2022

# A Study on Loop Unrolling at the Assembly Code Level

Joseph Zilonka

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

# A STUDY ON LOOP UNROLLING AT THE ASSEMBLY CODE LEVEL

By

JOSEPH ZILONKA

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

2022

Joseph Zilonka defended this thesis on July 18, 2022.
The members of the supervisory committee were:

Dr. David Whalley
Professor Directing Thesis

Dr. Grigory Fedyukovich
Committee Member

Dr. Gary Tyson
Committee Member

Dr. Soner Onder
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

First and foremost, my thanks to Dr. David Whalley for his mentorship and encouragement throughout this study. In addition to serving as the lead investigator for our research, he envisioned the idea for this thesis and authored the original implementation of our loop unrolling algorithms. From the beginning, he trusted that I could successfully pursue this thesis, and never hesitated to assist me when problems arose. Without his guidance, this study would not have been possible.

Next, I thank Dr. Gang-Ryung Uh for his help and his many contributions to our loop unrolling code. His code expanded the horizon of our original algorithms, and enabled us to unroll significantly more loops than we otherwise would have. His commitment to our research and the many hours he spent fixing bugs are profoundly appreciated by all involved in this study.

From the FSU research team, my thanks to Abigail Mortensen and Skylar Scorca, whose contribution included developing a set of tools that saved us uncountable hours in finding bugs and collecting statistics. Without their help, this study would not have been as comprehensive as it is.

Finally, I would like to thank Dr. Soner Onder and Scott Pomerville of the Michigan Technological University (MTU) research team for developing ADL and providing us with technical assistance. Their insight and guidance allowed us to produce performance statistics for the Results section. Without their assistance, we would not have been able to test our unrolling strategies for correctness.

This work was supported in part by NSF grants CCF-1900788, CCF-1901005, CCF-1823398, CCF-1823417, OISE-2103103, and OISE-2103105.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Loop unrolling is a compiler optimization that can improve the performance of applications without explicit intervention from programmers. In this study, we analyze how traditional loop unrolling techniques attempt to improve performance, and we propose new unrolling strategies that can unroll a greater number of loops than what is usually possible.

We perform loop unrolling using our assembly optimizer, which analyzes and optimizes code at the assembly level instead of at the compiler level. This gives us a different perspective of the code than what a typical compiler would have, as we do not concern ourselves with the source code or an intermediate language. We are free to make modifications at the instruction level, which allows us to optimize large sections of code that might otherwise be impossible to work on.

The optimizer collects statistics on the loops it unrolls, some of which are shown in our Results section. We ran our optimizer's unrolling algorithms on the SPEC 06 benchmark suite, and present both the statistics it generated as well as performance results from three different simulators. While performance gains are highly dependent on the application being optimized, we have shown it is possible to achieve considerable improvements by the use of loop unrolling.

# CHAPTER 1

# INTRODUCTION

Loop unrolling is a code optimizing transformation that aims to improve the performance of an application by modifying the structure of its loops. Making changes to a loop, such as removing a branch instruction and replicating iterations inside the loop body, can reduce the overhead associated with running the loop while also creating more opportunities for the compiler or the processor to perform instruction scheduling. When applied to applications that spend significant amounts of time executing loops, loop unrolling can provide performance gains in terms of overall execution time and the number of CPU cycles used.

At a high level, loop unrolling is the process of reducing the number of times a loop has to repeat its code, or *iterate*. We normally think of a loop as being "rolled up," since the loop body only appears once in the code yet is run many times during an application's execution. If we were to make a copy of the loop's code and place it inside the loop's body, we would have "unrolled" the loop by removing iterations and adding them to an individual iteration's work. This reduces the total number of iterations the loop will execute by a factor of how many times the loop body is copied. This quantity is called the *loop unroll factor*.

Take for example Figure 1.1a, which is a typical C style *for* loop. This loop runs $n$ times and increases $i$ by 1 after every iteration. Figure 1.1b represents the code in Figure 1.1a unrolled with an unroll factor of 4. The original code is replicated 3 times inside the loop's body, and each replication is modified to substitute for the work of the iteration it is replacing. For this unrolled loop, the modification necessary is to add an offset to $i$ each time $i$ is referenced. The increment of $i$ at the end of an iteration has also been adjusted. The loop should now iterate $\frac{n}{4}$ times, so $i$'s increment has changed to 4 instead of 1. Assuming $n$ is a multiple of 4, the unrolled loop will only need $\frac{1}{4}$ of the iterations the code in Figure 1.1a does to complete the same task. If 4 is not a factor of $n$, then more code needs to be added to the loop to ensure we do not execute too many (or too few) iterations. The solution we use to guarantee the correct number of iterations are executed will be explained in a later section of this thesis.

