FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ISOLATING ERRORS FOR AN ASSEMBLY OPTIMIZER

By

ABIGAIL MORTENSEN

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2022

Abigail Mortensen defended this thesis on April 27, 2022.
The members of the supervisory committee were:

David Whalley
Professor Directing Thesis

Xiaonan Zhang
Committee Member

Grigory Fedyukovich
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

I would like to thank Dr. David Whalley for his continued guidance, encouragement, and assistance throughout this process, for which I am so very grateful. I thank Dr. Gang-Ryung Uh and Dr. Soner Onder for their help with the planning and implementation of the simulation testing environment that was used for this work. Many thanks also goes out to Joseph Zilonka, Skylar Scorca, and Jack Patten for their contributions in regard to testing the isolator, providing suggestions for improving the isolator, and their support.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This paper introduces *asoptiso*, which is an assembly optimizer error isolator. *asopt* is an assembly optimizer for a set of new ISAs, called the SCALE ISAs, and is a tool which is significantly difficult to debug when it produces erroneous assembly-optimized code that fails to correctly execute during simulation. *asopt* is difficult to debug due to two reasons. First, simulations can often take a long time to test the execution of assembly-optimized code. Second, there are often a large number of transformations which *asopt* has applied to the program, any of which could be the cause of error. Manually searching for the erroneous transformations among all applied transformations is highly impractical. I have created *asoptiso* to automate and simplify the process of isolating the first erroneous transformation applied by the assembly optimizer. This tool has proved extremely useful in speeding up the testing of *asopt* by not only automating error isolation, but also providing the user with a variety of useful features.

CHAPTER 1

INTRODUCTION

There are many challenges when isolating errors in a compilation infrastructure for a new instruction set architecture (ISA). (1) Application benchmarks can consist of tens or even hundreds of thousands of lines of source code, which can easily expand by a factor of 10 to machine instructions after compilation. The number of lines of code is significantly increased when taking into account the library code that must also be compiled and comprises part of the executables. An application with corresponding libraries may easily consist of millions of instructions. Large applications significantly increase the challenge of isolating where a problem in the code exists that causes incorrect simulation. (2) Applications can simulate billions or even trillions of dynamic instructions. For instance, all SPEC 2006 benchmarks run for at least several hundred billion to multiple trillion instructions. Simulation is required for a new ISA and is orders of magnitude slower than native execution. This can lead to long simulation times, even for functional simulation, which can exacerbate the challenge of isolating errors in a compiler. (3)Compilers are complex system software tools. An instruction set can consist of hundreds of different types of instructions even for RISC instruction sets. The number of distinct instructions can directly impact the complexity of a compiler for that instruction set. The most complex portion of the code generation of a compiler is often the calling conventions associated with the instruction set. Likewise, the analysis and transformations to support optimizations introduce much complexity in compilers. The challenge of isolating errors in a compiler is exacerbated when the ISA is new and a working processor, simulator, and compiler for the ISA do not yet exist. Even once a problem has been isolated in an application, finding the cause of the problem in a compiler can be quite challenging. Due to these challenges, a systematic and automatic method for isolating errors is needed.

I have created an assembly optimizer error isolator, named *asoptiso*, which automates the process of isolating the **first** erroneous transformation which is causing the simulation error of a SCALE assembly optimized program. In this thesis, I will first describe the SCALE ISAs, simulation infrastructure, and compilation infrastructure. I then describe various code transformations, along with how to count transformations in the context of either an assembly file or program. I next

describe the process of error isolation within *asoptiso* and various useful features of the assembly optimizer error isolator. Finally, I will share the results and conclusions for my thesis.

1.1 Simulation and Compilation Infrastructure

1.1.1 The SCALE ISAs

This research is part of the NSF SCALE project, which involves developing both simulation and compilation support for a set of related, but distinct ISAs. Two ISAs have currently been implemented in this project, which are the *SCALE base* ISA and the *SCALE VLIW* ISA.

The SCALE base ISA is similar to the MIPS ISA with a number of small differences to allow other information to be encoded in the instruction set. A few of these differences includes memory references and branches. All SCALE loads and stores are only supported by a register deferred addressing mode, meaning that all loads and stores use a zero displacement from the base register. The rationale for this restriction is that it decreases the number of stages in the instruction pipeline and allows other information supporting more advanced features to be encoded with loads and stores in SCALE ISAs. In fact, many processors will dynamically split a load or store into an address generation instruction and a memory reference instruction anyway and treat them as separate instructions within the processor. By splitting these instructions at compile time, we expose these instructions to more compiler optimizations to avoid redundant effective address calculations and to more effectively schedule operations. We also support only *bnez* (branch not equal to zero) and *beqz* (branch equal to zero) instructions for integer branches. This means that each integer branch references a single register. To allow only these branches to be utilized we added a new *seq* instruction that is similar to an *slt* instruction in that it sets a destination register to 1 if the two registers being compared are equal or to 0 otherwise.

The SCALE VLIW ISA uses instructions within the SCALE base ISA, but supports very long instruction word (VLIW) execution. The generated code must be packaged into groups of independent instructions that are simultaneously issued. We refer to such groups as *VLIW packs* and the position of a SCALE instruction within a VLIW pack as a *lane*. The number of instructions in each VLIW pack and which types of SCALE instructions can be placed in each lane is configurable at compile and simulation time. The SCALE VLIW ISA allows instructions to follow a branch within a VLIW pack. An instruction after a branch within a VLIW pack is only committed if the branch is predicted to be not taken. Thus, conditional branches support a simple form of

predication in the SCALE VLIW ISA. An instruction after an unconditional transfer of control (jump, call, return) within a pack is never executed. Hence, only *nop* instructions should be placed after an unconditional transfer of control within a VLIW pack. There can be multiple transfers of control within a VLIW pack, but only the last transfer of control can be unconditional.

1.1.2 ADL Simulation System

We use the ADL simulation system, which takes a microarchitecture specification file written in the Architecture Description Language (ADL) as input and automatically produces an assembler, linker, and disassembler [7]. ADL provides constructs for specifying (1) microarchitectural features including pipelines, control, and memory hierarchy and (2) the instruction set architecture including the assembly syntax and corresponding binary representation. The assembler and linker together are used to produce a statically linked executable that is invoked by the ADL produced simulators. The simulators the ADL system produces can be functional or cycle accurate. These simulators perform a more realistic simulation than many commonly used simulators as instructions are actually fetched from the instruction cache, data values are actually loaded from the data cache, values are actually forwarded through the pipeline, etc. This more realistic simulation helps to ensure that the described techniques are correctly implemented and hence provides more reliable statistics.

The SCALE base ISA is used for both a SCALE functional simulator and a SCALE pipelined simulator. The SCALE VLIW ISA is used for the SCALE VLIW simulator. The SCALE functional simulator is the fastest simulator and is just used to check if the simulation provides correct results for the generated code. Functional simulation provides any cycle independent statistics, such as the number of instructions executed and memory references performed, as well as any other statistics related with the dynamic instruction stream. The SCALE pipelined simulator provides a five stage integer pipeline, which includes the stages IF (Instruction Fetch), ID (Instruction Decode), RF (Register Fetch), EX (EXecution) or MEM (MEMory access), and WB (Write Back). Note the MEM stage is performed in the same cycle as the EX stage as each load and store does not have a displacement for the base register and hence does not require the calculation of an effective address. A SCALE assembly file can be simulated by both the SCALE functional simulator and SCALE pipelined simulator with no change in how the file is produced. The SCALE VLIW simulator uses individual SCALE base instructions, but these instructions are placed in groups that we refer to as VLIW packs. If a useful instruction cannot be placed in a given lane (position) within a VLIW pack,

then a *nop* instruction must be placed in that lane. Instructions within a VLIW pack are fetched, decoded, and executed together. If any instruction within a VLIW pack stalls, then all instructions in the VLIW pack are stalled Code generated for the VLIW machine can also be simulated by the functional simulator. This is because the branch instructions within a VLIW pack implement *first taken* semantics. Likewise, an instruction that has an antidependence with a previous instruction is never scheduled before the previous instruction in a VLIW pack. A group of instructions in a pack can be executed either simultaneously or one at a time in left to right order, as would be the case with functional simulation.

1.1.3 Compilation System

The compilation support needed for the SCALE project has to be able to compile SPEC benchmarks and support low-level code generation and code-improving transformations. We decided to use a conventional compiler to generate code and perform translation and a variety of code-improving transformations at the assembly level. Figure 1.1 shows how we generate code for the various SCALE ISAs. We use *gcc* to produce conventional MIPS assembly files. This allows us to compile files in a variety of source languages, such as C, C++, and Fortran, and also leverage the optimizations that are provided by the *gcc* compiler. We developed our own assembly optimizer, called *asopt*, that takes an assembly file as input, translates instructions when necessary to a new instruction set, performs a variety of analyses and code-improving transformations, and produces modified assembly code as output. In order to properly determine which registers are live at any given point in a function, we need to know which registers are being passed to each function that is being called and which register if any is used to return a value. Rather than attempting to perform interprocedural analysis to determine this information, we instead gather information as a side effect of the *gcc* compilation. We use an option in *gcc* to produce a *.gkd* file that contains information about each *gcc* RTL (instruction), which we use to determine which registers are passed as values in function calls. We also generate a MIPS object file with symbolic debugging information from which we generate an *.objdump* file, which contains information about the function return type that is used to determine in which register a return value is placed. Both the *.gkd* and *.objdump* files were input to our *geninf* tool that produced a *.inf* file that could be easily parsed by *asopt*. We use this approach as *asopt* also needs to process some hand-written assembly files in the libraries we utilized. In these cases we generate a corresponding *.inf* file by hand for each hand-written assembly file. The assembly optimizer, *asopt*, reads both the *.inf* file and the MIPS assembly file

to produce the new SCALE assembly file. Various flags can be passed to *asopt* to both select the code-improving transformations to be performed and to select the ISA for the assembly target file.

