Using Binary Decision Diagrams to Determine Program Equivalence in a Superoptimizer

J E S P E R   S Ä R N E S J Ö

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2011

Using Binary Decision Diagrams to Determine Program Equivalence in a Superoptimizer

J E S P E R   S Ä R N E S J Ö

Master's Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Torbjörn Granlund
Examiner was Johan Håstad

TRITA-CSC-E 2011:144
ISRN-KTH/CSC/E--11/144--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

A superoptimizer is a tool capable of generating an optimal program that computes a desired function. This thesis describes a superoptimizer that uses binary decision diagrams to determine if a generated candidate programs is equivalent to a given target program known to compute the desired function. The superoptimizer is shown to work well for generating program consisting of instructions no more complex than addition, but to perform poorly for programs containing multiplication instructions.

Referat

Att använda binära beslutsdiagram för att avgöra ekvivalens mellan program i en superoptimerare

En superoptimerare är ett verktyg som kan generera ett optimalt program som beräknar en önskad funktion. Den här rapporten beskriver en superoptimerare som använder binära beslutsdiagram för att avgöra om ett genererat kandidatprogram är ekvivalent med ett givet målprogram, som beräknar den önskade funktionen. Det visas att superoptimeraren fungerar väl för att generera program bestående av instruktioner som inte är mer komplexa än addition, men att den inte fungerar väl för program innehållande multiplikationsinstruktioner.

Contents

Chapter 1

Introduction

In the context of compiler technology, an *optimizer* is a tool that attempts to improve the resource usage of programs given to it. Typically, an optimizer takes a program and transforms it using heuristics designed, for instance, to reduce execution time or memory usage. Despite its name, however, an optimizer rarely makes a program *optimal*.

A *superoptimizer* takes a different approach. Given a *target program* that computes a desired function, it attempts to find, typically using some type of exhaustive search, the fastest or shortest program that computes the same function. In this way, a superoptimizer can generate programs that are indeed optimal.

Due to the inefficient nature of exhaustive search, a superoptimizer can only generate very short programs, typically no longer than 5 or 6 instructions, in reasonable time. Further, a superoptimizer can only generate *straight-line* programs, that is, programs that contain no loops, branches or other jumps. Despite this, superoptimizers can be useful for optimizing inner loops and other short sequences of code required to be fast or small.

A central problem in the field of superoptimization, is that of determining whether a generated program computes the same function as the target program, in the sense that they always yield identical output given identical and valid input. Programs that compute the same function are referred to as *equivalent*.

Methods used to determine program equivalence in existing superoptimizers include probabilistic testing, as well as expressing programs in Boolean logic and comparing canonical representations of them. A difficulty met by the latter method, is that the Boolean-logic representations of many common programs are very large.

This thesis explores using *binary decision diagrams* (BDDs), which are data structures capable of efficiently representing Boolean functions, for the purpose of determining program equivalence. Chapter 2 provides a primer on the history and functionality of superoptimizers and BDDs. Chapter 3 describes the subset of the x86 architecture which serves as the target architecture for the superoptimizer implemented as part of this thesis project, while Chapter 4 describes the superoptimizer itself, focusing on its method of using BDDs to determine program equivalence. Chapter 5 gives an example of the operation of the superoptimizer. It also presents an analysis of the performance of the superoptimizer's method of creating and comparing BDD representations of programs. Finally, Chapter 6 contains a few suggestions for improving on the results presented in this thesis.

Chapter 2

Background

2.1 Superoptimizers

2.1.1 Massalin's superoptimizer

The term *superoptimizer* was coined by Massalin in a 1987 paper [1], to describe a tool for finding the shortest straight-line program that computes a given function.

Massalin's implementation takes as its input a target program written in the assembly language of the Motorola 68020 processor. It then consults a table containing a subset of the processor's instruction set, and begins generating all combinations of these instructions, beginning with those of length 1, then 2, and so on. Each generated program is tested for equivalence with the target program. When a program passes the test, it is printed, and the search terminates. Because programs are tested in order of increasing length, this program will necessarily be *optimal* in terms of length, meaning that no shorter equivalent program exists.

Naturally, exhaustively searching the space of all possible programs is a very inefficient process, considering that the number of programs of length n is b^n, where b is the *branching factor* of the search tree. To find a value of b, consider an architecture with r registers, and an instruction set consisting of i_0 instructions that take no arguments, i_1 instructions that take one argument, and i_2 instructions that take two arguments. For this architecture, the branching factor would be, in the worst case, $i_0+i_1r+i_2r^2$. Even for modest values of i_0, i_1, i_2 and r, the branching factor would end up in the hundreds or thousands. Hence, a naive superoptimizer is only capable of generating very short programs in reasonable time.

To address this, Massalin describes a method of *pruning* the search tree. Using this method, the superoptimizer keeps a lookup table, where short instruction sequences known to be non-optimal are marked as such. When a program is found to contain a non-optimal instruction sequence, it is rejected without being tested.

Massalin also describes two methods of determining whether a candidate program is equivalent to the target program.

The first method, referred to as the *Boolean test*, is to express both programs in Boolean logic, reduce them to an unspecified canonical form, and compare them minterm for minterm. Using this method, Massalin claims the superoptimizer is capable of testing 40 programs of an unspecified length per second, when running on a 16 MHz processor. Massalin also notes, however, that this method could not be used to test all generated programs in reasonable time, especially drawing attention to the large Boolean-logic representations of addition and multiplication instructions.

The second method, referred to as the *probabilistic test*, is to simply execute both programs with identical inputs, and compare their outputs. This method introduces the risk of false positives, in the form of programs that are equivalent only on the input used, not in the general case, making manual verification of the output necessary. Further, whereas the Boolean test can be used on any kind of program, Massalin's probabilistic test can only be used on programs written in the assembly language of the superoptimizer's host architecture, because it actually executes the candidate program directly on the host architecture. This method is significantly faster, however, allowing the superoptimizer to test 50000 programs per second.

It is worth noting, that even with such a significant increase in speed, only slightly longer programs may be generated in reasonable time with a realistic branching factor, as illustrated by Figure 2.1. Note that this is not a characteristic unique to Massalin's superoptimizer, but an intrinsic property of the exponentially large space of possible programs.

2.1.2 GSO

GSO, the GNU superoptimizer, is described in a 1992 paper by Granlund and Kenner [2].

Like Massalin's superoptimizer, GSO optimizes for program length. It uses a probabilistic equivalence test, and as such its output must be manually verified. Unlike the probabilistic test in Massalin's superoptimizer, which executes instructions directly on the host architecture, GSO's test *simulates* them using functions that operate on a virtual machine of sorts, consisting of a set of registers and a single carry bit. GSO is therefore capable of supporting a wide range of architectures, but also requires that the target program be specified as a compiled-in *goal function*.

Granlund and Kenner also describe numerous methods used by GSO to prune the search tree. When selecting arguments for generated instructions, GSO only considers live registers, that is, registers that contain input values or have previously been written to. Similarly, instructions that read the carry flag are only generated after it has been set. For commutative instructions, that is, instructions for which arguments can be ordered in different ways without changing the meaning of the instruction, only one argument ordering is tried.

When generating the last instruction in a program, GSO is even more restrictive. Specifically, it requires that the last instruction reads a register or the carry flag written to by the preceding instruction, since that instruction would otherwise be superfluous.

