	40	70
basic block scale	5	5	10
avg. number of blocks	6.92	18.1	31.1
avg. number of instructions	153	416	904
avg. number of loops	0.984	3.22	5.82

Figure 5.1. The parameters used to generate the three size categories and basic statistics of the generated functions.

tion), O1 (low optimization) and O3 (maximum optimization). Pairing the optimization level groups with the generated code size groups gives nine individual pairs. These pairs are denoted by the names s0 for small O0, s1 for small O1, etc. The implementation was made in LLVM 3.5, compiled with g++ 4.8.4 and executed on an Intel Xeon E5607 2.27GHz running Gentoo 2.2.

The test results were only verified with the assertions and verifiers that LLVM provides. It is difficult to construct an oracle to verify the produced output for the given input, as this would either require multiple compiler-independent implementations of the design, or a static code analyzer that can determine if the produced output is equivalent to the input. Furthermore, LLVM's assertions were already present in the code and could be used with no extra work involved.

In Section 2.4, we mentioned the random C code generator Csmith. The advantage Csmith has over the testing approach in this report is that it has no need for an external oracle; the oracle is implicitly given by the different compiler implementations. As the method in this report aims to generate code on a lower level of abstraction, the generated code format (instruction representation, register models, etc.) is compiler-specific and will not be compatible across compilers. Static code verification is a complex topic and is beyond the scope of this thesis. Therefore, a simpler assertion-based verification method is employed instead.

	s0	s1	s3	m0	m1	m3	l0	l1	l3
Success	100%	94.9%	94.9%	100%	81.9%	81.3%	100%	72.3%	72.4%
Failure 1	0%	5.1%	5.05%	0%	18.8%	18.7%	0%	27.6%	27.5%
Failure 2	0%	0%	0%	0%	.025%	.025%	0%	.1%	.1%
Failure 3	0%	.025%	.025%	0%	0%	0%	0%	0%	0%

MTBF	s0	s1	s3	m0	m1	m3	l0	l1	l3
Failure 1	-	18.57	18.76	-	4.314	4.335	-	2.618	2.638
Failure 2	-	-	-	-	3708	3708	-	955.3	955.3
Failure 3	-	2365	2365	-	-	-	-	-	-

Figure 5.2. A summary of the test results. The top table shows the number of successful and failed test cases in the individual categories, and the number of each failure type. The bottom table shows the mean time between failures (failure rate) of the failure types.

In testing, three distinct failure types were encountered. All types were the result of assertion failure. Figure 5.2 details the success and failure rates of the test groups. The figure also shows the mean time between failures (or MTBF) of the individual failure types. This is the average number of successful compilations before a failure of that type occurred.

5.2 Failure 1

Failure 1 was the most common of the three observed failures. It manifested as the assertion failure in figure 5.3.

```
llvm/lib/CodeGen/DFAPacketizer.cpp:84:
void llvm::DFAPacketizer:: reserveResources(const
llvm::MCInstrDesc*): Assertion ‘CachedTable.count(StateTrans) != 0’
failed.
```

Figure 5.3. The assertion failure for failure 1.

The assertion is located in the target-independent portion of the instruction packetizer. The packetizer is a part of the instruction scheduling stage, and is used (in Hexagon's case) to combine instructions into what LLVM calls *bundles*. Bundles are groups of instructions that are to be executed in parallel. As Hexagon is a VLIW architecture, the compiler is responsible for scheduling and bundling instructions to achieve the most efficient execution.

The Hexagon architecture is capable of executing four instructions in parallel. For this purpose, it has four instruction slots, numbered 0 to 3. Different instructions can be executed in all, some or only one of these slots. In LLVM, the slot requirements are encoded in the form of available resources, which instructions must reserve in order to be bundled together. If an instruction is to be added to a bundle but its resource has already been taken, it is not a viable candidate for bundling.

LLVM's packetizer uses a state machine to determine if instructions can be bundled with each other. Each state is a possible combination of reserved resources and each transition is a reservation of a certain resource. The resource requirements (called *itineraries*) are encoded into the Hexagon instruction definitions in LLVM. Figure 5.4 shows a toy example with two resources: $S0$ and $S1$; and three types of instructions: those which must use $S0$, those which must use $S1$ and those which can use either. The example demonstrates resource reservation; there is no transition from state b on $S0$, as that resource has already been taken by a previously bundled instruction.