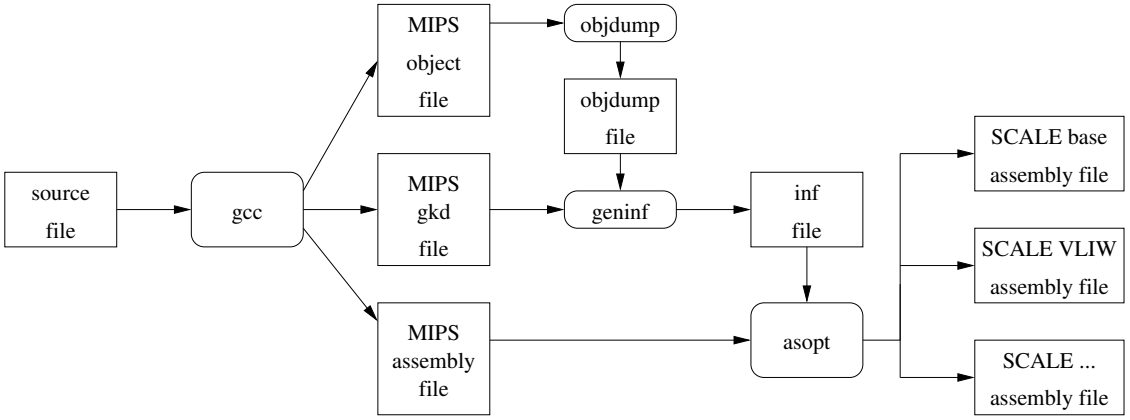


Figure 1.1: SCALE Code Generation Process

For each function, *asopt* reads in the instructions, identifies the type of each instruction and which registers are set and used, and builds the control flow graph for the function. It then translates each MIPS assembly instruction to its corresponding SCALE assembly instruction when the input MIPS instruction is not in the SCALE instruction set that is described in Section 1.1.1. It also expands any pseudo instructions so that each SCALE assembly instruction has a one-to-one mapping with a SCALE machine instruction. This step is necessary when performing some low-level code-improving transformations. For instance, scheduling instructions into VLIW packs requires packaging a specified number of machine instructions together. *asopt* then performs a number of code-improving transformations. For instance, it unrolls innermost loops when possible. We decided to implement loop unrolling in *asopt* to have more precise control over the heuristics when deciding whether or not to unroll a loop and the loop unroll factor to use when performing loop unrolling. *asopt* also applies a variety of transformations to eliminate true dependences, such as accumulator expansion and partitioning the increments of a basic induction variable to use separate registers. The assembly optimizer also applies other optimizations, such as common subexpression elimination and loop-invariant code motion. The *gcc* compiler produces many pseudo instructions that the assembly optimizer expands into multiple assembly instructions that have a one-to-one correspondence with machine instructions. After pseudo instruction expansion, there are often opportunities for these optimizations to be applied. When generating code for the SCALE VLIW

ISA, *asopt* schedules instructions into VLIW packs by performing register renaming and scheduling instructions both within and across basic block boundaries.

1.2 Testing the Assembly Optimizer

An error originating from the *asopt* assembly optimizer may become apparent as either an error from the execution of *asopt* itself, or an error in the simulation of *asopt*-produced code. We utilize multiple benchmark suites to test the assembly optimizer, which include SPEC '95 and SPEC '06. Despite their age, the SPEC '95 programs execute a small number of instructions, but have a similar control flow structure to that of SPEC '06. An example of their similarity exists with 126.gcc from '95 and 403.gcc from '06. Therefore, they are still quite valuable for verification. In our experience, once SPEC '95 code runs successfully, very few additional problems are discovered with the similar SPEC '06 program.

Ensuring the *asopt* executable could successfully generate code came first in the testing process, and was performed with both SPEC benchmark suites. The transformed code of the benchmarks then needs to be simulated to check execution accuracy; however, a baseline was needed to assess simulation correctness before this could begin. From both suites, each SPEC benchmark was simulated using its unaltered, initial MIPS assembly files using a MIPS simulator. If simulation was successful, the output is known to be accurate. These correct outputs now act as reference files for comparison against the simulation output of SCALE assembly optimized programs.

After creating reference files, *asopt* simulation testing began with code generated by the assembly optimizer, but with applying no optimizations. *asopt*'s pseudo-expansion code-improving transformation was then applied and simulated. Once all benchmarks transformed by pseudo-expansion produced the same simulation results as their respective reference files, this transformation type was considered correct and another transformation type could be tested in the same manner. An important aspect of the *asopt* testing process is that the correctness of one type of code-improving transformation can be tested before testing another transformation type.

CHAPTER 2

TRANSFORMATIONS

A code-improving transformation consists of a sequence of changes where the semantic behavior of the code in a function should remain the same, although its performance may be enhanced. A code-improving transformation can also be viewed as optional, as compared to a required transformation that is needed for correct execution. An optimization phase consists of the application of zero or more code-improving transformations of the same type.

An example of a required (non-optional) transformation is saving and restoring new callee-save registers which are allocated during a code-improving transformation. A transformation may be optional/non-optional depending on which simulation machine is being used. For instance, the standard VLIW simulator requires pseudo expansion be applied to all files and libraries of the simulated program. The MIPS assembly code that is being input to *asopt* contains pseudo instructions, in which one pseudo instruction may generate two or more machine instructions. The assembly optimizer "expands" these pseudo instructions so that there is a one-to-one correspondence between SCALE assembly instructions and SCALE machine instructions. When VLIW block scheduling is not being applied, pseudo expansion is an optional, code-improving transformation whose application gives *asopt* the opportunity for additional code-improving transformations. However, due to the fact that VLIW packs must contain a specific number of machine instructions, pseudo expansion is a non-optional transformation when *asopt* applies VLIW block scheduling.

2.1 Code-Improving Transformation Examples

In this section, I present some examples of code-improving transformations implemented in *asopt*.

2.1.1 Simple Transformations

Figure 2.1 displays a variety of transformations which have been sequentially applied to one section of assembly code. The original version of the code is represented by step *a* of this figure, and steps *b*-*h* display the first transformation through the seventh transformation applied by *asopt*.

We can see in step *a* that the data at the address of *g_qCount* is then loaded into register 2, and is incremented before being stored back to *g_qCount*.

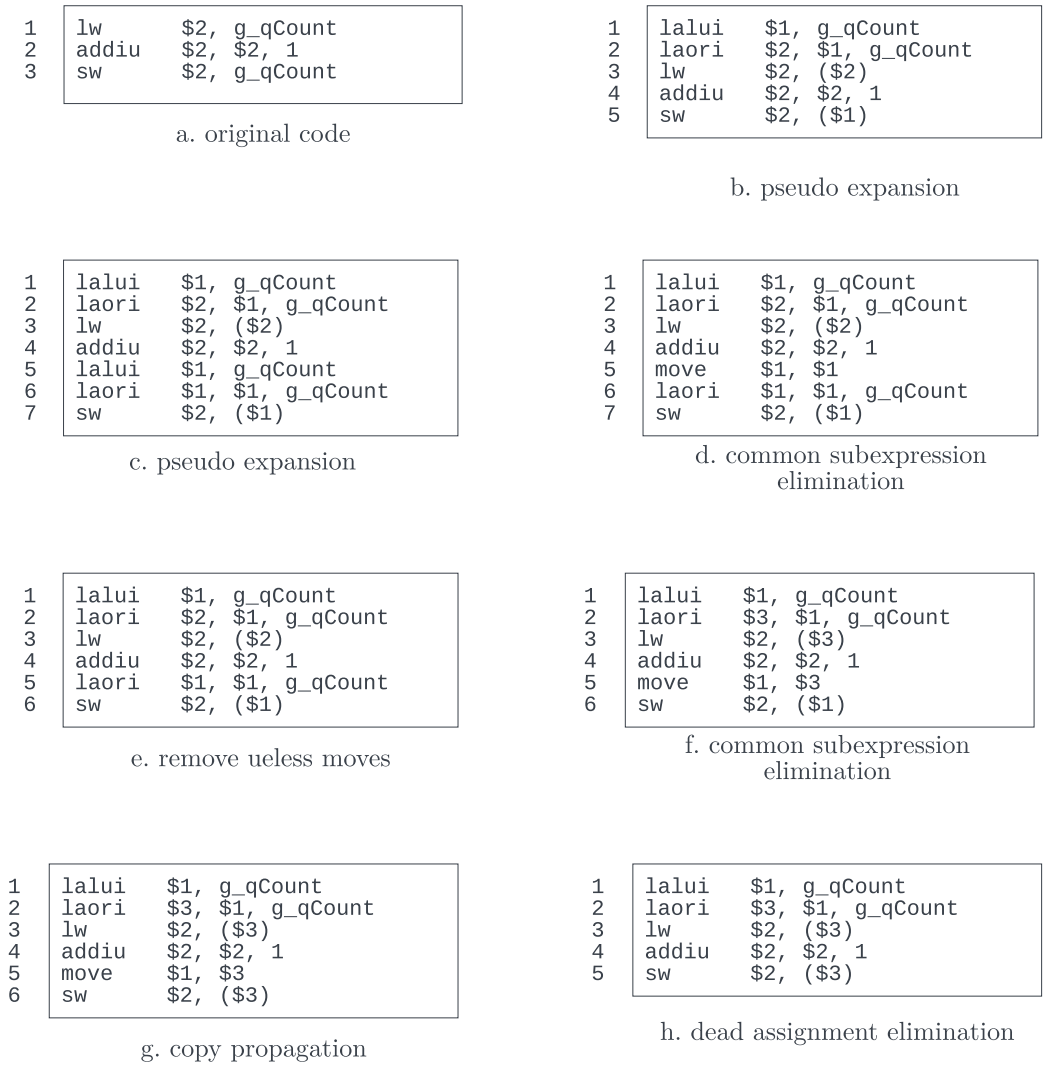


Figure 2.1: Sequential Applications of Various Code-Improving Transformations

A pseudo expansion transformation is the first transformation applied, which turns line 1 of step *a* into lines 1, 2, and 3 of step *b*. Similarly, step *c* shows the result of a second pseudo expansion transformation performed on line 5 of step *b*. These two pseudo expansion transformations do not change the semantics of the code, as they are each performing their respective load or store of the *g_qCount* address with three instructions rather than one. Intuitively, we know that the address of

g_qCount does not need to be loaded twice in step *c*. Instead, we only require one *lalui*, *laori* pair to initially load the *g_qCount* address. The remaining series of transformations will work together to remove this redundancy.