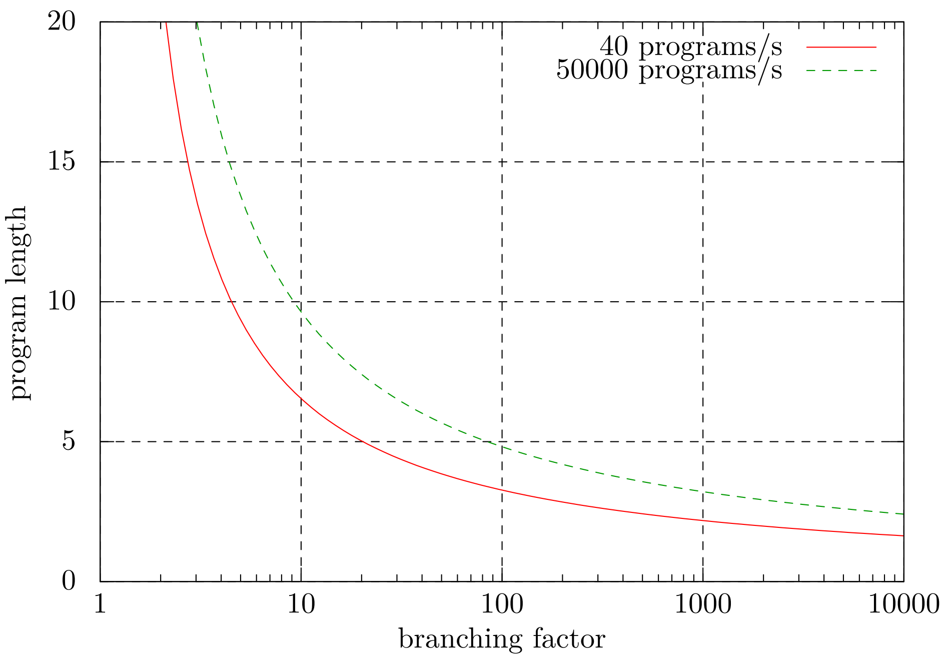

Figure 2.1. The maximum length of a program generated in 24 hours, as a function of the branching factor. The solid and dashed lines represent Massalin's superoptimizer using the Boolean and probabilistic tests, respectively. Note that the branching factor is displayed on a logarithmic scale.

2.1.3 Denali

Denali is a superoptimizer created by Joshi et al., then at Compaq Systems Research Center, first described in a 2002 paper [3].

Denali's design differs greatly from that of Massalin's superoptimizer and GSO. Rather than generating programs and then testing them, Denali starts by generating a set of programs equivalent to the target program, and then selects the optimal among them.

To accomplish the first step, Denali requires a set of *axioms*, which describe how instructions may be substituted for another without changing the meaning of the program. Such axioms may, for example, state that multiplication by 2^n is the same as right-shift by n. Axioms are also used to mark instructions as commutative or associative, and to specify their identity, for instance 0 for addition and 1 for multiplication. Denali applies these axioms to the target program, creating a set of programs equivalent to it.

The authors claim that Denali then selects, from this set, the program optimal in terms of *execution time*, that is, the number of cycles required for the program to terminate. This is remarkable, as determining the execution time for a program is complicated, with the number of cycles required to execute an instruction depending not only on the instruction itself, but also possibly on the instructions surrounding it, due to pipelining and other features of modern processors. In a follow-up paper [4], the authors describe a simplified design, named Denali-2, which instead optimizes for program length.

2.1.4 Bansal's superoptimizer

The superoptimizer created by Bansal and described in his 2008 PhD thesis [5] differs from the ones described above, in that it requires no human supervision, neither for verifying its output, nor for selecting its input. Instead, it reads binaries compiled for the x86 architecture, harvesting instruction sequences to optimize. Its output is not an optimized program, but rather the optimizations discovered. In this way, Bansal's superoptimizer is capable of autonomously generating a very large library of optimizations, which can then be used by an ordinary peephole optimizer.

Candidate optimizations are tested for correctness using a probabilistic test similar to that found in Massalin's superoptimizer and GSO. To eliminate the possibility of false positives, candidate optimizations which pass the probabilistic test are verified by being expressed in Boolean logic and made part of a satisfiability problem, using an unspecified method. The satisfiability problem is then handed off to an external SAT solver. No details of the performance of the Boolean verification process are given, although it is claimed that the superoptimizer can generate and verify programs containing 3 instructions in reasonable time.

Other notable features of Bansal's superoptimizer is that it is capable of optimizing both for execution time and for program size, and that it utilizes a meet-in-the-middle method of pruning the search tree. Using this method, the superoptimizer searches not only *forwards* for progressively longer programs, but also *backwards* from the state the architecture is required to be in following correct program execution.

2.1.5 TOAST

TOAST, short for *total optimisation using answer set technology*, is a superoptimizer created by Crick and described in his 2009 PhD thesis [6]. It is implemented using Answer Set Programming, a declarative logic programming language. Like Denali and Bansal's superoptimizer, it produces provably correct output.

2.2 Binary decision diagrams

A *binary decision diagram* (BDD) is a data structure used to represent Boolean functions. It is a directed acyclic graphs, with one or more nodes for each of the function's variables. Each node has one or more incoming paths, and exactly two outgoing paths, commonly referred to as the *high* and *low* path. When using a BDD to determine the value of a Boolean function, one follows a node's high path if the corresponding variable is 1, and its low path if the corresponding variable is 0, starting at the root node, until one arrives at a terminal node.

BDDs can be used to store and manipulate some Boolean functions very efficiently. The performance of the operations performed on a BDD, however, varies greatly depending on the function represented, and the order of its variables.

BDDs were introduced by Lee [7], under the name *binary-decision programs*. They were given their current name by Akers [8], who also explored the ideas of *reducing* BDDs by removing redundant nodes, and of representing multiple functions in a single BDD.

The concept of reducing BDDs was formalized by Bryant [9], who described an efficient algorithm that leaves a BDD containing no node with the same high and low path, and no two distinct nodes with isomorphic subgraphs, while still representing the same Boolean function. Bryant observed that a BDD that is both *reduced* and *ordered*, meaning that its variables appear in the same order on all paths from the root to a terminal node, is a *canonical* representation of its Boolean function. In other words, two Boolean functions are equivalent iff their reduced and ordered BDD representations are identical.

Bryant also described a set of algorithms for combining BDDs using Boolean operators to create BDD representations of more complex Boolean functions. These algorithms are efficient, requiring at worst a number of time steps proportional to the product of the number of nodes in the two BDDs.

Bryant further explored the idea of representing multiple functions in a single BDD, with common subexpressions reduced. This representation would later be referred to as a *shared* BDD by Minato et al. [10], who also noted that two equivalent Boolean functions represented in the same shared BDD, would have the same root node, making an equivalence test trivial.

Bryant further showed that the variable ordering used can have a dramatic effect on the number of nodes in a reduced BDD, which in the best case is linear in the number of variables, and in the worst case exponential. Bryant states that finding the variable ordering that minimizes the number of nodes is an NP-complete problem. Bollig and Wegener [11] would later show that determining whether an ordering exists for which the number of nodes is at most a given value, is an NP-complete problem.

Bryant asserts that for many functions, a good variable ordering can be determined by examining the structure of the function. As an example, he mentions that for functions in which variables primarily interact pairwise, ordering variables and their partners consecutively gives good results.

Bryant also states, however, that there are functions that can only be represented using a number of nodes exponential in the number of variables, regardless of variable ordering, and proves that integer multiplication is such a function.

2.2.1 Example

To illustrate how a moderately complex Boolean function can be represented by a BDD, consider an adder that takes two n-bit numbers, a and b, and produces their sum, as well a carry-out bit.

To express the adder in Boolean logic, we represent a as n Boolean variables $\langle a_0, a_1, ..., a_{n-1}\rangle$, where a_0 and a_{n-1} are the least and most significant bits of a, respectively. Doing the same for b, we get a total of $2n$ Boolean variables.