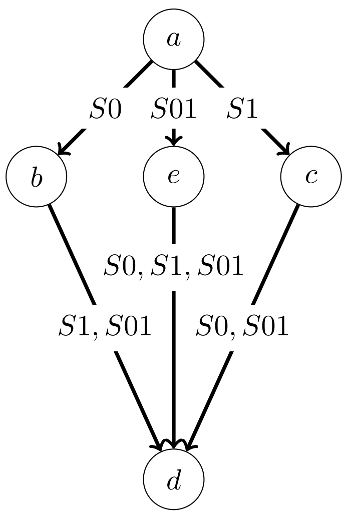

Figure 5.4. A toy example of a packetizer state machine with two resources and three instruction types. a is the start state for new bundles. As instructions are added to the bundle, the state machine moves along the transitions according to what resources the instructions require. If there is no available transition, the instruction cannot be added to the bundle and the bundle is closed.

5.2.1 Cause

The common element of all test cases that fail on this assertion is that they all contain a store (ST) instruction followed by a Hexagon new-value jump (NV/J) instruction. A truncated MLL example is given in figure 5.5. The Hexagon V4 Programmer's Reference Manual states "ST-class instructions can be placed in Slot 1. However, in this case Slot 0 must contain a second ST-class instruction." [36] This is called a dual-store. The manual also states that new-value jump instructions must be executed in slot 0. In the failing test cases there is only one store instruction followed by a new-value jump, so both instructions must be executed in slot 0. This is clearly impossible, so the new-value jump should be placed in a separate bundle.

```
[...]
STriw_pred {fi, 0}, {imm, 0}, {temp, 9}
CMPEQn1_t_Jumpnv_nt_V4 {temp, 8}, {mbb, 2}
JMP {mbb, 3}
```

Figure 5.5. An example of a store instruction (`STriw_pred`) before a new-value jump instruction (`CMPEQn1_t_Jumpnv_nt_V4`).

However, in the Hexagon-specific portion of the packetizer, an attempt is made to bundle the store with the new-value jump despite this not being a valid configuration. The assertion is triggered when the packetizer attempts to reserve the required resource for the new-value jump, but the packetizer state machine lacks the necessary transition and fails.

In the source code of the packetizer, an assumption is made that the failing call to `reserveResources` will be successful. The relevant excerpt is shown in figure 5.6.

```
// Here, we are sure that "reserveResources" would succeed.
ResourceTracker->reserveResources(nvjMI);
```

Figure 5.6. A source code excerpt from `HexagonVLIWPacketizer.cpp`.

The packetizer does perform a test in `isLegalToPacketizeTogether` to determine if the instruction being packetized (in this case, the `STriw_pred`) can be ”glued” with an upcoming new-value jump. The procedure includes a test for store instructions to prevent a new-value jump from being glued to a store, but the test is never performed as the procedure erroneously considers the two instructions to be independent of each other.

5.3 Failure 2

The second observed failure was from the assertion shown in figure 5.7.

```
llvm/include/llvm/ADT/ilist.h:215:
llvm::ilist_iterator<NodeTy>& llvm::ilist_iterator<NodeTy>::
operator--() [with NodeTy = llvm::MachineInstr;
llvm::ilist_iterator<NodeTy> = llvm::ilist_iterator
<llvm::MachineInstr>]: Assertion ‘NodePtr && "--’d off the
beginning of an ilist!"’ failed.
```

Figure 5.7. The assertion failure for failure 2.

This assertion is less obvious as the previous one, as it is occurring in LLVM’s abstract list type implementation and could be the result of any invalid list operation. A debugger stack trace shows that the failure originates in Hexagon’s conditional transfer optimization pass. This pass attempts to convert a conditional multiplexer instruction into a new-value predicated transfer.