Step *d* of Figure 2.1 displays the result of a common subexpression elimination transformation, which transforms the *lalui* instruction on line 5 of step *c* into a *move* instruction on line 5 of step *d*. If there are two instructions in which the second calculates the same value as the first, common subexpression elimination will replace the second instruction with an instruction which moves the destination register of the first instruction into the destination register of the second instruction (if this transformation will not alter the semantics of the code). In step *c*, the upper 16 bits of the *g_qCount* address are being assigned twice, so the destination register of the first *lalui* may be moved into the destination register of the second *lalui*.

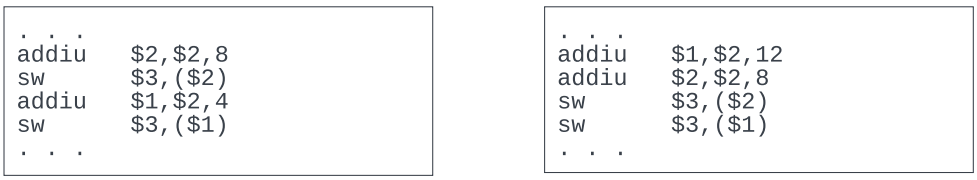
Note that in this example, both destination registers are register 1, so the "move $1,$1" instruction in step *d* is formed. This creates an opportunity for the fourth *asopt*-applied transformation, a "remove useless moves" transformation, to eliminate line 5 of step *d*. The result of the fourth transformation is shown in step *e*. Common subexpression elimination and remove useless moves are code-improving transformations that rid the assembly of redundant or useless computations without changing semantics.

Step *f* of Figure 2.1 shows the result of another common subexpression elimination transformation, which replaces the *laori* instruction on line 5 of step *e* with the *move* instruction on line 5 of step *f*. This common subexpression elimination requires the allocation of an unused register, register 3, because register 2 had been reset in line 3 of step *e*.

The transformation shown in step *g* is a copy propagation transformation that changes the base register of the *sw* command in line 6 of step *f* from register 1 to register 3. Copy propagation occurs when a copy instruction—the *move* instruction in this example—sets register "R1" equal to "R2" and the subsequent uses of R1 may be feasibly changed to R2, given that neither R1 nor R2 are reset in the time that R1 is alive. This is not a required transformation, but may result in less code if it results in a dead assignment.

Step *g* is an example of a copy propagation transformation which allows for a dead assignment elimination—the seventh transformation whose application is shown in step *h*. Copy propagation has made line 5 of step *g* a dead assignment, an instruction in which the destination register is set, but never subsequently used. Dead assignment elimination is a code-improving transformation which removes these useless instructions, decreasing code size.

Figure 2.2 shows a single "remove increment true dependencies" transformation. Part *a* shows the assembly code before the transformation and part *b* shows the result of the transformation. When comparing the two parts, we can see that the second *addiu* instruction from *a* has been hoisted up before the first *addiu* instruction from *a*. The immediate value of the moved instruction has also been changed from a 4 to a 12. In part *a*, we can see that a true dependency exists between the first and second *addiu* instruction because of register 2. The first *addiu* is an increment which sets and uses the same instruction while using an immediate. Therefore, the location of the second *addiu* may be moved so long as its immediate value is changed accordingly and the semantics of the code remains the same. Moving the second *addiu* above the first removes the true dependency between them, and as a result of this type of transformation, *asopt*'s opportunities for scheduling instructions are increased due to fewer restrictive true dependencies.



a. original code

b. remove increment true dependencies

Figure 2.2: Remove Increment True Dependencies Transformation Example

2.1.2 Loop Invariant Code Motion

Figure 2.3 shows the sequential application of transformations on a loop. The original code is shown in step *a*, where some block within the loop is loading the address of *iDist* and accessing the data at that address. Loop invariant code motion is a transformation which hoists loop invariant code into the loop preheader to decrease the number of transformations executed in the loop body. Loop invariant code is code that may be moved outside of the loop body without changing the semantics of the code.

In this example, the upper 16 bits of the *iDist* address do not need to be loaded on each iteration of the loop; instead, this value can be calculated in the loop preheader and referred to within the loop body. Figure 2.3 shows how this change can be achieved with loop invariant code motion, and is further improved by subsequent transformations. Step *b* shows the result of a single loop invariant code motion transformation on the *lalui* instruction in step *a*. The *lalui* is hoisted to the

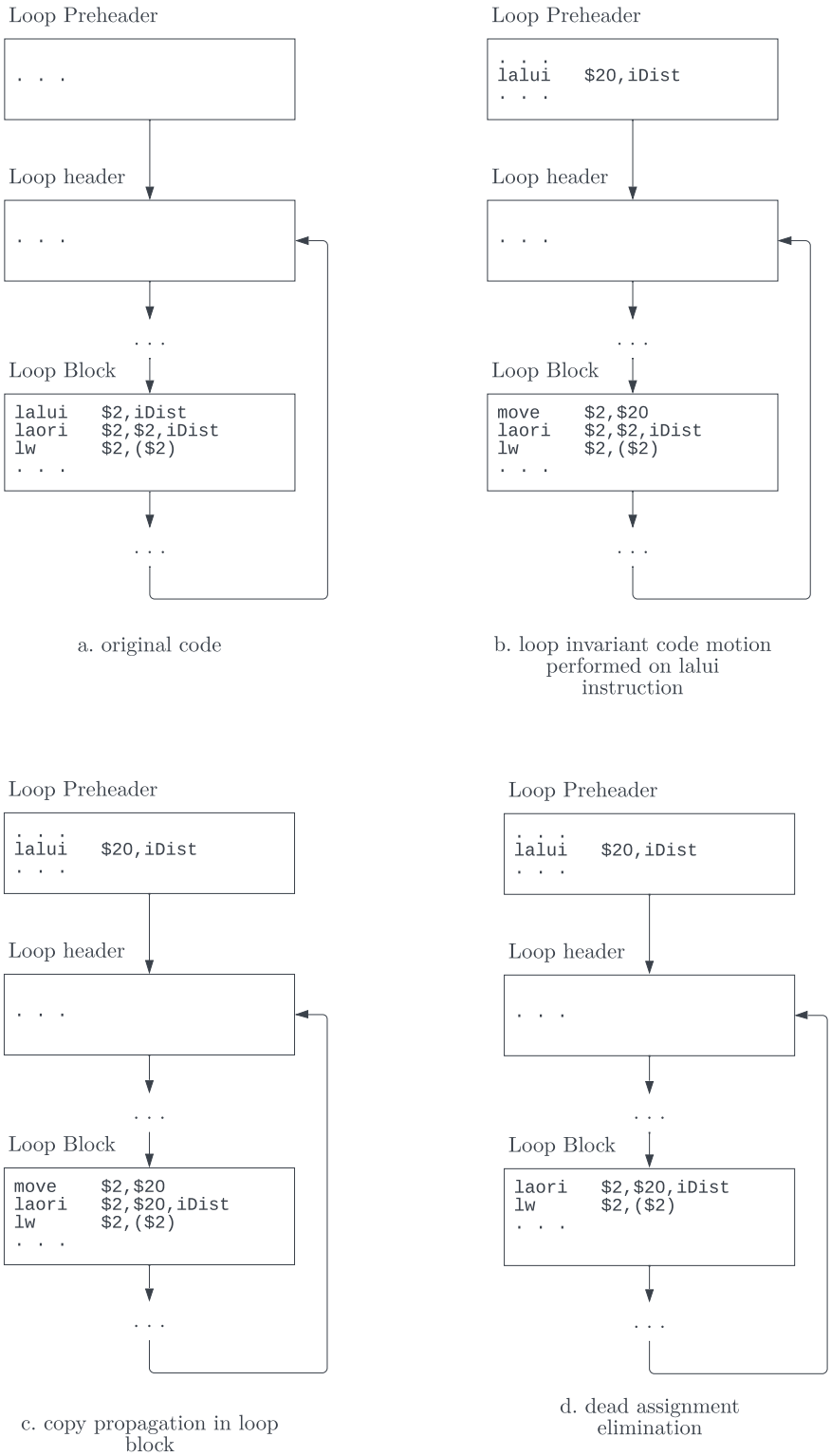


Figure 2.3: Loop Invariant Code Motion with Other Transformations

preheader, where it is stored in an available, new register—register 20 in this example. A *move* instruction replaces the original *lalui* to access the upper 16 bits of *iDist* within the loop.

In step *c*, a copy propagation transformation in the loop body then is able to propagate register 20 into the *laori* instruction. The transformation in step *c* turns the *move* into a dead assignment, as register 2 is dead because it is set in the following *laori* instruction. A dead assignment elimination transformation then removes the *move* instruction, and the resulting code is shown in step *d*. Not shown in Figure 2.3, this same applications of steps *a*, *b*, and *c* will then be applied to the *laori* instruction. This will result in all 32 bits of the address being loaded in the preheader, and accessed in the loop with register 20.

2.1.3 Loop Unrolling and Accumulator Expansion

Figure 2.4 displays a source code level example of the loop unrolling and accumulator expansion optimizations. Part *a* of this figure shows the original code, while *b* displays the code after loop unrolling has been applied with an unroll factor of 2. Loop unrolling is an optimization in which the loop overhead and number of loop iterations is decreased by performing the work of multiple iterations in one. The number of original iterations that are being executed within a single iteration of an unrolled loop is called the loop unroll factor. Although increasing code size, the repetition of the instructions within the unrolled loop body presents opportunities for parallelization.

Accumulator expansion is an optimization that may sometimes be performed after loop unrolling has also been applied. Figure 2.4 part *c* shows code that has had both optimizations applied. An accumulator is defined as a variable which, for each instruction that uses said variable in the loop, the variable is both set and used and is operated on by a commutative operation. The accumulator in this figure is the *sum* variable. When a loop is unrolled, a single accumulator can be formed into multiple accumulators, like how *sum* is broken up into *sum* and *sum2* in part *c* of Figure 2.4.