The carry-out bit, c_{n-1}, depends on every variable, and can be defined recursively as follows:

$$c_k = \begin{cases} a_k \wedge b_k & k = 0 \\ (a_k \wedge b_k) \vee (a_k \wedge c_{k-1}) \vee (b_k \wedge c_{k-1}) & 0 < k < n \end{cases}$$

For $n = 2$, the function c_1 determines the carry-out bit of the adder. The truth table of c_1 is shown in Table 2.1, while Figure 2.2 shows c_1 as represented by an ordered BDD, using the variable ordering $a_0, b_0, ..., a_{n-1}, b_{n-1}$. Note that the row of terminal nodes in the BDD correspond to the column for c_1 in the truth table. Figure 2.2 also shows a reduced version of the same BDD. Note that this representation is quite compact, and that not all variables are present on every path. For instance, if a_0 is 0, there is no need to check b_0, and we instead skip ahead to a_1.

Not all variable orderings allow such compact BDD representations of this function, however. If we instead use the variable ordering $a_0, ..., a_{n-1}, b_0, ..., b_{n-1}$, we end up with a larger BDD. This becomes more clear for larger values of n, as illustrated in Figure 2.3.

Upon inspection, we find that in the reduced BDD representation of this particular function, using these two variable orderings, the number of nodes is:

$$|c_{n-1}| = \begin{cases} 3n - 1 & \text{using variable order } a_0, b_0, ..., a_{n-1}, b_{n-1} \\ 2^{n+1} - 2 & \text{using variable order } a_0, ..., a_{n-1}, b_0, ..., b_{n-1} \end{cases}$$

For $n = 32$, the reduced BDD would consequently contain only 97 nodes with the first variable ordering, but over 8 billion with the second.

Note that Bryant's suggested method of ordering interacting variables consecutively, works very well in this case.

a_0	b_0	a_1	b_1	c_0	c_1
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Table 2.1. Truth table for the Boolean function c_1.

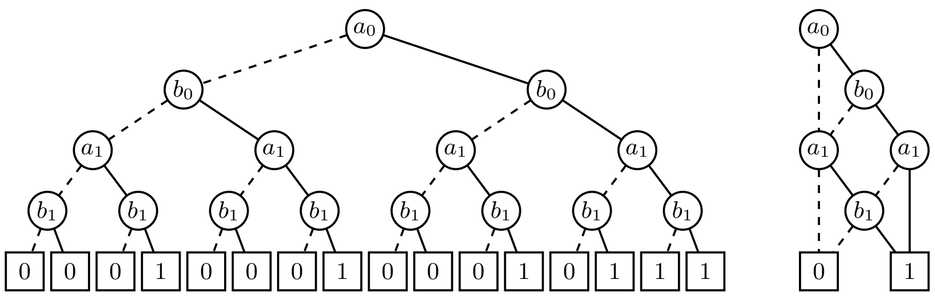

Figure 2.2. An ordered BDD representation of the Boolean function c_1 (left), and its canonical reduced representation (right). The solid lines represent high paths and the dashed lines represent low paths.

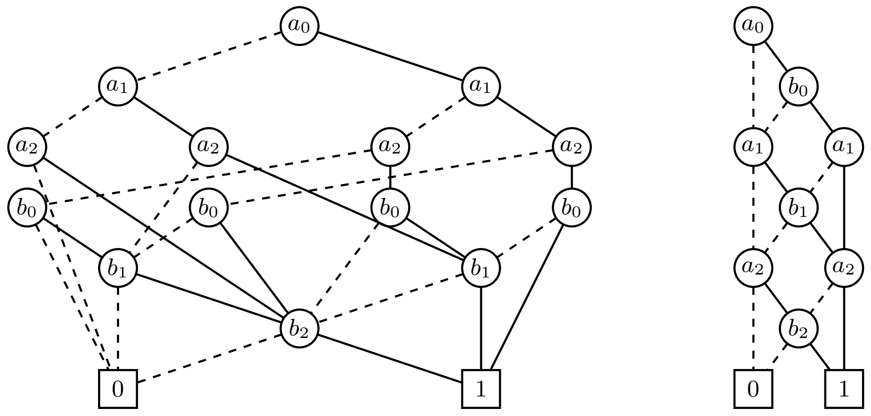

Figure 2.3. Reduced BDD representation of the Boolean function c_2 using the variable order $a_0, a_1, a_2, b_0, b_1, b_2$ (left) and $a_0, b_0, a_1, b_1, a_2, b_2$ (right).

Chapter 3

Target architecture of the superoptimizer

The superoptimizer implemented as part of this thesis project targets the x86 architecture, as described by the manuals published by Intel Corporation [12, 13, 14].

Since the x86 architecture is large and quite complex, it should be stressed that the superoptimizer only supports a very small subset of its features. While there are hundreds of instructions in the x86 general purpose instruction set, and many more in its numerous extensions, the superoptimizer's instruction set contains only 33 instructions. These instructions operate only on registers and flags. Using immediate values or adressing the memory is not supported.

The assembly language used to describe programs accepted as input or produced as output by the superoptimizer is similar, but not identical, to the one used by Intel. Programs generated by the superoptimizer can, with minor syntactical modifications, be assembled and executed on a real x86 machine.

3.1 Registers

Registers are used to hold integer values, and are read from and written to by instructions. Two's-complement arithmetic is used to represent negative numbers, but since registers are typeless, whether a given register contains a signed or an unsigned integer is determined solely by how it is used. Indeed, registers do not have to represent integers, but can be seen simply as sets of bits.

There are 8 registers in the superoptimizer's architecture, each 32 bits in size.

3.2 Flags

Flags are single-bit values used to signify various conditions. They are written to and read from by instructions, in general as a side effect rather than directly.

There are 17 different flags in the x86 architecture, but the superoptimizer only supports the 5 for which a corresponding `cmov` instruction (see below) exists: the *carry* (`cf`), *overflow* (`of`), *parity* (`pf`), *sign* (`sf`) and *zero* (`zf`) flags.

`cf` and `of` are used to signify that overflow has occurred, meaning that the result of an operation had to be truncated to fit into a single register. `cf` is used for unsigned arithmetic, while `of` is used for signed arithmetic, but their exact meaning varies from instruction to instruction.

`pf`, `sf` and `zf` can be determined entirely by inspecting the result of an operation. `pf` signifies that an even number of the least significant 8 bits of the result are set to 1, `sf` that the result is negative if interpreted as a signed number, since the most significant bit is 1, and `zf` that every bit of the result is 0.

3.3 Language

Programs accepted as input or produced as output by the superoptimizer, are written in assembly language, using Intel's syntax with a few modifications. Registers are referred to as `r0` through `r7`, rather than the usual names `eax`, `ebx`, etc. Statements are terminated by semicolons rather than newlines.

For instructions that take one or two arguments, the first is referred to as the *destination* register, while the second is referred to as the *source* register. Despite its name, the destination register may be read from, written to, or both. The source register, however, is only read from.

An example of a program written in the superoptimizer's assembly language is the following, which sets `r0` to the sum of `r1` and `r2`:

```
mov r0,r1; add r0,r2;
```

3.4 Instruction set

When selecting a set of instructions for a superoptimizer to support, some care should be taken. Only instructions which are likely to be used should be included, to avoid increasing the branching factor of the superoptimizer's search tree. Further, since some instructions take longer than others to execute on an x86 architecture [15], one might wish to avoid including slower instructions, to ensure that a program optimal in terms of length, also performs well in terms of execution time.

Since the focus of this thesis project is on testing programs for equivalence, however, the set of supported instructions was chosen to exhibit variety in terms of behavior with regards to registers and flags, and to demonstrate both the strengths and the weaknesses of the method used. The supported instructions are listed in Table 3.1.