A conditional multiplexer (or just mux) instruction assigns one of two values to a destination register based on the value of a predicate register. A new-value predicated transfer is simply a regular copy (immediate or register-to-register) that has been predicated. Converting one instruction to the other saves one clock cycle. The example given in the source code is shown in figure 5.8.

```
  {p0 = cmp.eq(r0,r1)}
  {r3 = mux(p0,#1,#3)}
becomes
  {p0 = cmp.eq(r0,r1)
   if (p0.new) r3 = #1
   if (!p0.new) r3 = #3}
```

Figure 5.8. The example for conditional transfer optimization given in `HexagonSplitTFRCondSets.cpp`. The top code sample requires two bundles while the bottom sample only requires one. Note that the example solely uses immediates as the source parameters for the selection; this is not a strict requirement.

5.3.1 Cause

The common features of the test cases for failure 2 are twofold; they all contain a conditional mux (named *condset* in LLVM) at or near the top of a basic block, and both source operands are the same temporary. During register allocation, the mux is allocated the same register for all three register operands: destination and the two sources. The examples in figure 5.9 show the generated MLL and a debug printout of the same code after register allocation.

When performing the optimization to convert this instruction into a new-value predicated transfer, the pass makes a faulty assumption. It assumes that both source operands are not the same as the destination and that the mux is not at the top of the basic block. The source code excerpt in figure 5.10 shows the problem.

If both source and destination operands are the same register, no new instructions will be added. This is the correct optimization to perform. However, if the mux was located at the top of the basic block, the instruction iterator `MII` will underflow after the mux instruction `MI` is removed, causing the assertion failure.

```
b1:
  TFR_condset_rr {temp, 1}, {temp, 14}, {temp, 2}, {temp, 2}
  [...]
```

```
BB#1:
  Live Ins: %P1 %R0
  Predecessors according to CFG: BB#3
    %R0<def> = TFR_condset_rr %P1<kill>, %R0<kill>, %R0
    [...]
```

Figure 5.9. An example of a mux at the top of a basic block. The top block is the generated MLL and the bottom block is the Machine IR debug printout after register allocation. In the top block, temporary 1 is the destination, temporary 14 is the predicate and temporary 2 is the source.

```
// Minor optimization: do not emit the predicated copy if the source
// and the destination is the same register.
if (DestReg != SrcReg1) {
  [...]
}
if (DestReg != SrcReg2) {
  [...]
}
MII = MBB->erase(MI);
--MII;
```

Figure 5.10. The source code from `HexagonSplitTFRCondSets.cpp` responsible for the failure. The semantics for adding the new-value conditional transfer instructions are not relevant and have been omitted for simplicity.

5.4 Failure 3

The third failure was only observed once in all 12000 test cases. The assertion is given in figure 5.11.

```
llvm/include/llvm/Support/BranchProbability.h:34:
llvm::BranchProbability::BranchProbability(uint32_t, uint32_t):
Assertion ‘d > 0 && "Denomiator cannot be 0!"’ failed.
```

Figure 5.11. The assertion failure for failure 3. The typo of “denominator” is as is written in LLVM’s source code.

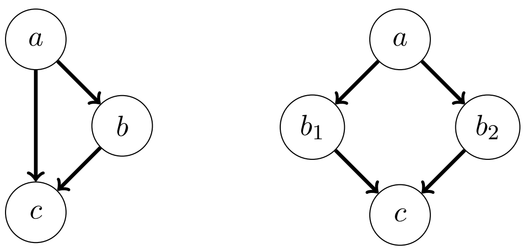

Figure 5.12. Examples of triangle and diamond shapes in control flow graphs. If the blocks named b only contain predicable instructions, those blocks can be merged into a and c and become one single basic block.

The assertion is located in the branch probability analysis pass, called by the Hexagon new-value jump optimization pass. However, the original cause of the failure lies in LLVM's if-conversion pass and Hexagon's instruction definition implementation.

If-conversion is an optimization to simplify control flow graphs by folding basic blocks consisting of predicable instructions into their predecessor. Predicable instructions are instructions which only perform their operation based on a computed condition or predicate register value [40]. In Hexagon, many instructions are predicable which makes the architecture a suitable target for this optimization.

Candidate subgraphs for if-conversion are shown in figure 5.12. If the control flow graph contains a diamond or triangle shape and the branching block or blocks consist solely of predicable instructions, these blocks can be eliminated by folding them into their predecessor, resulting in a single large block.