During accumulator expansion, the number of new accumulators which result from one original accumulator will be the loop unroll factor - 1. A loop unroll factor of 4 will create 3 additional accumulators if there are enough available registers. All newly-formed accumulators which have stemmed from an original accumulator will then be joined together after the loop to rebuild the original accumulators. Accumulator expansion breaks up the original accumulators so that each new accumulator may become independent from the original, breaking true dependencies and increasing instruction scheduling flexibility and opportunities for parallelization.

```
# arr is an array with 100 randomly-initialized elements
sum = 0;

for(i = 0; i < 100; i++){
   sum = sum + arr[i];
}
```

a. Loop before Loop Unrolling and Accumulator Expansion

```
# arr is an array with 100 randomly-initialized elements
sum = 0;

for(i = 0; i < 100; i = i+2){
   sum = sum + arr[i];
   sum = sum + arr[i+1];
}
```

b. Loop after Loop Unrolling With Unroll Factor of 2
before Accumulator Expansion

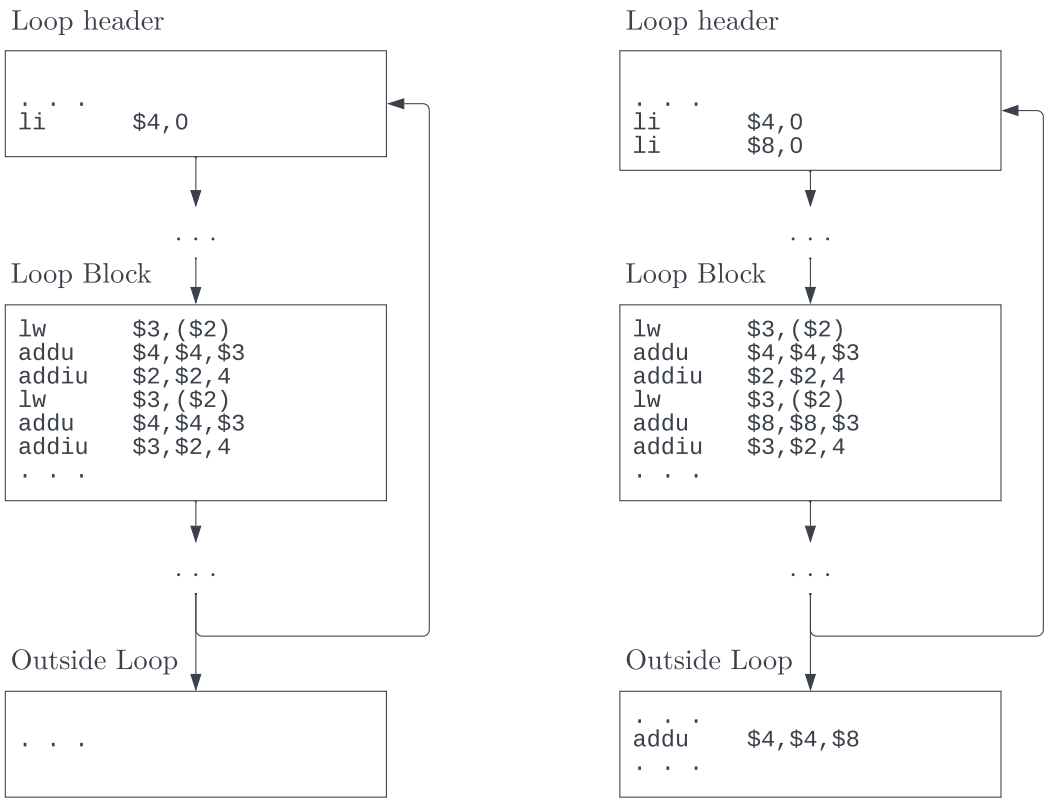
```
# arr is an array with 100 randomly-initialized elements
sum = 0; sum2 = 0;

for(i = 0; i < 100; i = i+2){
   sum = sum + arr[i];
   sum2 = sum2 + arr[i+1];
}
sum = sum + sum2;
```

c. Loop after Loop Unrolling With Unroll Factor of 2
after Accumulator Expansion

Figure 2.4: Loop Unrolling and Accumulator Expansion

Figure 2.5 shows the application of one accumulator expansion transformation within an unrolled loop. In this example, the loop has been unrolled by a factor of two, as each instruction in part *a* is repeated once. At the assembly level, an accumulator is a register. Register 4 is an accumulator in the unrolled loop because each instruction in the loop which uses register 4 has

the same commutative operation and also sets the register. In part *b* of Figure 2.5, accumulator expansion has been applied, assigning a new destination register to the second instruction that uses the accumulator. This effectively splits the original accumulator, register 4, into two separate accumulators that are merged back together with their respective commutative operation once outside the loop.



a. unrolled loop

b. accumulator expansion

Figure 2.5: An Accumulator Expansion Transformation

2.2 Transformation Counting

A transformation range is a set of code-improving transformations sequentially applied by *asopt* to either a program or assembly file. For instance, the first 50 transformations applied by the assembly optimizer to a program is a transformation range. A transformation count is a number assigned to a specific transformation in order to identify it.

When the simulation of *asopt*-produced code produces incorrect output, we must isolate which code-improving transformation is the cause of error. The *asoptiso* tool finds the **first** code-improving transformation causing the simulator to produce incorrect output for a given program. *asoptiso* was implemented as a C program that performs *system()* calls that allow it to perform unix shell commands, which include invoking *asopt*, the assembler, the linker, and the simulator. *asoptiso* performs a binary search on the applied transformations, using simulation to check for correctness of various transformation ranges until the range is narrowed down to a single transformation. Note that the terms transformation range and binary search range are synonymous in the context of error isolation with *asoptiso*.

asopt and *asoptiso* have different responsibilities relating to the transformations applied to a program. The assembly optimizer only works with one assembly file at a time, therefore it is responsible for tracking and controlling the assembly of transformations applied to a single assembly file within a program. However, the assembly optimizer error isolator is responsible for tracking and controlling the number of transformations applied to all assembly files within a program. *asoptiso* is able to interact with all transformations by repeatedly running *asopt* on each assembly file within a program, and storing the transformation data produced by *asopt* during its execution.

2.2.1 Transformation Counting in *asoptiso*

In order to perform a binary search on all transformations of a program, the isolator needs access to file-offset and program-offset transformation counts. Figure 2.6 displays the distinction between these two categories. In this diagram, multiple transformations of type "V" and type "G" are being executed on two assembly files which, in this case, comprise the entire program. *asopt* applies the following to Assembly File 1 in this order: one transformation of type V, one transformation of type G, and one transformation of type V. Assembly File 2 has five transformations of its own, executed in the order of V, G, G, V, V. These eight individual transformations act as columns in Figure 2.6, where the rows represented by sections *a-f* show each column's transformation count calculated in a different way. A single transformation is not used in calculating the transformation counts for the given row if its slot is blank and it is below a dotted, rather than solid, black line.

Section *a* of Figure 2.6 represents generic file-offset counting, or file-offset counting, which numbers a transformation by the order in which it was performed by *asopt*, relative to the assembly file in which the transformation is applied. The file-offset count of the second V transformation

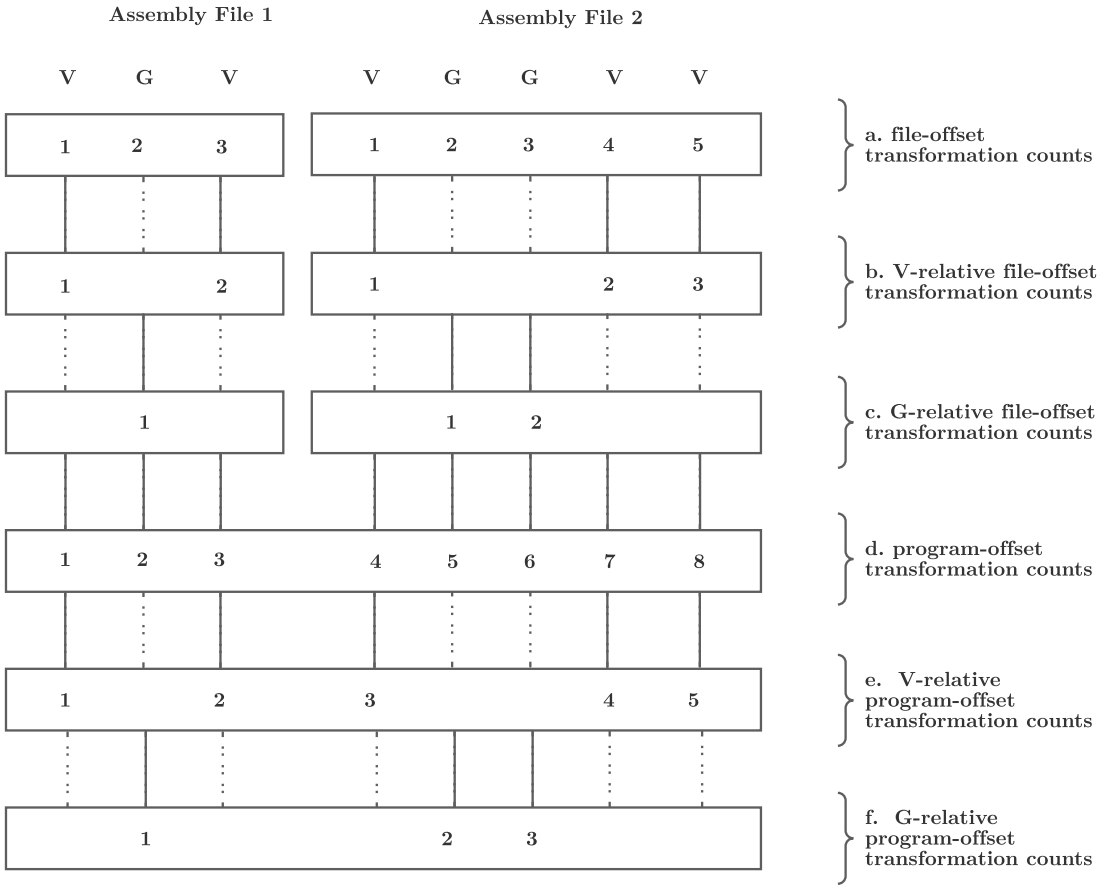


Figure 2.6: File-Offset vs. Program-Offset Transformation Counts

in Assembly File 2 is 4. Generic file-offset transformation counting does not distinguish between different transformation types, unlike specified file-offset transformation counting.