3.4.1 Flag control instructions

The simplest supported instructions are the flag control instructions. `stc` and `clc` set `cf` to 1 and 0, respectively. `cmc` complements `cf`, flipping it from 0 to 1 or vice versa. These instructions do not modify registers in any way.

instruction	d	s	cf	of	pf	sf	zf
stc			W				
clc			W				
cmc			RW				
mov	W	R					
cmovc	RW	R	R				
cmovo	RW	R		R			
cmovp	RW	R			R		
cmovs	RW	R				R	
cmovz	RW	R					R
cmovnc	RW	R	R				
cmovno	RW	R		R			
cmovnp	RW	R			R		
cmovns	RW	R				R	
cmovnz	RW	R					R
cmova	RW	R	R				R
cmovbe	RW	R	R				R
cmovg	RW	R		R		R	R
cmovge	RW	R		R		R	
cmovl	RW	R		R		R	
cmovle	RW	R		R		R	R
and	RW	R	W	W	W	W	W
or	RW	R	W	W	W	W	W
xor	RW	R	W	W	W	W	W
not	RW						
add	RW	R	W	W	W	W	W
adc	RW	R	RW	W	W	W	W
sub	RW	R	W	W	W	W	W
sbb	RW	R	RW	W	W	W	W
cmp	R	R	W	W	W	W	W
inc	RW			W	W	W	W
dec	RW			W	W	W	W
neg	RW		W	W	W	W	W
imul	RW	R	W	W	U	U	U

Table 3.1. The instruction set of the superoptimizer. The columns marked *d* and *s* describe the instruction's behaviour with regards to its destination and source register, respectively, while the columns marked `cf`, `of`, `pf`, `sf` and `zf` describe its behaviour with regards to flags. *R* and *W* mark a register or flag as read from or written to, respectively, while *U* marks a flag as left in an undefined state by an instruction.

3.4.2 Data transfer instructions

The data transfer instructions copy the value of the source register to the destination register. `mov` does so unconditionally, while the `cmov` family of instructions does so only if a given condition is true, otherwise leaving the destination register unmodified. There are 16 `cmov` instructions in the superoptimizer's instruction set.

`cmovc`, `cmovo`, `cmovp`, `cmovs` and `cmovz`, as well as `cmovnc`, `cmovno`, `cmovnp`, `cmovns` and `cmovnz`, only inspect the value of a single flag, transferring data if that flag is set to an expected value. `cmova`, `cmovbe`, `cmovg`, `cmovge`, `cmovl` and `cmovle`, on the other hand, are slightly more complex, since they inspect the values of several flags, comparing them not only to expected values, but also to each other.

As shown in Table 3.1, `mov` is considered to read from one register and write to another, while the `cmov` instructions are considered to read from both, and write to one. This, together with a requirement that instructions only read from registers with defined values, ensures that the value of the destination register remains well-defined.

The data transfer instructions do not modify any flags.

3.4.3 Logic instructions

The instruction set contains four instructions that perform bitwise Boolean logic.

`and`, `or` and `xor` all take two arguments, and perform bitwise logical *and*, *inclusive-or* and *exclusive-or*, respectively. These instructions always set `cf` and `of` to 0, while `pf`, `sf` and `zf` are set based on the result as described in Section 3.2.

`not` takes a single argument and performs bitwise logical negation on it. It does not modify any flags.

3.4.4 Addition and subtraction instructions

A large number of the supported instructions perform some variant of addition or subtraction.

The most basic of these is `add`, which takes two arguments and performs integer addition on them. Since two's-complement arithmetic is used, `add` could be considered to perform both signed and unsigned addition simultaneously, with `cf` and `of` set to indicate unsigned and signed overflow, respectively. `pf`, `sf` and `zf` are set based on the result as described in Section 3.2.

`adc` performs addition with *carry-in*, meaning that it adds not only its two arguments, but also `cf`.

`sub` and `sbb` are the subtraction counterparts of `add` and `adc`. `cmp` behaves the same as `sub` with regards to flags, but does not write its result to a register.

`inc` and `dec` behave like `add` and `sub`, but take only one argument, substituting the value 1 for the second argument. They do not alter the value of `cf`, but modify the other flags in the same way as `add`.

`neg` takes one arguments and performs two's-complement negation on it. It sets `cf` to 0 if its argument is 0, and 1 otherwise, and modifies the other flags in the same way as `add`.

3.4.5 Multiplication instructions

The x86 instruction set contains many multiplication instructions, each with slightly different behavior. The only instruction supported by the superoptimizer is the form of `imul` that takes two arguments and performs integer multiplication on them, truncating the result to fit into a single register.

`cf` and `of` are both used to indicate that the result of the multiplication did not fit into a single register prior to being truncated, meaning that overflow has occurred. `pf`, `sf` and `zf`, however, are left in an *undefined* state.

Chapter 4

Design and implementation of the superoptimizer

The superoptimizer implemented as part of this thesis project takes as its input a target program written in the assembly language described in Section 3.3. It then begins generating candidate programs in the same assembly language, testing them for equivalence with the target program. Programs are generated in order of increasing length, meaning that the first program found to be equivalent to the target program, will be optimal in terms of length. The superoptimizer continues generating and testing programs until it reaches a user-specified maximum length, or is terminated by the user.

The superoptimizer was implemented using C and C++. A suite of unit tests was created to aid development, but no formal verification of correctness was performed.

4.1 Determining register usage

The first action performed by the superoptimizer, is to determine the input and output registers of the target program. This information is used both when generating candidate programs, and when determining whether they are equivalent to the target program.

Any register that is read from before it is written to by the target program, is marked as an input register. Input registers are assumed to contain well-defined values at the start of the program's execution. It should be noted that the superoptimizer does not require that generated candidate programs read from all, or even any, of these registers, since it is possible that some inputs to the target programs are superfluous.

The last register written to by the target program, is marked as the output register. Generated programs are required to use the same output register as the target program. Indeed, the superoptimizer determines equivalence based solely on the output register, ignoring the values of all other registers as well as the flags. Programs which compute the same function, but write the result to different registers, are not considered equivalent.

4.2 Generating candidate programs

The superoptimizer uses an iterative deepening depth-first search to find programs equivalent to the target program. Initially, the depth limit is 1, meaning that programs of of length 1 are generated. Once all programs of length 1 have been generated or pruned, the depth limit is increased to 2, and so on. This continues until a user-specified maximum length has been reached, or the superoptimizer is terminated by the user.

The superoptimizer prunes the search tree by not generating some types of programs which cannot possibly be equivalent to the target program.

In particular, the superoptimizer does not generate programs that read from registers or flags whose values are not well-defined. For this, the superoptimizer keeps a *live set* of registers, which initially consists of the input registers, as well as a live set of flags, which initially is empty. Only instructions that read from registers in the live set, or instructions that read no registers at all, are generated. Similarly, instructions that read flags are only generated if those flags are in the live set. After generating an instruction that writes to a register or flags, the superoptimizer adds that register or those flags to their respective live sets.

A few instructions are given special treatment by the superoptimizer.

`xor`, `sub` and `sbb` can all be used to set a register to a well-defined value, regardless of the register's previous value, by using that register both as the destination and source register. The superoptimizer therefore has a special rule which permits, but does not require, these instructions to read from, and write to, a register not in the live set.

For `mov` and the `cmov` instructions, the destination register is required to be different from the source register, since the instruction would otherwise have no effect. For `mov`, the destination register is also required to be a register *not* in the live set, or the program's output register. While *allowing* `mov` to write to a register not in the live set is necessary, since there would otherwise be no way of making a copy of a value required by multiple instructions, it should be noted that *requiring* it to do so is a flaw in the design of the superoptimizer, which sometimes produces programs that use too many registers unnecessarily.

When generating the very last instruction in a candidate program, the superoptimizer requires that the instruction writes to the program's output register, since the instruction would otherwise have no effect.

4.3 Determining equivalence with the target program

Each generated candidate program is tested for equivalence with the target program using a probabilistic test similar to the one used by GSO. This test is fast, but may report false positives. Programs that pass the probabilistic test are therefore verified by being expressed as Boolean-logic functions represented by BDDs. This verification is slower than the probabilistic test, but yields correct results, barring any bugs in its implementation.