5.4.1 Cause

The cause of the failure lies in a badly performed if-conversion. Observe the machine code in figure 5.13. Block 2 forms a triangle with blocks 4 and 5. If all of the instructions in block 2 are predicable, all three blocks will be merged and the instructions in block 2 will be predicated based on the condition of the jump in block 4.

The branch instruction used in block 4 to jump to block 2 (`JMP_fnew_t`) is what Hexagon calls a *speculative jump*. These branch instructions are denoted by the suffix `new_t` or `new_f`; the jump is given a branch predictor hint during compilation to potentially speed up execution. The hint is either `t` or `nt` for *taken* and *not taken*, respectively.

The source code of `HexagonInstrInfo.cpp::RemoveBranch` (used by the branch folder to complete the if-conversion) shows that the only instructions

```
BB#2:
  Predecessors according to CFG: BB#4
    %R0<def> = LDriw %R29<kill>, 16
    %R0<def> = SXTH %R0<kill>
    STriw %R29<kill>, 16, %R0<kill>
    JMP <BB#5>, %PC<imp-def>
  Successors according to CFG: BB#5

BB#4:
  Live Ins: %R3
  Predecessors according to CFG: BB#0 BB#1 BB#3
    %R0<def> = ASRH %R3<kill>
    %R1<def> = OR_rr %R0, %R0
    %P0<def> = CMPEQrr %R1<kill>, %R0<kill>
    JMP_fnew_t %P0<kill>, <BB#2>, %PC<imp-def,dead>
  Successors according to CFG: BB#2 BB#5

BB#5:
  Predecessors according to CFG: BB#2 BB#4
    %R0<def> = LDriw %R29<kill>, 16
    DEALLOCFRAME %R29<imp-def>, %R30<imp-def>, [...]
    TCRETURNtg 928567614, %R0<imp-use,kill>
```

Figure 5.13. The post-register allocation dump of the failing test case. Parts of the dump have been omitted for clarity.

it is capable of removing when removing a branch are the simple branch instructions `JMP`, `JMP_t` and `JMP_f`. This means that after block 2 has been predicated and blocks 2 and 4 have been merged, the `JMP_fnew_t` instruction remains in the middle of the new block. This code is invalid for two reasons. First, the instruction retains a reference to the old block 2, a basic block that was removed in the merge. Second, the block contains a branch instruction that is not located at the bottom of the block. Figure 5.14 shows the resulting machine dump.

```
Merging into block: BB#4:
  Live Ins: %R3
  Predecessors according to CFG: BB#0 BB#1 BB#3
    %R0<def> = ASRH %R3<kill>
    %R1<def> = OR_rr %R0, %R0
    %P0<def> = CMPEQrr %R1<kill>, %R0<kill>
    JMP_fnew_t %P0, <BB#-1>, %PC<imp-def,dead>
    %R0<def> = LDriw_cNotPt %P0, %R29<kill>, 16, %R0<imp-use,undef>
    %R0<def> = SXTH_cNotPt_V4 %P0, %R0<kill>, %R0<imp-use,undef>
    STriw_cNotPt %P0, %R29<kill>, 16, %R0<kill>
  Successors according to CFG: BB#5
```

Figure 5.14. A machine dump of block 4 after the failed if-conversion. Note that the old branch instruction from block 4 is still in the middle of the block. The instruction also has a broken block reference (`<BB#-1>`), as block 2 was removed in the merge.

5.5 Analysis

The results of the testing summarized in figure 5.2 tells us that the most common failure is failure 1, followed by 2 and 3. No failures occurred at optimization level o0. This is due to the fact that none of the optimizations in which the failures originate are run at that level; new-value jump optimization, if-conversion and packetization are only run at o1 or higher. The chance of experiencing failures 1 and 2 goes up with the code size, but not for failure 3. This is because failure 3 was only triggered by a very specific combination of instructions, and having more instructions makes it harder to fulfill that requirement.

Although the three failures have widely varying causes, there are still a number of interesting similarities between them. They are all caused by generating code which violates some invariant of the Hexagon backend. For the first failure, the backend assumes that it will be able to reserve the resources for a new-value jump despite it being packeted with a store; for the second failure, the backend assumes that there will never be a conditional mux at the top of a basic block and for the third failure, the backend assumes that no pass will ever attempt to remove a branch after the simple branch instructions have been optimized out by some other pass.