A specified file-offset transformation count is more specifically called an X-relative file-offset transformation count, where X is the flag representing the transformation type to which the file-offset counting is relative. Figure 2.6.*b* shows the V-relative file-offset counts, while *c* shows G-relative file-offset counts. The V-relative file-offset count for the second V transformation in Assembly File 2 is 2 because it is the second transformation among only the V transformations in that assembly file.

Section *d* of Figure 2.6 shows generic program-offset counting, or program-offset counting, which orders transformations relative to the entire program. The second V of Assembly File 2 has a program-offset count of 7. We also track specified program-offset counts which are called X-

relative program-offset transformation counts, where X is the flag representing the transformation type to which the program-offset counting is relative. Section *e* displays V-relative program-offset counts, and Section *f* shows the G-relative program-offset counts. The second V of Assembly File 2 is the fourth V type transformation of all V type transformations within the program, so it has a V-relative program-offset transformation count of 4.

A single transformation of type X will have four transformation counts associated with it: file-offset, program-offset, X-relative file-offset, and X-relative program-offset. *asoptiso* will calculate generic and specified program-offset transformation counts, but will use all types of transformation counts during its execution. Due to the fact that *asopt* operates on one assembly file at a time, it is is responsible for calculating and using both generic and specified file-offset transformation counts.

2.2.2 Transformation Counting in *asopt*

We have implemented the ability to control and track the amount of transformations being performed by *asopt*. Command line flags passed to *asopt* control which types of transformations will be applied. For example, passing the V flag will tell *asopt* to apply the "VLIW block scheduling" optimization. There are also some optimizations, called "minor optimizations", which will always be applied each time *asopt* is run, even if they are not specified with a flag.

A user is able to specify a maximum limit to the number of code-improving transformations applied by *asopt* to the given assembly file. A number immediately following a command line flag X will implement a maximum X-relative file-offset transformation count, where all following transformations of any type are not applied. There can be one maximum limit per flag, for any number of flags; however, only the limit that is reached first in sequential transformation application order will matter because all transformations are stopped after a limit is reached. For example, if we run *asopt* with command line argument "VG2" on Assembly File 2 of Figure 2.6, all transformations will stop after applying the transformation with a G-relative file-offset count of 2. This specific transformation has a generic file-offset count of 3; therefore, all transformations with a generic file-offset count of 4 or greater will not be applied. Specifying flag Y will represent all types of transformations being applied, rather than a specific type. If, for Assembly File 2 of Figure 2.6, we pass "VGY4" this will implement a maximum generic file-offset transformation count of 4. *asopt* will apply only the first 4 transformations of any type to Assembly File 2.

After the maximum limit transformation is applied, all required transformations will continue to be applied by the assembly optimizer even though all code-improving (optional) transformations

will cease. For example, let's say that a maximum limit on a code-improving transformation type has been implemented on SCALE VLIW code. The SCALE VLIW simulator requires instructions to be scheduled into VLIW packs, so after a maximum limit has been reached, the code within the remaining basic blocks that have not yet been scheduled by *asopt* must be placed such that there is one instruction per pack.

Having control over the type and number of transformations applied in *asopt* is critical for implementing the use of the *isolation flag* and *non-isolation flags* in *asoptiso*. The isolation flag is the *asopt* command line flag which engages transformations of a specific type, where the first erroneous transformation is suspected to be of this type. The non-isolation flags are an optional string of command line flags used in conjunction with the isolation flag. Note, only the isolation flag transformations will be isolated, although the non-isolation flag transformations will also be used to optimize the program if they are specified.

Let's say we are isolating on the program represented by Figure 2.6, where G is the isolation flag and V is the non-isolation flag. *asoptiso* will run a binary search to find the first erroneous transformation of type G. Therefore, the binary search ranges will be transformation ranges from the G-relative program-offset transformation counts because we are trying to isolate the first erroneous G transformation within the program, among all G transformations within the program. For this example, we want to know which of the three transformations represented in section *f* of Figure 2.6 is the first transformation causing the simulation to produce incorrect output. The V transformations being performed within the limits of the binary search range will still be applied.

If a transformation type suspected to contain the first erroneous transformation is not known, the isolation flag may be set to Y to represent all applied transformations and all transformations will be isolated. If we want to find the first erroneous transformation in the program of Figure 2.6, which could be of type V or G, we would set "Y" as the isolation flag and "VG" as the non-isolation flags. This would then result in the binary search ranges to be transformations ranges from the generic program-offset transformation counts in section *d* of Figure 2.6.

Two functions within *asopt*, *optimize()* and *incropt()*, track and control the application of file-offset transformations. *asopt*'s *optimize()* function is represented in Figure 2.7, and is responsible for the application of optimization phases on a function within the assembly file input to *asopt*. In *optimize()*, each optimization is applied to the function within the assembly code by a call to other functions within *asopt* itself. In the code snippet of *asopt* shown in Figure 2.8, we can see

that *optimize()* is applied to each function within an assembly file before the optimized assembly for that function is dumped to an output file.

```
1  moreopts = TRUE;
2
3  optimize()
4  {
5     setjmp(...);
6     if(moreopts){
7        // apply optimizations
8        . . .
9     }
10
11    // apply any remaining required transformations
12    . . .
13 }
```

Figure 2.7: Transformation Counting in *asopt* : *optimize()*

```
1  // process each function in the file
2  while(readinfunc())
3  {
4
5     // perform code-improving transformations on the function
6     optimize();
7
8     // dump out the assembly code
9     dumpfunc();
10
11    // free up the function's dynamically allocated structures
12    free_func_structs();
13 }
```

Figure 2.8: Transformation Counting in *asopt* : calling *optimize()*

Figure 2.7 shows that before *optimize()* is called, a global boolean called *moreopts* is set to true. This boolean is a flag that indicates whether or not *asopt* should continue to apply code-improving transformations to the current function that *asopt* is processing. When we reach a maximum limit, and code-improving transformations can no longer be applied to the assembly file, the control will be returned to the *setjmp()* within *optimize()*. At this point, *moreopts* will be set to false and *asopt* will proceed with applying required transformations to the rest of the function within the assembly

file. Once a maximum limit has been reached, it will stay in effect for the duration of the assembly file. Code-improving transformations will continue to not be applied in subsequent files.

A key component to implementing maximum limits is being able to identify the start of a code-improving transformation and then choose whether or not to perform it. We implemented a function named *incropt()*–short for "incrementing optimizations"–in the assembly optimizer which checks if a transformation should be performed. Figure 2.9 shows the code for the *incropt()* function. *incropt()* is called immediately before *asopt* reaches the point after the analysis determines that a code-improving transformation can be applied. This function takes in the argument *opt*, which represents the transformation type of the transformation about to be applied.

The code shown in Figure 2.9 can be interpreted in the following way: if this transformation of type *opt* which is about to be applied will exceed a maximum limit, set *moreopts* to false and use a *longjmp()* to return to the associated *setjmp()* in *optimize()* in order to stop the further application of code-improving transformations. If this transformation will not exceed a maximum limit, adjust the *totopts* structure accordingly. This combination of calling *setjmp()* and *longjmp()* allows us to stop applying code-improving transformations at any point in the binary search when isolating assembly optimizer errors.

The *totopts* array has an entry for each type of code-improving transformation. Each entry contains the fields *max* and *count*. The maximum number of applied transformations of the given type that *asopt* allows in the assembly file is represented by the *max* field. The *count* field contains the number of times a transformation of the given type has been applied in the assembly file. The *max* field holds a maximum limit. There may exist a maximum limit associated with each type of transformation in *totopts*, but after the first maximum limit is reached by *asopt*, all subsequent code-improving transformations will not be applied to the assembly file.

We also implemented a second type of maximum which limits the number of all types of transformations within an assembly file. As seen in line 10 of Figure 2.9, a valid optimization type is *ALL_OPTS*, which allows *totopts* to store *max* and *count* information for transformations of all types. Line 9 increases the *count* that is specific to the transformation's own type, and line 10 increments the *count* of all transformations. The given transformation will not be applied if the execution of this transformation would exceed either type of maximum limit, which is checked at lines 3 and 4 of Figure 2.9.

The transformation information in *totopts* will be output as a file called "trans_count.txt" once *asopt* has finished processing an assembly file. "trans_count.txt" contains the number of

```
1  incropt(opt)
2  {
3     if(totopts[opt].count == totopts[opt].max ||
4        totopts[ALL_OPTS].count == totopts[ALL_OPTS].max){
5        moreopts = FALSE;
6        longjmp(...);
7     }
8
9     totopts[opt].count++;
10    totopts[ALL_OPTS].count++;
11 }
```

Figure 2.9: Transformation Counting in *asopt* : *incropt()*

transformations applied per transformation type, the number of transformations applied in total regardless of type, and the number of transformations applied per function regardless of type. "trans_count.txt" will also contain the name of the assembly file to which these transformations were applied. "trans_count.txt" supplies critical file-offset transformation count data to the error isolator.

CHAPTER 3

THE ERROR ISOLATOR

3.1 Error Isolation Process

asoptiso performs a binary search on all applied transformations of a specified type, or of any type, in order to find the first erroneous transformation being applied by *asopt* throughout a program. The *asoptiso* process consists of three general steps, the first of which is to calculate program-offset transformation count data. The second and third relate to the binary search: apply transformations to the program up to a specific transformation count and simulate the assembly optimized program.