The primary benefit of unrolling a loop like this is the removal of expensive and unnecessary operations. Every time an iteration of a rolled up loop finishes, the control flow jumps to the top

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

(a) A C style *for* loop.

```
for (i = 0; i < n; i += 4) {
    arr[i] = i;
    arr[i+1] = i+1;
    arr[i+2] = i+2;
    arr[i+3] = i+3;
}
```

(b) Same loop after unrolling with an unroll factor of 4.

Figure 1.1: An example of unrolling a C style *for* loop.

of the loop and checks if the loop should continue running. These jumps and checks can add a lot of overhead to the loop, especially if the loop will run for a long time. It does not make sense to perform these extra operations if we know in advance that the total number of operations can be safely reduced by some constant factor. If we can figure out that factor beforehand, we can utilize loop unrolling to decrease the number of operations and potentially improve performance.

A secondary benefit to unrolling a loop is the new opportunities it introduces for other optimizations. Since loop unrolling adds more instructions to the code while eliminating k-1 instances of the loop branch (where k is the loop unroll factor), an optimizing compiler or assembly optimizer may find more occasions to perform optimizations after unrolling. Also, for architectures that support Out-of-Order (OoO) execution or Very Long Instruction Words (VLIW), more instructions without branches creates more opportunities where instruction scheduling can dynamically or statically reorder instructions to avoid stalls.

Loop unrolling is typically done at the source or intermediate code level, as this allows the optimization to be performed in a machine independent manner. As shown in Figure 1.1, a high-level programming language is sufficient for a programmer or compiler to unroll a loop and potentially improve performance. Most of the time, the code optimizer within a compiler does the unrolling as it is simpler for a compiler to perform the necessary safety checks.

Since C style *for* loops often have some counter variable that is incremented, they are sometimes the only targets on which loop unrolling is performed. However, many loops not expressed as *for* loops are also unrollable; they just require more work to safely unroll. If we do not perform loop unrolling on the source code and instead perform it on the intermediate or assembly language, we might be able to unroll a larger fraction of loops in a given application.

This study shows that, by exclusively working at the assembly level, we are able to unroll a large fraction of the innermost loops in many applications. We have developed an assembly optimizer that, among other things, can unroll the innermost loops of previously compiled assembly code. Working at this level gives us detailed heuristics which can be used to determine when and how unrolling is applied to a loop. Our optimizer contains three different strategies to unroll loops, and each strategy's effectiveness has been tested using several benchmarks. The performance results of these strategies, as well as the statistics gathered about the characteristics of successfully unrolled loops, will be presented in a later section.

We begin this study with an overview of our testing environment and our assembly optimizer. We then go into detail about the three strategies used to unroll innermost loops, with code examples to explain the unrolling process. Afterward, we explain which team members of our assembly optimizer project contributed to the current unrolling implementation, and how the implementation has improved over the course of this study. We then analyze the various loop characteristics encountered while unrolling, and describe the performance results obtained after unrolling with all three strategies. Finally, we present our conclusions from this study.

# CHAPTER 2

# SIMULATION AND COMPILATION INFRASTRUCTURE

This study and the loop unrolling infrastructure created for it are part of the National Science Foundation (NSF) Statically Controlled Asynchronous Lane Execution (SCALE) project, under NSF grants CCF-1901005 and CCF-1900788. This project involves developing simulation and compilation support for a set of related but distinct Instruction Set Architectures (ISAs). This chapter begins by describing two ISAs that are part of the SCALE project. We then delineate the simulation system used to simulate these ISAs. We next provide a detailed description of the compilation system, which produces the files necessary to run the simulators. Of particular importance to this study is the compilation system, as that is where loop unrolling takes place.

## 2.1 SCALE ISAs

Two ISAs implemented as part of the SCALE project are the *SCALE base* ISA and the *SCALE VLIW* ISA. Both ISAs were used to obtain results for this study.