A possible reason for why these failures have never occurred before in production code is due to the execution order of passes and the semantics of LLVM's instruction selector. It is possible that a store will never be selected before a jump, preventing failure 1 from appearing. Failure 2 will most likely never occur due to optimization during instruction selection; after all, there is no need to conditionally select between the same temporary, which makes the instruction in question a no-op. Failure 3 would never be considered a problem if the order of the passes prevents the observed behavior; for example, if the speculative jump optimization pass occurs strictly after all passes that modify the CFG.

From these observations, it is absolutely possible that these defects do not currently pose an issue for code generation. Nevertheless, these defects could still become apparent in the future if changes are made to instruction selection, selection patterns or other parts of the Hexagon backend in LLVM. We can also conclude that our methods are capable of uncovering defects which cannot be found with other testing methods, such as source code randomization of higher level languages like C.

Chapter 6

Conclusion

This chapter summarizes the work done during this project and presents the conclusions drawn from the experimental evaluation. Future work and improvements are discussed.

We proposed that random testing and generating random programs on a low level is a suitable method for testing the code generation stages of a modern compiler. We designed a method to generate random low level code functions by employing inverse reducibility transforms for control flow generation and a bottom-up temporary resolution algorithm for data flow generation.

We implemented this design in a modern compiler framework, LLVM, and developed a method to emit and inject a textual format of LLVM's post-instruction selection representation, Machine IR, out of and into LLVM. The embedded digital signal processor architecture Hexagon was chosen as the testing and generation target. A testing suite of 12000 randomly generated Machine IR files was assembled and split into three size categories. These files were subsequently compiled for three different optimization levels and any errors were logged and categorized.

Three different failure types were observed during testing. All three were triggered by assertion failures in LLVM and Hexagon's target-specific backend implementation. The three failures were primarily caused by unexpected instructions and instruction patterns that the Hexagon backend was not capable of handling or handled incorrectly. We believe that this is the first time these defects have been observed, as all of the involved code is very well exercised during compilation. For this reason, we conclude that these patterns will not occur in production code.

Although we conclude that the observed failures will not be observed during real compilation of C or IR code, the evaluation shows that it is possible to generate code patterns in a lower level representation that can cause compilation failure. We believe this is compelling evidence that the methods and testing approach described in this paper are an important step towards greater robustness of compilers and the code they produce.

6.1 Future work

There are many improvements and additions in the testing and generation methods that can be made to the work in this thesis. This section describes potential future improvements.

6.1.1 Extended data flow model

The current data flow generation algorithm only takes temporary definition order into account when generating instructions, as a result of the rules of SSA form. However, there are similar restrictions on use of memory. Accessing a portion of memory before it has been initialized is undefined behavior. As the algorithm does not attempt to initialize memory locations before usage, extending the SSA form with a "memory SSA" [41, 42] is a meaningful improvement.

Certain types of instruction are not considered for generation. This includes features such as calls, memory barriers, intrinsic operations, indirect jumps and probably many more. Extending the model to deal with these features would increase the testing coverage.

The data flow generation algorithm also has no sense of what the different values in temporaries mean. If the memory model were extended to give the algorithm a sense of which values are integers, pointers, etc., the robustness of the memory model would improve.

All of the improvements touched upon here could allow the code to be run on the target architecture, permitting testing of not only the compiler but also the architecture implementation itself.

6.1.2 Test minimization

When generating the random data sets for testing the compiler, much time is spent finding the common element in all the test cases. This is manual and tedious work, involving stepping through the failing execution path manually with a debugger to determine the cause of the failure.

Delta debugging is a technique in which test cases are iteratively split or minimized to produce the smallest possible failing example of the given case [22, 43]. Delta debugging is intended to be fully automated, allowing the tester to focus on solving the issue rather than finding its cause. Delta debugging is very applicable to this thesis, as the test data (textual low level IR code) follows a strict structure and should be easy to automatically minimize.

6.1.3 Generalized implementation

The current implementation described in Chapter 4 is very specific to Hexagon. It currently does not support any other architecture, despite the algorithms being largely general. This was due to difficulty in extracting certain portions of the architecture description from LLVM, such as calling and return conventions, as well as some operand types (immediate size/alignment/signedness).