The probabilistic test and BDD-based method of verification can be seen as two independent implementations of the architecture defined in Chapter 3. If should be noted, however, that neither of them are capable of correctly handling programs which read from undefined registers or flags, since such programs are not expected to be generated.

4.3.1 Testing for equivalence probabilistically

The superoptimizer's probabilistic method of testing a candidate program for equivalence with the target program, is to simply execute both programs with identical input and compare their output.

Like GSO, the superoptimizer implemented as part of this thesis project does not execute the instructions in a program directly on the host architecture, but rather simulates them. These simulated instructions, which are written in C, operate on registers represented by unsigned integers and flags represented by single bits. Since the instructions are generally simple enough to be implemented in a straight-forward manner using only a few C operators, they will not be described in further detail.

When testing a candidate program for equivalence with the target program, the superoptimizer uses two sets of registers and flags, with the registers in the two sets given identical values taken from a hardcoded test vector.

The superoptimizer executes the candidate and target program, one on each set of registers and flags, and then compares the values of the output registers in the two sets. If they differ, the programs are certainly not equivalent, and the test ends. If they are equal, the test starts over, using new values taken from the test vector. When all values in the test vector have been used, the candidate program passes the test, as it is equivalent to the target program at least for the values used.

Since it is still possible that the candidate program is in fact not equivalent to the target program in the general case, one might ask why the programs are not tested using *all* possible values. Indeed, for a program with a single 32-bit input register, there are only 2^{32} possible values to test. Considering that programs generated by a superoptimizer are no longer than 5 or 6 instructions, running them 2^{32} times should take less than a minute on a modern computer.

This method will not work well even for programs with two input registers, however. Such a program would need to be run 2^{64} times, which would take several thousand years.

4.3.2 Verifying equivalence using BDDs

To determine if two programs are equivalent, one can express both in Boolean logic, reduce them to some canonical form, and compare them. As described in

Section 2.2, reduced and ordered BDDs are canonical representations of Boolean functions, and can therefore be used for this purpose.

The superoptimizer's method of determining program equivalence using BDDs was implemented in C++, using the software package BuDDy [16]. BuDDy was chosen out of a small pool of alternatives, for its reasonably well-written documentation and apparent popularity.

Overview

The superoptimizer keeps a reference to a BDD representing a Boolean function for each bit in each register, and for each flag. All bits and flags are represented in a single, shared BDD. Initially, the bits in a register a reference the variables $\langle a_0, a_1, ..., a_{n-1}\rangle$, where n is the number of bits in the register. The variables are ordered so that those which represent bit 0 in a register all come before those which represent bit 1, and so on, which was found to work well in practice.

When creating a BDD representation of a program, the superoptimizer steps through its instructions in order. For each instruction, the superoptimizer performs Boolean operations on the existing BDDs, changing the references in the instruction's destination register to point to new BDDs. With each operation, the shared BDD is reduced, which takes some time.

The initial state of the BDD model, as well as the state after applying an addition instruction, are shown in a simplified way in Figure 4.1. Note that the registers in the figure only contain 2 bits, and that not all references are shown. The implementation of the addition instruction is detailed below.

Once the superoptimizer has stepped through all instructions in the program, the bits in the program's output register will be represented by BDDs that are canonical representations of Boolean functions on the initial variables.

To determine if a candidate program is equivalent to the target program, the superoptimizer performs the process described above for both programs, and then compares the references to the BDDs representing the bits in their output registers pairwise. If all references are equal, the two programs are equivalent.

For each instruction in the superoptimizer's instructions set, a corresponding function was created to apply the equivalent Boolean-logic operations on the registers and flags. In the descriptions of these functions below, a register a is represented by n Boolean variables $\langle a_0, a_1, ..., a_{n-1}\rangle$, where a_0 and a_{n-1} are the least and most significant bits of a, respectively. The flags are represented as f_c, f_o, f_p, f_s and f_z, and are treated as global variables.

Flag control instructions

The flag control instructions are trivial to express in Boolean logic, with STC and CLC setting f_c to 1 and 0, respectively, and CMC setting f_c to its logical negation:

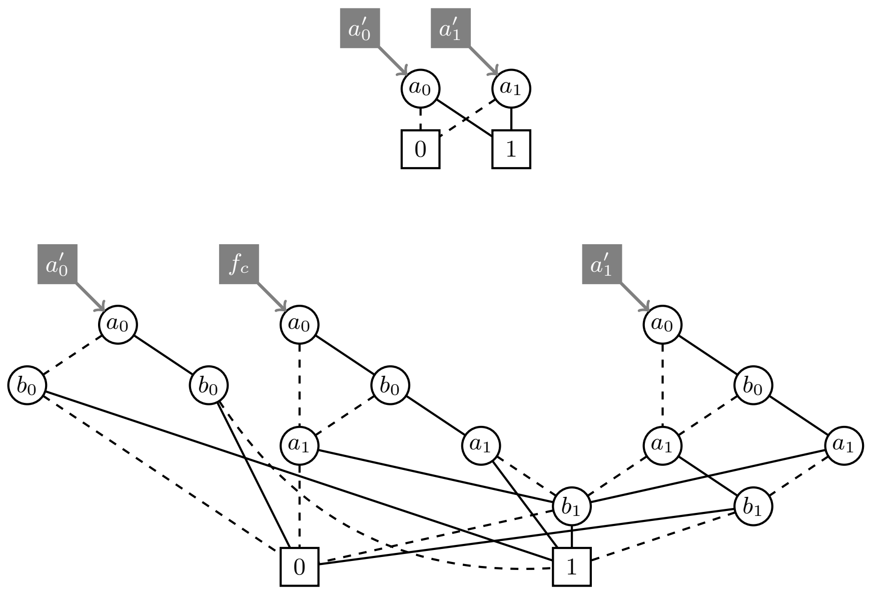

Figure 4.1. A simplified 2-bit version of the BDD model used by the superoptimizer in its initial state (top) and after applying the function ADD(a, b) (bottom). a'_0 and a'_1 mark the BDDs currently representing bits 0 and 1 in register a, while f_c marks the BDD representing the carry flag. a_0, a_1, b_0 and b_1, are the variables initially referenced by the bits in a and b. Neither the BDDs representing the bits in b, nor those representing the other flags, are marked.

```
STC()
    $f_c \leftarrow 1$
```

```
CLC()
    $f_c \leftarrow 0$
```

```
CMC()
    $f_c \leftarrow \neg f_c$
```

Data transfer instructions

The MOV instruction is straight-forward, since it contains no BDD operations, and consists only of an array assignment:

```
MOV($a, b$)
    for $i \leftarrow 0$ to $n - 1$
        $a_i \leftarrow b_i$
```

The CMOV instructions are slightly more complex. Using Boolean logic, a conditional value can be expressed as $(\neg c \wedge a) \vee (c \wedge b)$, where c is the condition, and a and b the two possible values. Since the condition is the same for each bit, the CMOV instructions all have the following form, with different definitions of *condition*:

```
CMOV$(a, b)$
    $c \leftarrow$ condition
    for $i \leftarrow 0$ to $n-1$
        $a_i \leftarrow (\neg c \wedge a_i) \vee (c \wedge b_i)$
```

Logic instructions

The logic instructions are similarly straight-forward, since each bit i in the result depends only the corresponding bit i in the input or inputs.