The SCALE base ISA is similar to the MIPS ISA, with some key differences to allow information to be encoded in the instruction set. One such difference is that all load and store instructions are only supported by a register deferred addressing mode, which means all loads and stores use a zero displacement from the base register. This is done to decrease the number of stages in the instruction pipeline, and to allow information supporting more advanced features to be encoded in loads and stores. Another difference is the restriction of integer branch instructions to *bnez* (branch not equal to zero) and *beqz* (branch equal to zero). This means each integer branch references a single register. To support these two branch instructions, a new *seq* (set equal) instruction was added which sets a destination register to 1 if the two argument registers contain the same value.

The SCALE VLIW ISA uses instructions from the SCALE base ISA, but is restricted to Very Long Instruction Word (VLIW) execution. Code generated for this ISA is packaged into groups of instructions that are simultaneously issued. We refer to such groups as *VLIW packs* and the position of an instruction within a VLIW pack as a *lane*. The number of instructions in a VLIW

pack, as well as what kinds of instructions can go in each lane, is configurable at compile and simulation times. The SCALE VLIW ISA also allows instructions to be placed after a branch instruction within a VLIW pack. An instruction located after a branch is only executed if the branch is predicted not to be taken. This lets conditional branches support a simple form of predication in the ISA. Meanwhile, an instruction after an unconditional transfer of control (jump, call, return) within a pack is never executed. This requires us to fill empty lanes with *nop* (no operation) instructions. There can be multiple transfers of control within a VLIW pack, but only the last one can be unconditional.

## 2.2 ADL Simulation System

To simulate the SCALE ISAs, we use the Architecture Description Language (ADL) simulation system. This system takes a microarchitecture specification file written in ADL as input and produces an assembler, linker, and disassembler for that microarchitecture [6]. The description language provides constructs for specifying microarchitectural features such as pipelines, control, and memory hierarchy. It also provides the ability to create new ISAs, including assembly syntax and the corresponding binary representation. Once an architecture specification is finalized, it is used to generate simulators that run statically linked executables created by the assembler and linker.

The simulators produced are either functional or cycle-accurate. They perform a more realistic simulation than many commonly used simulators, as instructions are explicitly fetched from the instruction cache, data values are explicitly loaded from the data cache, values are explicitly forwarded through the pipeline, and so on. This ensures that the ISAs are correctly implemented and that the statistics generated from running simulations are reliable.

## 2.3 SCALE simulators

The SCALE simulators described in this section and used for this study are developed by Dr. Soner Onder and his students at Michigan Technological University (MTU).

The SCALE base ISA was first used to create a SCALE functional simulator. The functional simulator is the fastest of the various simulators and is only used to check if the input executable generates the correct output. Simple measurements, such as the number of instructions executed and the number of times memory was accessed, are logged during program execution. For this study,

we use the SCALE functional simulator to ensure that the transformations performed during loop unrolling are valid and that the semantic behavior of the executable remains the same.

The SCALE base ISA was also used to create a SCALE pipelined simulator. The pipelined simulator provides a five-stage integer pipeline, which includes the following stages: IF (Instruction Fetch), ID (Instruction Decode), RF (Register Fetch), EX (EXecution)/MEM (MEMory access), and WB (Write Back). The MEM stage is performed in the same cycle as the EX stage, as load and store instructions do not have a displacement for the base register and thus do not require the calculation of an effective address. In this study, we use the pipelined simulator as one of our testing environments to observe the effect of loop unrolling on an executable's performance.

In addition, the SCALE base ISA was used to create a superscalar/Out-of-Order (OoO) simulator. This simulator imitates a typical OoO processor that includes in-order issue, OoO instruction execution, and in-order commit through a reorder buffer. Like the pipelined simulator, we use the superscalar simulator as a testing environment to scrutinize the effects of loop unrolling on an OoO executable's performance.

The SCALE VLIW ISA was used to create a SCALE VLIW simulator. The SCALE VLIW simulator uses SCALE base instructions that are placed into VLIW packs. Instructions within a VLIW pack are fetched, decoded, and executed together. If any instruction in the pack stalls, then all instructions in the pack stall. This simulator is used as a testing environment to see the effects of loop unrolling on a SCALE VLIW executable's performance.