Constructing an independent architecture description that fits the generation methods better would make it possible to create an implementation that can generate MLL files for any architecture, improving the coverage of the testing.

Bibliography

[1] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM international conference on Embedded software*, pp. 255–264, ACM, 2008.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 283–294, 2011.

[3] M. Lam, R. Sethi, J. Ullman, and A. Aho, *Compilers: Principles, Techniques and Tools, 2/e.* Pearson Education, 2006.

[4] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[5] I. Cutcutache and W.-F. Wong, "Fast, frequency-based, integrated register allocation and instruction scheduling," *Software: practice and experience*, vol. 38, no. 11, pp. 1105–1126, 2008.

[6] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 1994.

[7] F. Chow, "Intermediate representation," *Queue*, vol. 11, pp. 30:30–30:37, Oct. 2013.

[8] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of signal processing systems.* Springer Science & Business Media, 2010.

[9] G. S. H. Blindell, "Survey on instruction selection: An extensive and modern literature study," *KTH Royal Institute of Technology, Stockholm, Sweden, Tech. Rep*, 2013.

[10] D. R. Koes and S. C. Goldstein, "Near-optimal instruction selection on dags," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 45–54, ACM, 2008.

[11] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 98–101, 1982.

[12] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[13] 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 (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[14] J. Janssen and H. Corporaal, "Making graphs reducible with controlled node splitting," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 6, pp. 1031–1052, 1997.

[15] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," in *Proceedings of the fourth annual ACM symposium on Theory of computing*, pp. 238–250, ACM, 1972.

[16] E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 25–35, ACM, 1989.

[18] P. Briggs, K. D. Cooper, and L. T. Simpson, "Value numbering," *Software-Practice and Experience*, vol. 27, no. 6, pp. 701–724, 1997.

[19] D. Graham, E. v. Veenendaal, I. Evans, and R. Black, "Foundations of Software Testing: ISTQB Certification," 2006.

[20] D. Hoffman, "A taxonomy for test oracles," in *Quality Week*, vol. 98, pp. 52–60, 1998.

[21] Y. Cheon, "Abstraction in assertion-based test oracles," in *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pp. 410–414, IEEE, 2007.

[22] C. Artho, "Iterative delta debugging," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 3, pp. 223–246, 2011.

[23] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[24] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[25] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 870–879, 1990.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 75–84, IEEE, 2007.

[27] G. Fink and M. Bishop, "Property-based testing: a new approach to testing for assurance," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 74–80, 1997.

[28] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[29] T. Pavlidis, "Linear and context-free graph grammars," *Journal of the ACM (JACM)*, vol. 19, no. 1, pp. 11–22, 1972.

[30] "llvm-stress - generate random .ll files – LLVM 3.7 documentation." http://llvm.org/docs/CommandGuide/llvm-stress.html. Accessed: 2015-05-07.

[31] F. Sheridan, "Practical testing of a c99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007.

[32] R. Muth, "Register liveness analysis of executable code," *Manuscript, Dept. of Computer Science, The University of Arizona, Dec*, 1998.

[33] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2, pp. 53–62, 1998.

[34] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming.* Academic Press Ltd., 1972.

[35] R. C. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte, "Constraint-based register allocation and instruction scheduling," in *Principles and Practice of Constraint Programming*, pp. 750–766, Springer Berlin Heidelberg, 2012.

[36] Qualcomm, *Hexagon V4 Programmer's Reference Manual*, 2013. 80-N2040-9 Rev. A.

[37] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004.

[38] C. A. Lattner, *LLVM: An infrastructure for multi-stage optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[39] W. Strauss, "DSP Market Bulletin – 11/12/12: Qualcomm Leads in Global DSP Silicon Shipments," 2012.

[40] J. Z. Fang, "Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations," in *Languages and Compilers for Parallel Computing*, pp. 135–153, Springer, 1997.

[41] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in SSA form," in *Compiler Construction*, pp. 253–267, Springer, 1996.

[42] D. Novillo and R. H. Canada, "Memory SSA - a unified approach for sparsely representing memory operations," in *Proc of the GCC Developers' Summit*, Citeseer, 2007.

[43] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10, ACM, 2002.