The three main steps of *asoptiso* are shown at a high-level in Figure 3.1, where the first, second, and third steps are represented by the *a*, *b*, and *c* sections, respectively. Before these main steps begin, *asoptiso* reads in a configuration file which specifies information necessary for isolation, including which types of transformations should be applied to the given program via the isolation and non-isolation flags. After reading in the configuration file, but before step *a*, *asoptiso* performs two checks. The first makes sure that the given isolation flag is performing transformations in the program. The second then ensures that the simulation output is correct when applying only non-isolation flag transformations to the program.

3.1.1 Formation of Program-Offset Transformation Counts

Step *a* of Figure 3.1 depicts *asoptiso* creating the "total_trans_count.txt" file. To create this file, *asoptiso* runs the assembly optimizer on each assembly file in the program such that *asopt* is applying all transformations of the types specified by the isolation and non-isolation flags. After *asopt* processes an assembly file, the isolator will read in "trans_count.txt" to gain file-offset transformation count data. The information in "trans_count.txt" will be used to calculate and append program-offset transformation count data to "total_trans_count.txt".

For example, let's say that *asoptiso* is running on the program depicted by Figure 3.2, where the isolation flag is V and non-isolation flag is G. During step *a* of Figure 3.1, the isolator will use the V-relative file-offset transformation counts to form V-relative program-offset transformation counts. The isolator will also use the generic file-offset transformation counts to calculate and store

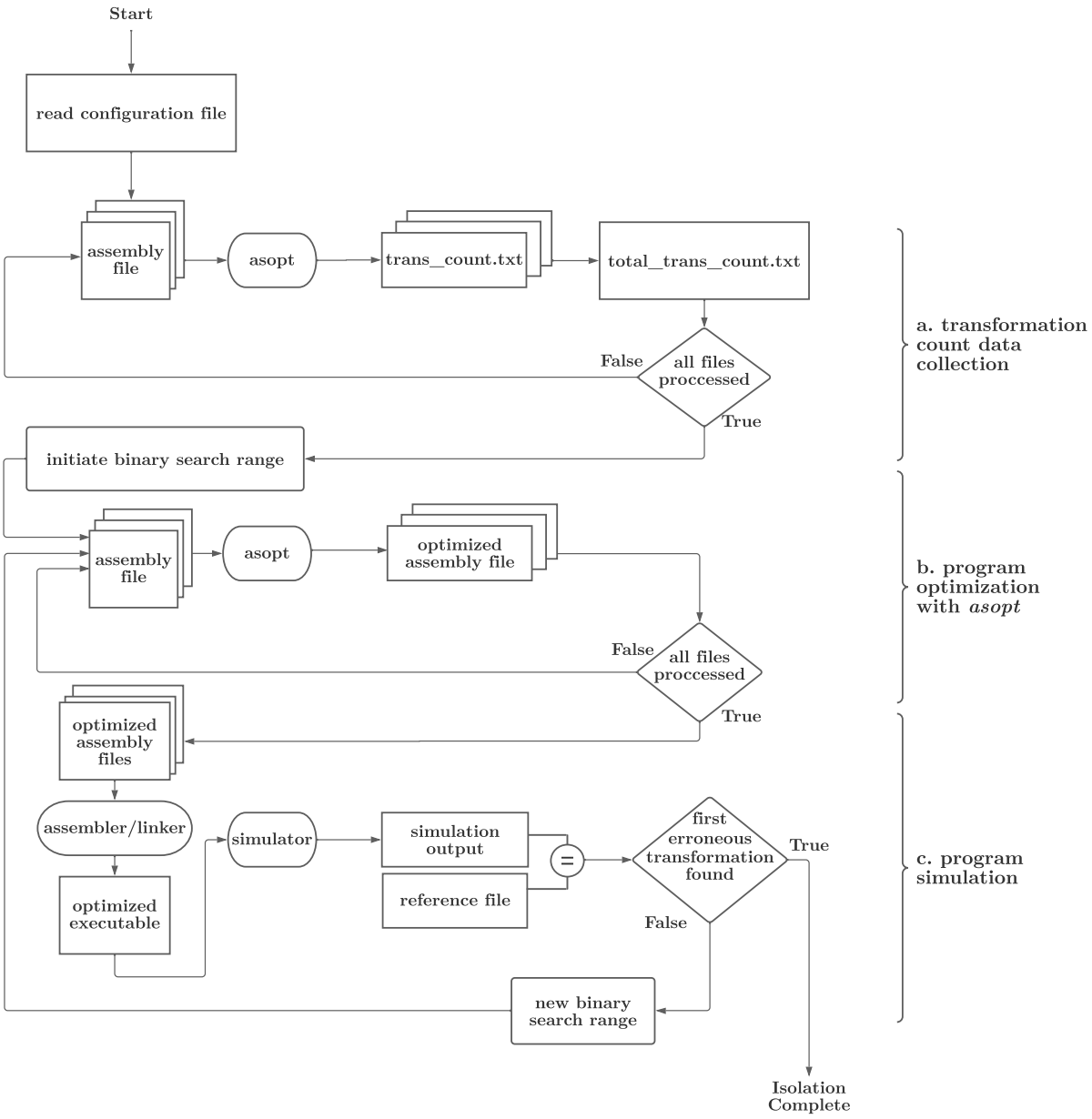


Figure 3.1: Optimization Error Isolation Process

the generic program-offset transformation counts. Due to the fact that the isolator will only isolate on transformations performed by the isolation flag, V-relative program-offset counts will be the only specified program-offset counts in "total_trans_count.txt". The "total_trans_count.txt" which would be formed after the completion of step *a* of Figure 3.1 is shown as Figure 3.3, where the ellipses represent program-offset counts relating to the assembly file's functions.

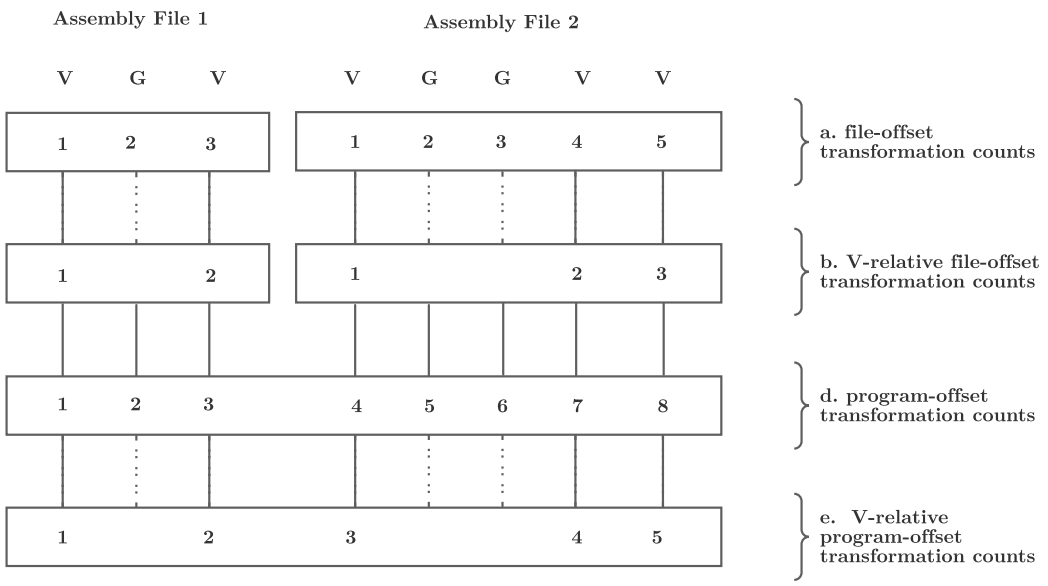


Figure 3.2: Transformation Counts with an Isolation Flag of V

```
Total transformations for Assembly File 1: 1 - 3
Transformation couns for V: 1 - 2
    . . .

Total transformations for Assembly File 2: 4 - 8
Transformation counts for V: 3 - 5
    . . .
```

Figure 3.3: total_trans_count.txt of Program Depicted in Figure 3.2

3.1.2 Binary Search

Step *b* of Figure 3.1 represents producing the optimized assembly of the program to be simulated. When transitioning from step *a* to step *b*, the binary search range is initialized to be the range of all applied transformations of the isolation flag type within the program. If a suspected transformation type is not known, the isolation flag may be set to Y and the binary search range will be initialized to the range of all transformations applied in the program. A binary search range "midpoint" is set to be the middlemost transformation within the binary search range. When transitioning from step *c* to step *b*, the midpoint will be chosen based on an updated binary search range, which itself has

been chosen based on the most recent simulation result. The midpoint is a specified program-offset transformation count to which transformations will be applied inclusive. The details of step *c* will be explained later in this section.

In order for *asoptiso* to apply transformations to the program based on the midpoint, it must first convert from a specified program-offset count to a specified file-offset count. For example, let's say that *asoptiso* is isolating on V transformations applied in the program represented by Figure 3.2. In this example, *asoptiso* has just transitioned from step *a* to step *b* of Figure 3.1. *asoptiso* has initialized the midpoint to the V-relative program-offset count of 3 since 5 V transformations are applied to the program. The isolator will then transfer the V-relative program-offset midpoint of 3 into the V-relative file-offset midpoint of 1—using the information in "total_trans_count.txt"—and store the name of the assembly file in which it is located. In this example, this file is Assembly File 2 of Figure 3.2, but we will refer it to more generally as the "midpoint assembly file".

As *asoptiso* is processing each assembly file as shown in step *b* of Figure 3.1, it will first check if it is the midpoint assembly file. The isolator will inform *asopt* to apply transformations without restriction to all assembly files prior to the midpoint assembly file. Once at the midpoint assembly file, the isolator will have *asopt* apply all transformations up to, and including, the specified file-offset midpoint. In the example using Figure 3.2, in which the file-offset midpoint is 1, *asoptiso* restricts the assembly optimizer to only applying the first transformation in Assembly File 2. The isolator then specifies that no code-improving transformations be applied to the following assembly files after the midpoint assembly file. In this example, neither G nor V transformations are applied to any assembly files after processing Assembly File 2.