## 2.4 Compilation System

Our compilation system is designed to compile the Standard Performance Evaluation Corporation (SPEC) 95 and 06 benchmarks for the SCALE simulators. This system can compile the SPEC benchmarks into low-level code for the various ISAs and perform code-improving transformations on the code in the process.

A conventional compiler is used to transform the source code into assembly code. Figure 2.1 shows the process of generating code for the various ISAs starting from the source code. We first 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, while also leveraging code optimizations provided by *gcc*. We then developed a new assembly optimizer, called *asopt*, that takes an assembly file as input and produces modified assembly code as output. This optimizer can translate instructions

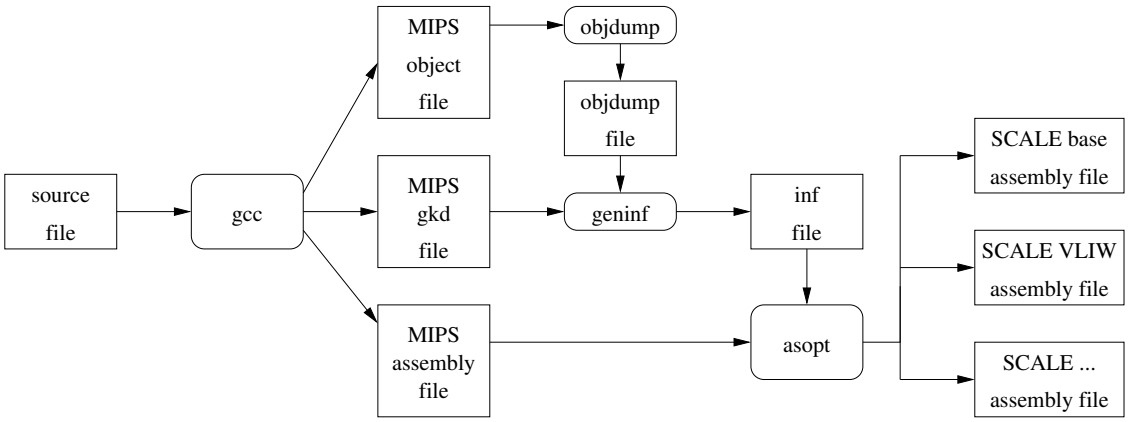



Figure 2.1: SCALE Code Generation Process

into new ISAs when necessary (for example, MIPS to SCALE VLIW), as well as perform a variety of analysis and code-improving transformations on the input assembly.

To properly determine which registers are live at any given point in a function, we need to know which registers are being passed to function calls and which registers are being used as return values. Rather than attempting to perform interprocedural analysis to determine this information, we gather it from a side effect of the *gcc* compilation process [4]. We use an option in *gcc* to produce a *.gkd* file that contains information about each *gcc* RTL (instruction). This file is then used to determine which registers are used to pass values into function calls. We also generate a MIPS object file with symbolic debugging information, which gets passed to the Linux program *objdump* to generate a *.objdump* file. The *.objdump* file contains information about each function's return type, including whether registers are used to store return values. Both the *.gkd* and the *.objdump* files are then passed to a tool called *geninf* [4] that parses the files and condenses the necessary information into a new *.inf* file. *asopt* then uses the *.inf* file to determine which registers are live at any given point in the code.

Creating a simplified information file for use with our optimizer is helpful as sometimes *asopt* needs to process assembly code written by hand. This is the case for some of the system libraries used by our simulators. Because those files are not generated by *gcc*, we do not have corresponding *.gkd* and *.objdump* files for them. Unlike *.inf* files, *.gkd* and *.objdump* files are large and very difficult to write by hand. Having a simple information file that *asopt* can use instead allows us to easily create the file when the previous option is impractical.

When *asopt* is run, it reads in both the *.inf* and MIPS-assembly files and produces a SCALE assembly file as output. Various flags are passed to *asopt* to either select a code-improving transformation to perform or select a target ISA for the output. For this study, we expanded the set of flags in *asopt* to support different loop unrolling techniques. The techniques implemented can unroll loops with a compile time, execution time, or an unknown number of iterations (we refer to this last one as *naive loop unrolling*). More details on these techniques will be discussed in the next section of this thesis.