AND, OR and XOR are all similar in form:

```
AND$(a, b)$
    for $i \leftarrow 0$ to $n-1$
        $a_i \leftarrow a_i \wedge b_i$
    $f_c \leftarrow 0$
    $f_o \leftarrow 0$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

```
OR$(a, b)$
    for $i \leftarrow 0$ to $n-1$
        $a_i \leftarrow a_i \vee b_i$
    $f_c \leftarrow 0$
    $f_o \leftarrow 0$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

```
XOR$(a, b)$
    for $i \leftarrow 0$ to $n-1$
        $a_i \leftarrow a_i \oplus b_i$
    $f_c \leftarrow 0$
    $f_o \leftarrow 0$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

NOT is even simpler, in that it only takes a single argument:

```
NOT($a$)
    for $i \leftarrow 0$ to $n-1$
        $a_i \leftarrow \neg a_i$
```

Addition and subtraction instructions

The instructions that perform addition or subtraction are somewhat more complex, since each bit i in the result depends on bits $0 - -i$ in the input or inputs. They are all implemented as *ripple-carry adders*, with c_i being the carry-out of position i.

ADD and ADC are defined similarly, differing only in that ADC uses f_c as a *carry-in*:

```
ADD($a, b$)
    $c_0 \leftarrow a_0 \wedge b_0$
    $a_0 \leftarrow a_0 \oplus b_0$
    for $i \leftarrow 1$ to $n-1$
        $c_i \leftarrow (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$
        $a_i \leftarrow a_i \oplus b_i \oplus c_{i-1}$
    $f_c \leftarrow c_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

```
ADC($a, b$)
    $c_0 \leftarrow (a_0 \wedge b_0) \vee (a_0 \wedge f_c) \vee (b_0 \wedge f_c)$
    $a_0 \leftarrow a_0 \oplus b_0 \oplus f_c$
    for $i \leftarrow 1$ to $n-1$
        $c_i \leftarrow (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$
        $a_i \leftarrow a_i \oplus b_i \oplus c_{i-1}$
    $f_c \leftarrow c_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

SUB and SBB are also very similar:

```
SUB$(a, b)$
    $c_0 \leftarrow \neg a_0 \wedge b_0$
    $a_0 \leftarrow a_0 \oplus b_0$
    for $i \leftarrow 1$ to $n - 1$
        $c_i \leftarrow (\neg a_i \wedge b_i) \vee (\neg a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$
        $a_i \leftarrow a_i \oplus b_i \oplus c_{i-1}$
    $f_c \leftarrow c_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

```
SBB$(a, b)$
    $c_0 \leftarrow (\neg a_0 \wedge b_0) \vee (\neg a_0 \wedge f_c) \vee (b_0 \wedge f_c)$
    $a_0 \leftarrow a_0 \oplus b_0 \oplus f_c$
    for $i \leftarrow 1$ to $n - 1$
        $c_i \leftarrow (\neg a_i \wedge b_i) \vee (\neg a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$
        $a_i \leftarrow a_i \oplus b_i \oplus c_{i-1}$
    $f_c \leftarrow c_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

CMP is defined almost exactly as SUB, with the only difference being that the result of the subtraction is stored in a temporary register t, which is used to when setting f_p, f_s and f_z:

```
CMP$(a, b)$
    $c_0 \leftarrow \neg a_0 \wedge b_0$
    $t_0 \leftarrow a_0 \oplus b_0$
    for $i \leftarrow 1$ to $n - 1$
        $c_i \leftarrow (\neg a_i \wedge b_i) \vee (\neg a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$
        $t_i \leftarrow a_i \oplus b_i \oplus c_{i-1}$
    $f_c \leftarrow c_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(t_0 \oplus t_1 \oplus ... \oplus t_7)$
    $f_s \leftarrow t_{n-1}$
    $f_z \leftarrow \neg(t_0 \vee t_1 \vee ... \vee t_{n-1})$
```

INC and DEC behave like ADD and SUB, but with b_0 set to 1 and all other bits in b set to 0. Further, they do not modify f_c. Once simplified, their implementations are as follows:

```
INC($a$)
    $c_0 \leftarrow a_0$
    $a_0 \leftarrow \neg a_0$
    **for** $i \leftarrow 1$ **to** $n-1$
        $c_i \leftarrow a_i \wedge c_{i-1}$
        $a_i \leftarrow a_i \oplus c_{i-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

```
DEC($a$)
    $c_0 \leftarrow \neg a_0$
    $a_0 \leftarrow \neg a_0$
    **for** $i \leftarrow 1$ **to** $n-1$
        $c_i \leftarrow \neg a_i \wedge c_{i-1}$
        $a_i \leftarrow a_i \oplus c_{i-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

NEG performs two's-complement negation by logically negating each bit and adding 1. Once simplified, its implementation is as follows:

```
NEG($a$)
    $c_0 \leftarrow \neg a_0$
    **for** $i \leftarrow 1$ **to** $n-1$
        $c_i \leftarrow \neg a_i \wedge c_{i-1}$
        $a_i \leftarrow \neg a_i \oplus c_{i-1}$
    $f_c \leftarrow a_0 + a_1 + ... + a_{n-1}$
    $f_o \leftarrow c_{n-1} \oplus c_{n-2}$
    $f_p \leftarrow \neg(a_0 \oplus a_1 \oplus ... \oplus a_7)$
    $f_s \leftarrow a_{n-1}$
    $f_z \leftarrow \neg(a_0 \vee a_1 \vee ... \vee a_{n-1})$
```

Multiplication instructions

IMUL is the most complex of the supported instructions, in particular when represented by BDDs. Its implementation uses the *shift-and-add* method of multiplication, which is similar to the common method of long multiplication illustrated in Figure 4.2.

This methods works particularly well with binary or Boolean representations, since multiplication by a single digit then becomes nothing more than a logical *and*.

$$\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\ \hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\ \hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

Figure 4.2. Long multiplication of two 4-digit numbers.

Multiplication of two n-bit numbers thus becomes little more than n additions. Further, the terms do not all need to be calculated before adding them, but can be calculated as needed.

In the implementation below, a and b are assumed to have been extended to $2n$ bits, the new bits set to 0. Further, each bit in p is assumed to be set to 0 initially.

```
IMUL($a, b$)
    for $j \leftarrow 0$ to $n - 1$
        $x \leftarrow a_0 \wedge b_j$
        $c_j \leftarrow p_j \wedge x$
        $p_j \leftarrow p_j \oplus x$
        for $i \leftarrow j + 1$ to $2n - 1$
            $x \leftarrow a_{i-j} \wedge b_j$
            $c_i \leftarrow (p_i \wedge x) \vee (p_i \wedge c_{i-1}) \vee (x \wedge c_{i-1})$
            $p_i \leftarrow p_i \oplus x \oplus c_{i-1}$
    for $i \leftarrow 0$ to $n - 1$
        $a_i \leftarrow p_i$
    $f_c \leftarrow f_o \leftarrow p_n \vee p_{n+1} \vee ... \vee p_{2n-1}$
```

As seen, the last step of the implementation is to set a to the n least significant bits of p. f_c and f_o are then both set based on the remaining n most significant bits of p.

Since we really only need to know of any of the n most significant bits of p is 1, however, we can also implement the multiplication instruction as follows:

```
IMUL$(a, b)$
    for $j \leftarrow 0$ to $n - 1$
        $x \leftarrow a_0 \wedge b_j$
        $c_j \leftarrow p_j \wedge x$
        $p_j \leftarrow p_j \oplus x$
        for $i \leftarrow j + 1$ to $n - 1$
            $x \leftarrow a_{i-j} \wedge b_j$
            $c_i \leftarrow (p_i \wedge x) \vee (p_i \wedge c_{i-1}) \vee (x \wedge c_{i-1})$
            $p_i \leftarrow p_i \oplus x \oplus c_{i-1}$
        for $i \leftarrow n$ to $2n - 1$
            $o \leftarrow o \vee a_{i-j} \wedge b_j$
        $o \leftarrow o \vee c_{n-1}$
    for $i \leftarrow 0$ to $n - 1$
        $a_i \leftarrow p_i$
    $f_c \leftarrow f_o \leftarrow o$
```

Naturally, the equivalence of the two implementations was confirmed using the method described in this section.

Chapter 5

Results and conclusions

This chapter presents a number of results obtained experimentally. It attempts to determine the performance of the superoptimizer in general, and of its method of determining program equivalence using BDDs in particular.

While some of the results presented here are highly dependent not only on the characteristics of BDDs, but also on the specific implementation and hardware used, they do provide some insight into how feasible it is to use BDDs for the purposes outlined in this thesis.

All results presented in this chapter were obtained with the superoptimizer running on a 2.66 GHz Intel Core i7 processor.

5.1 Superoptimizing a program

The superoptimizer was used to find programs that compute the SIGN function, which can be expressed as follows in pseudocode:

```
SIGN($a$)
  if $a > 0$
      return 1
  if $a < 0$
      return $-1$
  return 0
```

As seen, SIGN takes a single number and returns 1, -1 or 0 if it is positive, negative, or equal to zero, respectively. The superoptimizer was given a target program computing `r1` $\leftarrow$ SIGN(`r0`), and made to search for equivalent programs up to 5 instructions in length.

The number of candidate programs generated by the superoptimizer, the number of programs that passed the probabilistic test, and the number of programs verified as equivalent to the target program using BDDs, are listed in Table 5.1. It should be noted that these numbers depend on the target program, the number of input

length	generated	passed probabilistic test	verified as equivalent
1	3	0	0
2	155	0	0
3	11849	0	0
4	1082001	2	1
5	121268867	326	209

Table 5.1. The number of candidate programs generated by the superoptimizer, the number of programs that passed the probabilistic test, and the number of programs verified as equivalent to the target program using BDDs.

registers, and that the output register is not also an input register. Even so, they do provide some information about the behavior of the superoptimizer.

Most importantly, the numbers show that the probabilistic test catches the vast majority, but not all, of the programs not equivalent to the target program. Despite this, a significant number of candidate programs that pass the probabilistic test are in fact not equivalent to the target program, although this could be seen as a sign of poorly chosen test vectors. An example of a candidate program that passes the probabilistic test without being equivalent to the target program is given below.

By looking at the number of generated programs of each length, one can also see that the branching factor increases with program length.

The superoptimizer required roughly 105 seconds to generate, test and verify these programs. Virtually all of the time was spent on the programs consisting of 5 instructions, with all programs consisting of 4 or instructions or less taking less than 1 second to generate, test and verify.

5.1.1 Analysis of generated programs

As seen in Table 5.1, the superoptimizer generated two programs consisting of 4 instructions that passed the probabilistic test, but only one of these was verified as equivalent to the target program. To illustrate how this happened, and also give an example of the strange and seemingly obfuscated nature of programs generated by the superoptimizer, we shall inspect these two programs.

First, the program that was verified as equivalent to the target programs, is as follows:

```
add r0,r0; sbb r1,r1; sub r1,r0; adc r1,r0;
```

At first, it is not obvious how this program computes `r1` ← SIGN(`r0`). To understand it, one can step through it instruction by instruction, for the three cases where `r0` is initially positive, negative, or equal to zero, while taking note of the values of `r0`, `r1` and `cf` following the execution of each instruction, as shown in Table 5.2. While strange, this program is indeed correct.

On the other hand, the program that passed the probabilistic test, but was rejected as not equivalent to the target program, is as follows:

```
add r0,r0; sbb r1,r1; neg r0; adc r1,r1;
```

	r0 > 0	r0 < 0	r0 $= 0$
`add r0,r0`	r0 $\leftarrow X$, cf $\leftarrow 0$	r0 $\leftarrow Y$, cf $\leftarrow 1$	r0 $\leftarrow 0$, cf $\leftarrow 0$
`sbb r1,r1`	r1 $\leftarrow 0$	r1 $\leftarrow -1$	r1 $\leftarrow 0$
`sub r1,r0`	r1 $\leftarrow -X$, cf $\leftarrow 1$	r1 $\leftarrow -1 - Y$, cf $\leftarrow 0$	r1 $\leftarrow 0$, cf $\leftarrow 0$
`adc r1,r0`	r1 $\leftarrow 1$	r1 $\leftarrow -1$	r1 $\leftarrow 0$

Table 5.2. Behavior of the program `add r0,r0; sbb r1,r1; sub r1,r0; adc r1,r0;` for different initial values of `r0`.

`add r0,r0`	r0 $\leftarrow 0$, cf $\leftarrow 1$
`sbb r1,r1`	r1 $\leftarrow -1$
`neg r0`	r0 $\leftarrow 0$, cf $\leftarrow 0$
`adc r1,r1`	r1 $\leftarrow -2$

Table 5.3. Behavior of the program `add r0,r0; sbb r1,r1; neg r0; adc r1,r1;` when the initial value of `r0` is the most negative 32-bit integer.

P_0	`mov r1,r0;`
P_1	`xor r1,r1; and r1,r0;`
P_2	`sub r1,r1; and r1,r0;`
P_3	`xor r1,r1; add r1,r0;`
P_4	`sub r1,r1; add r1,r0;`
P_5	`add r0,r0; sbb r1,r1; inc r1;`
P_6	`add r0,r0; sbb r1,r1; sub r1,r0; adc r1,r0;`
P_7	`add r0,r0; sbb r1,r1; neg r0; adc r1,r1;`
P_8	`xor r1,r1; xor r2,r2; xor r3,r3; inc r2; dec r3; cmp r1,r0; cmovl r1,r2; cmovg r1,r3;`

Table 5.4. The programs used to evaluate the performance of the superoptimizer's method of determining program equivalence using BDDs.

This program does, in fact, behave like the first one for every input value except one: the most negative 32-bit integer, -2^{31}, which was not included in the test vector used by the superoptimizer's probabilistic test. Using two's complement, this number is stored with its most significant bit set to 1, and all other bits set to 0. For this number, the program behaves as shown in Table 5.3.

While it could certainly be argued that the most negative 32-bit integer should have been included in the test vector, this program does help to illustrate that probabilistic testing is insufficient to fully determine the equivalence of two programs.

5.2 Performance of creating and comparing BDD representations of a few programs

The example given above shows that the superoptimizer is capable of generating, testing and verifying over 300 programs in less than two minutes. To gain a deeper insight into the amount of time spent determining program equivalence using BDDs, and to understand which factors influence the amount of time required, a special version of the superoptimizer, with added timing functionality, was created. This version of the superoptimizer was used to measure the time required to create BDD representations of the 9 programs labeled P_0 to P_8 in Table 5.4, as well as the time required to determine their pairwise equivalence.

The programs were all written to have `r0` and `r1` as their input and output

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
0.119 ms	0.175 ms	0.284 ms	0.281 ms	0.389 ms	0.447 ms	0.966 ms	0.679 ms	0.673 ms

Table 5.5. The time required to create BDD representations of the programs listed in Table 5.4.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
P_0	0.116 ms	0.176 ms	0.284 ms	0.281 ms	0.389 ms	0.447 ms	0.965 ms	0.679 ms	0.670 ms
P_1		0.235 ms	0.343 ms	0.340 ms	0.449 ms	0.507 ms	1.025 ms	0.757 ms	0.729 ms
P_2			0.468 ms	0.449 ms	0.553 ms	0.613 ms	1.131 ms	0.845 ms	0.837 ms
P_3				0.422 ms	0.529 ms	0.592 ms	1.107 ms	0.821 ms	0.834 ms
P_4					0.636 ms	0.698 ms	1.212 ms	0.927 ms	0.941 ms
P_5						0.752 ms	1.270 ms	0.984 ms	1.001 ms
P_6							1.781 ms	1.535 ms	1.530 ms
P_7								1.171 ms	1.224 ms
P_8									1.155 ms

Table 5.6. The time required to determine pairwise equivalence of the programs listed in Table 5.4.

registers, respectively. Additionally, P_0 is equivalent to P_3 and P_4, as is P_1 to P_2, and P_6 to P_8, while P_5 and P_7 are not equivalent to any test program other than themselves. Finally, P_0 was written to provide a baseline measurement of the method's overhead, as it consists of a single `mov` instruction, which is implemented without any actual BDD operations.

The time required for the superoptimizer to create BDD representations of the programs is shown in Table 5.5. As expected, P_0 requires the least time. The entries for $P_1 - P_4$ illustrate the complexity of addition and subtraction instructions relative to logic instructions. Looking at $P_4 - P_7$, which all consist entirely of addition and subtraction instructions, one might infer that longer programs require more time. This conclusion is weakened, however, by the considerable difference between the entries for P_6 and P_7, which are of the same length, as well as the entry for P_8, which contains twice as many, albeit simpler, instructions as P_6, and requires less time.

The time required for the superoptimizer to determine if the programs are equivalent to each other is shown in Table 5.6. It should be noted that the time required to determine if two programs P_a and P_b are equivalent, is roughly equal to the sum of the time required to create BDD representations of the programs, minus the baseline time of P_0. This shows, as expected, that virtually all time is spent creating the BDD representations of the programs, rather than comparing them.

The superoptimizer requires the most time for determining if P_6 is equivalent to itself, although this still takes less than 2 milliseconds. From this we can draw the conclusion that the superoptimizer spent very little time verifying programs in the example given in Section 5.1, and that the performance bottlenecks are to be found elsewhere.

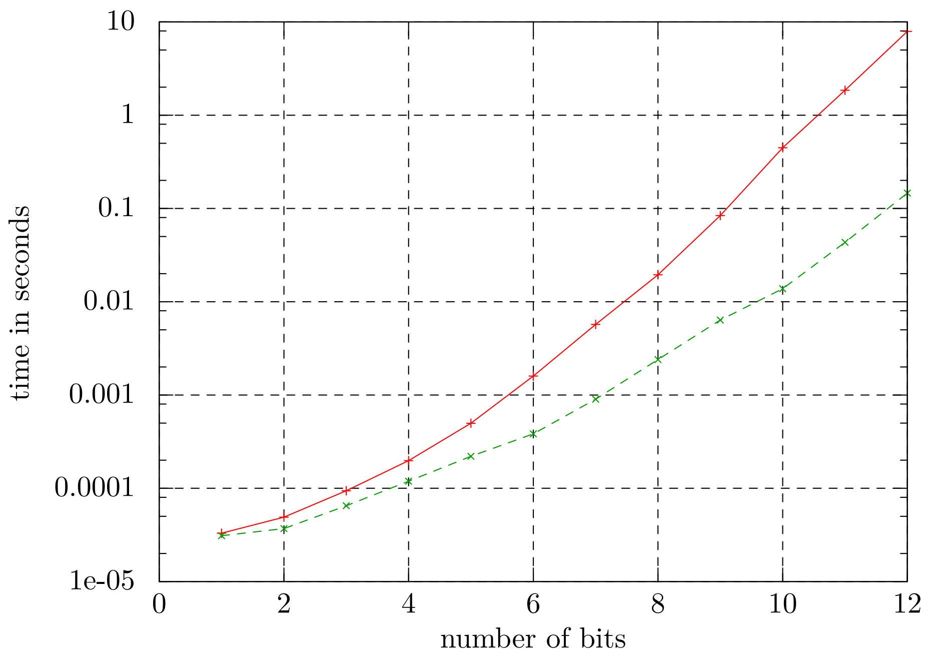

Figure 5.1. The time required to create a BDD representation of a single n-bit multiplication instruction as a function of n. The solid line represents the implementation of the multiplication instruction that determines the full product, while the dashed line represents the implementation that only determines the bits necessary to set the carry and overflow flags. Note that the time is displayed on a logarithmic scale.

5.3 Performance of creating BDD representations of programs containing multiplication instructions

While the results presented in Section 5.2 seem very promising, it should be noted that none of the programs in Table 5.4 contain any multiplication instructions. The reason for this is unfortunately that creating a BDD representation of even a single 32-bit multiplication instruction turned out to take an unreasonable amount of time, and that the multiplication instruction was therefore removed from the superoptimizer's instruction set.

To determine the time required for the superoptimizer to create BDD representation of multiplication instructions operating on fewer bits, another special version of the superoptimizer, with support for an architecture with a variable number of bits per registers, was created. This version of the superoptimizer was used to measure the time required to create BDD representations of both implementations of the multiplication instruction described in Section 4.3.2. The results are shown in Figure 5.1.

While the implementation that only determines the bits necessary to set the carry and overflow flags, is noticably more efficient than the implementation that determines the full product, the superoptimizer still requires over 100 ms to create a BDD representation of a 12-bit multiplication.

Further, the time required to create a BDD representation of an n-bit multiplication instruction appears to grow exponentially as a function of n, making support for 32-bit multiplication instructions impossible in practical terms.

Chapter 6

Recommendations and future work

While the method of using BDDs to determine program equivalence described in this thesis works very well for many programs, it performs poorly for programs containing multiplication instructions. For this reason, it is hard to give it an unconditional recommendation.

The performance could perhaps be improved by replacing the shift-and-add implementation of the multiplication instruction with a more efficient algorithm. While the resulting BDD would be identical once reduced, a more efficient algorithm might require fewer operations to construct it.

Another possibility to consider is using 8-bit or 4-bit, rather than 32-bit, arithmetic when creating BDD representations of programs. The only architectural feature currently supported by the superoptimizer for which this would not work, is the parity flag, which is defined specifically by the 8 least significant bits of a result. Since the parity flag is of limited utility, removing support for it would perhaps not be an issue. A more serious problem, however, is that adding support for immediate values would become difficult. It should also be noted, that with 8-bit or 4-bit arithmetic, testing programs with all possible input values as described in Section 4.3.1 is feasible even for programs with many input registers.

Even without support for multiplication and similarly complex instructions, the method of using BDDs to determine program equivalence described in this thesis could be made useful. There are many simple but important instructions, such as shift and rotate, for which support could be added. One could also add support for programs with multiple output registers with relative ease.

As a final thought, the method of using BDDs to determine program equivalence might be best utilized stripped of the program generator and the probabilistic test, intended to be used as an external verification tool for a more capable superoptimizer or similar piece of software.

Bibliography

[1] H. Massalin, "Superoptimizer: a look at the smallest program," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–126, IEEE Computer Society Press, 1987.

[2] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the GNU C compiler," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 341–352, ACM, 1992.

[3] R. Joshi, G. Nelson, and K. Randall, "Denali: a goal-directed superoptimizer," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 304–314, ACM, 2002.

[4] R. Joshi, G. Nelson, and Y. Zhou, "Denali: a practical algorithm for generating optimal code," *ACM Transactions on Programming Languages and Systems*, vol. 28, pp. 967–989, November 2006.

[5] S. Bansal, *Peephole superoptimization.* PhD thesis, Stanford University, 2008.

[6] T. Crick, *Superoptimisation: provably optimal code generation using answer set programming.* PhD thesis, University of Bath, 2009.

[7] C. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, July 1959.

[8] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

[9] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

[10] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 52–57, ACM, 1990.

[11] B. Bollig and I. Wegener, “Improving the variable ordering of OBDDs is NP-complete,” *IEEE Transactions on Computers*, vol. 45, pp. 993–1002, September 1996.

[12] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*, January 2011. Order number 253665-037US.

[13] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A: Instruction Set Reference, A-M*, January 2011. Order number 253666-037US.

[14] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B: Instruction Set Reference, N-Z*, January 2011. Order number 253667-037US.

[15] T. Granlund, “Instruction latencies and throughput for AMD and Intel x86 processors,” March 2011. `http://gmplib.org/~tege/x86-timing.pdf`.

[16] “BuDDy.” `http://buddy.sourceforge.net`.

TRITA-CSC-E 2011:144
ISRN-KTH/CSC/E--11/144-SE
ISSN-1653-5715