3.1.3 Simulation

Step *c* in Figure 3.1 represents the simulation of the program optimized by step *b* of Figure 3.1. The optimized assembly files are input to the assembler and linker, which produces an optimized executable program. This executable is input to the simulator, and the simulation output will be compared against the program's reference output file to determine if the simulation was successful. A new binary search range is determined based on whether simulation output was correct or incorrect and the isolator returns to step *b* of Figure 3.1 if the new range is larger than one transformation. The isolation is complete when the binary search range has been narrowed down to one transformation, which is the first erroneous transformation.

3.2 Error Isolation Features

3.2.1 The Configuration File

The process depicted in Figure 3.1 requires flexibility from *asoptiso*. The isolator must work with multiple simulators, code transformation combinations, libraries, and user preferences. Information about these components is specified in a configuration file named "iso.config", an example of which is shown in Figure 3.4. This is the same configuration file which is read by *asoptiso* at the beginning of isolation, as can be seen in Figure 3.1. There must be an "iso.config" in the directory in which *asoptiso* is being run. Although this creates redundancies, it allows for necessary flexibility since a compiler writer may often need to be simultaneously isolating errors on multiple programs, where each may require a different isolation configuration.

```
1   machine: scale
2   non-isolation flags: VG
3   isolation flag: U
4   binary search run against golden (y/n): n
5   minimize diff of iso_result files (y/n): n
6   binary search start range: 30
7   binary search end range: 50
8   path to src files dir:
9   path to opt output dir:
10  makefile instruction to create sim obj/asm files: 'make build_only'
11  makefile instruction to run simulation: 'make test'
12  makefile instruction to run timed simulation: 'make test_timed'
13  use simulation timeout (y/n): n
14  timeout as failure (y/n):
15  timeout (minutes):
```

Figure 3.4: Optimization Error Isolation Configuration File

Line 1 of this figure shows which simulation machine is to be used with testing. Lines 2 and 3 let the user declare non-isolation flags and an isolation flag. In this configuration example file, the program that is being run through *asoptiso* will apply V, G, and U type transformations, but only the U transformations will be isolated.

Line 4 allows the user to choose if, in step *b* of Figure 3.1, the files that are processed following the midpoint assembly file are the original MIPS assembly files (the "golden" files) or *asopt*-produced assembly with only the non-isolation flags applied. If "n" is selected, files following the midpoint assembly file are *asopt*-produced. This feature is automatically turned off when the simulation machine is set to the SCALE VLIW simulator, because SCALE instructions must be scheduled in VLIW packs.

Lines 6 and 7 allow the user to isolate on a specific range of isolation flag transformations. If we know that the first erroneous U transformation lies somewhere between the 30th and 50th U transformations, we can begin the binary search with this range. Combined with the fact that *asoptiso* prints status updates as it executes, which includes the transformation range, this means the user may stop isolation and resume it at a later point. This feature is useful when the processor crashes during the error isolation process.

Lines 8 and 9 let the user clarify whether the first erroneous transformation is in the assembly files of a program or within a library being used by the program. Lines 13-15 are a feature to handle infinite loops or long runtimes. For instance, if some isolation flag transformation causes an infinite loop on the execution of an otherwise 2-minute simulating program, the user can choose to treat a 3-minute runtime as a simulation failure.

The configuration file provides a link between the process of isolation and checking simulation results. The execution of simulation, assemblage and linkage for simulation, and reference file comparison are the responsibilities of a program's makefile. The makefile will print a Unix "cmp" result after simulation, which is then stored and read by the isolator. The *asoptiso* configuration file takes the name of the relevant makefile rules in lines 10-12 of Figure 3.4 so that the specifics of simulation are abstracted from the isolator.

Certain transformations cannot be isolated when they are not optional. In the early stages of developing the error isolator, required transformations were being isolated when Y was set as the isolation flag. This kept *asoptiso* from finding the first erroneous transformation. The lack of a required transformation's application to a program after the binary search midpoint was reached caused each simulation of the program to produce incorrect output. This occurred when accidentally isolating the required (non-optional) transformation of saving and restoring new callee-save registers which are allocated during a code-improving transformation. When this problem with the error isolator occurred, it was known that the required transformations were not the cause of error, and therefore the problem was solved by not tracking the transformation counts of required transformations in *asopt*. There may be restrictions on transformations depending on which simulation machine is being used. For instance, the standard VLIW simulator requires pseudo expansion be applied to all files and libraries of the simulated program. This is because the VLIW simulator requires a 1-1 mapping between assembly instructions and machine instructions for proper VLIW pack alignment. When the standard VLIW machine is specified, pseudo expansion may never be isolated on, and in fact must always be applied.

3.2.2 Diff Minimization

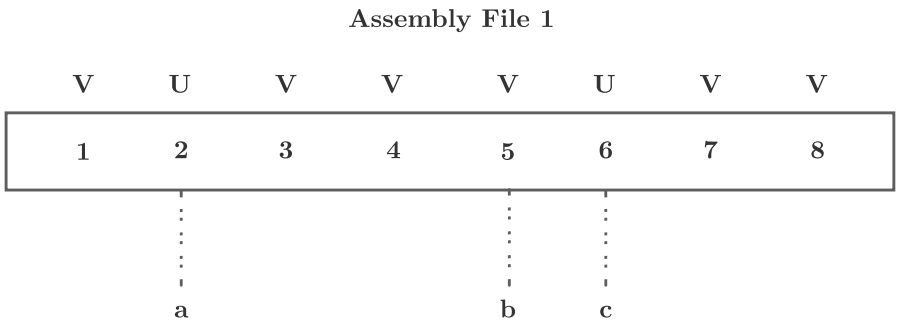


Figure 3.5: Optimization Error Isolation Diff Minimization

An additional action called *diff minimization* is performed to further enhance the error isolation. Figure 3.5 shows eight *asopt* transformations applied to Assembly File 1. Note that there are three transformations of interest labeled as *a*, *b*, and *c*, which are file-offset transformations 2, 5, and 6, respectively. Let's say that isolation is performed on a program with an isolation flag of U and non-isolation flag of V, so only U type transformations are being isolated. The result of *asoptiso* reveals that Figure 3.5's transformation *c* is the problem. This means that the U transformations up to, and including, transformation *a* are not causing the simulation error. To perform diff minimization, the isolator will simulate again, applying transformations up to transformation *b* inclusive. If this simulation is correct, we can decisively know that transformation *c* is the first erroneous transformation among all types. Otherwise, the first erroneous transformation is located after transformation *a* but before transformation *c*, and a second binary search is now performed in this new search range.

Diff minimization provides a check that the transformations of the isolation flag type actually contain the error—although this is likely the case. However, it is possible that the first erroneous transformation may not be of the isolation flag type due to interactions between flags that are not obvious to a compiler writer testing only whole flags at a time. Although rare, this scenario has occurred during the testing of the pseudo expansion transformation. A pseudo expansion transformation appeared to be causing incorrect simulation output, but rather it was the interaction

between a minor optimization and the pseudo expansion optimization. This caused the first erroneous transformation to be a minor optimization, despite the minor optimization not causing incorrect simulation when used alone.

Although we have not yet encountered this during testing, it is also possible that the first erroneous transformation may lie outside of the isolation flag transformation range due to interactions between transformations. This situation can be handled by running *asopt* with the Y flag, with a new binary search range that can be found by looking at the isolator terminal output. For example, if all binary search iterations produced correct simulator output, the user can then isolate on the range of all types of transformations performed after the generic program-wide transformation count equivalent to the last transformation performed by the fail flag.

Diff minimization serves to increase the precision of the isolator results. As shown in the example in Figure 3.5, a number of transformations may be occurring between the first erroneous isolation flag transformation and the last known correct isolation flag transformation. Diff minimization ensures that there is only a difference of one transformation, of any type, between the first found erroneous transformation. This results in a smaller difference between the assembly output files produced by *asoptiso*, allowing for easier error spotting by the compiler writer. This method provides the same scope as isolating on all transformation types, while typically resulting in a much more practical and faster way to narrow down the first erroneous transformation.

3.2.3 Isolator Output

Before the binary search begins, *asoptiso* prints a confirmation of all configuration data for the isolation which is about to occur. As it is executing, *asoptiso* prints status updates to the user. Among the information presented is the transformation range being tested and its associated midpoint assembly file, specified file-offset midpoint, and simulation result (success or failure). If the simulation run was a failure, it will print out the error message resulting from the execution of assembly optimized code.

asoptiso also provides the user a simple method to assess and debug results. Once the isolation is complete, the user receives three output files: the assembly file applying the first erroneous transformation, this same assembly file without the first erroneous transformation, and a simulation result data file. These two assembly files may be used with a diff tool to quickly spot errors since they will only differ by the one erroneous transformation. The third file lists relevant data about the isolation run, including the specified file-offset transformation count of the found first erroneous

transformation, which can be used for efficient debugging. For instance, if the first erroneous transformation is found to be the second U transformation of some assembly file, a conditional breakpoint can be placed at the *incropt()* call at this point. Because *incropt()* is called before a code-improving transformation is about to be applied, this breakpoint will stop immediately prior to the application of the second U transformation, and the compiler writer can determine why the transformation was erroneously applied.

The *asoptiso* tool may also help to find errors originating from the simulators themselves. This may occur if no problem is apparent in the diff of files produced by *asoptiso*. Additionally, a transformation found by the isolator to be erroneous may work for a first simulator but not a second one, revealing that the problem comes from the second simulator. We found that this scenario was often the case, and certain errors were revealed to be originating from the VLIW simulator when the same code would work with the functional simulator. This benefit is especially helpful when a new simulator is being introduced.

CHAPTER 4

RESULTS

asoptiso speeds up isolation by decreasing the number of simulations performed as much as possible. *asoptiso* takes in a isolation flag, which is useful since compiler writers often implement one new optimization at a time. If the flag combination "AB" has been successfully simulated, but "ABC" has not, we can assume the problem likely lies in the newest flag added. Knowing which transformation type on which to focus the error isolation provides a significant advantage: it decreases the initial range of transformations which may contain an error, resulting in fewer steps in the binary search and therefore minimizing the number of simulations required to isolate the first erroneous code-improving transformation. For any given program, there will be a relatively sizeable amount of transformations performed by *asopt*, and starting a binary search with a isolation flag could be the difference between isolating on 10,000 transformations of all types versus 100 transformations of a single type. When the programs being tested may take hours to simulate, even a small decrease in the number of simulations is highly beneficial.

We can calculate the number of required simulations in order for *asoptiso* to find the first erroneous transformation among all transformations of any type as $\lceil \log_2 n \rceil + 1$, where n is the total number of transformations performed in a program. The + 1 accounts for one simulation that occurs before step *a* of Figure 3.1 in order to check that the program simulates correctly when only non-isolation flags are applied. We can also approximate the number of required simulations when *asoptiso* is using an isolation flag as $\lceil \log_2 n_i \rceil + 2$. For this calculation, n_i is the number of transformations of the isolation flag's type applied in the program, and the + 2 is, in addition to the simulation check before step *a* of Figure 3.1, the simulation required by diff minimization to ensure the found transformation is the first erroneous transformation of all types.

Table 4.1 shows the number of transformations and the resulting required number of simulations for isolation on multiple benchmark programs from the SPEC 2006 integer benchmark suite. The column "Isolating on All Trans" displays the transformations and required simulations for isolating on transformations of any type. In this case, *asoptiso* was isolating on the V, G, F, and U type transformations. "Isolating on Unrolling Trans" displays transformations and required simulations for isolating on *asopt*'s newest code-improving transformation, loop unrolling. In this case, *asoptiso*

was using an isolation flag of U and non-isolation flags of V,G, and F. Both categories' "num trans" were obtained by counting the number of transformations applied to only the benchmark program itself, and not the libraries used by the program.

Table 4.1: Assembly Optimizer Error Isolation Results

Benchmark	Isolating on All Trans		Isolating on Unrolling Trans	
	num trans	num sims	num trans	num sims
bzip	6,574	14	23	7
gcc	339,921	20	243	10
gobmk	87,357	18	155	10
h264ref	59,502	17	232	10
hmmer	31,232	16	52	8
libquantum	3,790	13	8	5
mcf	1,216	12	1	2
perlbench	122,691	18	67	9
sjeng	13,570	15	32	7

There is a significant decrease in the number of required simulations for isolating on loop unrolling versus isolating on transformations of any type. This difference is insignificant for quick-running programs on the most simple simulator; however, it is highly significant in saving time for longer-running programs and will become increasingly relevant as compiler writers progress in the testing of different SCALE ISAs and the different simulators for those ISAs. For instance, the SCALE pipelined simulator is approximately four times slower than the SCALE functional simulator. Even minute differences in the number of simulations saved will become more important as the simulation machines used for testing become slower and more complex.

CHAPTER 5

RELATED WORK

There has been a significant amount of work to assist in the testing of compilers. This includes complex methods such as compiler verification and translation validation, in addition to more practical methods such as constructing test programs, determining whether the output of a compiler is correct or not, optimizing the testing process, and post-processing of test results [4]. Isolating the code-improving transformation that causes incorrect output and the place in the compiler where that transformation is applied falls in the last category.

Compiler verification is a method of compiler testing in which the goal is to rigorously prove the correctness of a compiler's output in advance. Compiler verification aims to prove that a compiler's output will always be correct, rather than verifying correctness of a compiler's output on a set of test programs. Although providing a more generic form of testing which is not reliant on the chosen test cases, compiler verification is a highly complex task. Furthermore, each time a change is made to the compiler, the verifying proofs must be redone, hindering the incentive to implement changes and improvements.

In response to the implementation difficulty of compiler verification, there arose a method called translation validation [1], which proves that a compiler's produced output is the same as the provided input, independent of how the output has been produced. Automated translation validation requires a common semantic framework between the source code and generated target code, automated production of the proof which shows that a compiler's output is the same as its input, and a proof checker.

A modern example of translation validation is Alive2, which is a bounded translation validation tool for the intermediate representation (IR) used by the open source compiler LLVM [6]. Alive2 is an open source, fully automated tool which bounds the resources used to perform verification. Although proving the functional correctness of LLVM is a highly impractical task, Alive2 offers a way in which to check that individual executions of the compiler are correct. Although providing benefits over compiler verification, translation validation is still a complex testing method which may not always be practical.

A tool called *pLiner* was developed to isolate lines of code that perform floating-point operations that have compiler-induced variability [5]. Floating-point (FP) arithmetic does not satisfy associative and distributive laws. Thus, a compiler optimization that changes the order of FP operations can induce inconsistences in the output. The authors wrote a tool that manipulates the abstract syntax tree of a program to rewrite the source code. They change the code to use *long double* types instead of *float* and *double* types to detect changes in the program output. They apply a binary search to isolate the problem at each of the following levels: function, loop, basic block, source line. This allows them to isolate portions of the code that can benefit from using the additional precision.

A tool known as *bugfind* was developed to assist in the debugging of optimizing compilers [3]. The *bugfind* tool attempts to determine the highest optimization level at which each file within a program can be compiled and produce correct output. To isolate a function that was not optimized correctly, one has to place each function within the application in a separate file. This tool also relies on a different compilation of each function that produces correct code. The *bugfind* tool uses the *make* facility in Unix and is generalized enough to work with different compilers.

The *vpoiso* tool finds not only the failing module, but also the first code-improving transformation within a function that causes incorrect results [2, 8]. The transformation number can be used to access the point in the *vpo* compiler when the transformation is about to be applied. This finer level of isolating errors is important when optimization errors occur in large functions or code size increasing transformations are performed.

The *asoptiso* is most similar to the *vpoiso* tool in that both tools isolate the first code-improving transformation that causes incorrect output during execution. *asoptiso* in addition allows for the isolation of only a code-improving transformation for a specified type of optimization to decrease the error isolation time, which is important when isolating errors using long-running simulations. We have shown this type of error isolation can significantly decrease the required number of simulations when a specific compiler optimization is being tested that is likely the cause of the error.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The *asoptiso* tool has been crucial in testing the simulation output accuracy of *asopt*-produced code for a setting in which both the simulator and assembly optimizer are for a new ISA. Throughout the testing process, *asoptiso* has significantly reduced the amount of time required for compiler writers to find and debug various erroneous transformations applied by *asopt*. The process of testing one new transformation at a time works well with the implementation of an isolation flag. A configuration file gives *asoptiso* flexibility regarding which simulation machine is used, flag combinations, starting binary search ranges, and timed simulation testing. After the completion of an error isolation run, the three output files provide information which allows the user to go directly to the first erroneous transformation with a debugger.

There are several tasks which could be implemented for future work on the assembly optimizer error isolator. The first would be to allow multiple isolation flags. In these earlier stages of testing the assembly optimizer, only two or three flags are typically being used with the isolator. Given this, and the ability to use diff minimization to find a first erroneous transformation of a non-isolation flag type so long as it is in the isolation flag range, there has been very little need to isolate on multiple flags at once. However, as the assembly optimizer becomes more complex, and a greater number of optimizations are tested together, there will likely arise a need for testing multiple isolation flags at once due to possible future interactions between transformations.

An additional improvement which could be made to *asoptiso* is increased flexibility with new simulators or simulation tools. In the future, the way that the simulation tools interact with the assembly optimizer may change. It is also possible that the assembly optimizer may have to vary it's transformation application slightly depending on which simulation machine is being used—a current example of this already exists in that pseudo expansion must be applied when implementing the SCALE VLIW ISA. Currently, the isolator interacts with the assembler, linker, and simulator via each program's makefile rules, but future changes may allow for a more efficient method. As the SCALE research continues, the isolator may need an improved or more thorough configuration file to adjust for different simulator tools as smoothly as possible.

asoptiso could be further improved by a change in how it is used. Currently, the isolator is executed once and the user must wait for the result. This process could be sped up through parallelism by simultaneously running multiple instances of the isolator on the same program, but with different search ranges. There is opportunity for further speedup by using this method on a faster, more powerful multiprocessor machine. A program could be made to automatically run simultaneous instances of the isolator and analyze the results. With the current use of the isolator, waiting for the isolator to find the first erroneous transformation on a program which executes for a single simulation for many hours or days is not a feasible option. This has not been an issue so far, since errors in faster running programs often fix the same problems within the longest running programs; however, this may not be the case in the future.

BIBLIOGRAPHY

[1] A.Pnueli, M. Siegel, and F. Singerman. "Translation Validation". In: *Proceedings of TACAS '98*. Mar. 1998.

[2] M. R. Boyd and D. B. Whalley. "Isolation and Analysis of Optimization Errors". In: *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. June 1993, pp. 26–35.

[3] J. Caron and P. Darnell. "Bugfind: A Tool for Debugging Optimizing Compilers". In: *SIGPLAN Notices* 25.1 (Jan. 1990), pp. 17–22.

[4] J. Chen et al. "A Survey of Compiler Testing". In: *ACM Computing Surveys* 53.1 (May 2020), pp. 1–36.

[5] H. Guo, I. Laguna, and C. Rubio-Gonzalez. "pLiner: Isolating Lines of floating-Point Code for Compiler-Induced bariability". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2020.

[6] Nuno P. Lopes et al. "Alive2: Bounded Translation Validation for LLVM". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. June 2021.

[7] Soner Önder and Rajiv Gupta. "Automatic generation of microarchitecture simulators". In: *IEEE International Conference on Computer Languages*. Chicago, May 1998, pp. 80–89.

[8] D. B. Whalley. "Automatic Isolation of Compiler Errors". In: *ACM Transactions on Programming Languages and Systems* 16.5 (Sept. 1994), pp. 1648–1659.

BIOGRAPHICAL SKETCH

Abigail Mortensen completed her Bachelor of Arts in Computer Science at Florida State University in 2020. In the last year of her undergraduate program, she began her time as a research assistant for Dr. David Whalley. She has continued to be a research assistant with Dr. Whalley in the compilers and computer architecture office until the completion of her Master's degree. Intermittently, Abigail has been a teaching assistant for Florida State University's undergraduate software engineering and computer architecture courses, where she gained experience in teaching recitations. As a research assistant, Abigail has worked on building testing environments, testing tools, and implementing functionality of an assembly optimizer for a new instruction set architecture.