Before code-improving transformations take place, *asopt* reads in a function of the input file instruction by instruction. It identifies the type of each instruction, determines which registers are set and used by an instruction, and builds a control flow graph for the function. It then translates each MIPS instruction into a corresponding SCALE instruction if that MIPS instruction is not compatible with SCALE. MIPS pseudo instructions are expanded at this step so that each underlying operation has a one-to-one mapping with a SCALE instruction. This expansion is necessary when performing some low-level code-improving transformations, such as scheduling instructions into VLIW packs. Putting instructions into VLIW packs requires us to package a specific number of machine instructions together at compile time, which causes a problem when using pseudo instructions. Pseudo instructions are expanded by the assembler, so there is no way to guarantee a pseudo instruction will not overflow a VLIW pack after expansion.

After translation into SCALE compatible instructions, *asopt* performs the code-improving transformations. This step is where loop unrolling takes place. Other transformations, such as accumulator expansion and elimination of true dependencies, are also performed at this point if enabled by command-line arguments or configuration files. If code is generated for SCALE VLIW, *asopt* schedules instructions into VLIW packs immediately after performing these transformations. The VLIW scheduling itself performs other code-optimizing transformations, such as register renaming and scheduling instructions across basic block boundaries.

After code-improving transformations are complete, *asopt* prints the resulting SCALE code from the previously read-in function to standard output. It then moves on to the next function in the input file, as *asopt* can only process one function at a time. The steps outlined above are repeated for each function until the end of the MIPS assembly file is reached and the entire SCALE assembly output is produced. After all assembly files that comprise a program have been processed, the output is passed to the ADL assembler and linker to generate an executable for an ADL simulator.

# CHAPTER 3

# LOOP UNROLLING STRATEGIES

This chapter goes over the loop unrolling techniques we researched for this study. We begin with a discussion of the general strategy used to unroll a loop, along with explanations of the terminology used to describe the unrolling process. We then cover the three techniques implemented in *asopt* to perform loop unrolling, along with code examples to show what loops look like before and after unrolling.

## 3.1 General unrolling strategy

The typical starting point when unrolling a loop is to look for a *basic induction variable* in the loop's exit condition. A *basic induction variable* is a variable in the loop that is either increased or decreased by some constant during each iteration. If a basic induction variable is used in the loop's exit condition, then the constant changing the variable is referred to as the loop's *stride*. If the exit condition is testing the basic induction variable against some limit to determine if the loop should continue, we may be able to modify the stride to prepare for removing some of the loop's iterations.

There is one requirement a basic induction variable must meet in order for it to be usable for loop unrolling: the value it is testing against in the exit condition must be *loop invariant*, which means it remains constant during the loop's execution. This allows an unrolling algorithm to calculate the number of iterations the loop will execute at run time, which is used to ensure no iterations are added or removed during unrolling.

Going back to Figure 1.1a, it is easy to see that this *for* loop meets this requirement. The variable $i$ is a basic induction variable in the loop and is used in the loop's exit condition. The loop's stride is 1 since $i$ is increased by 1 after every iteration. The exit condition tests $i$ against $n$ every iteration, and $n$'s value is loop invariant.

Once a suitable basic induction variable is found, the next step is to start replicating the loop's code inside of the loop's body. To obtain the desired unroll factor, we make unroll factor minus 1

replications of the loop body. Depending on whether the loop's code is contiguous, we may be able to copy the loop's code as-is and append it to the original body.

A loop is considered contiguous if all basic blocks between the top and bottom blocks are part of the loop. If the loop is contiguous, we can copy the body however many times are necessary to match the unroll factor. If the loop's code is not contiguous, a choice must be made as to whether to try and reorganize the loop into a contiguous form. Our assembly optimizer attempts to do this, but not all loops can safely be modified into a contiguous form. If the reorganization fails or is too difficult for *asopt* to perform, then the loop is marked as not unrollable and the algorithm moves on without unrolling it.

After replication, references to the basic induction variable in the loop are updated as needed. We see an example of this in Figure 1.1b. Since the unroll factor in that example is 4, the loop body is replicated 3 times. Each reference to $i$ in the unrolled loop is then offset in increments of 1. Finally, the stride of $i$ is changed to 4 to remove the iterations that are covered by the replications of the original loop's body.

Not all loops follow the structure of the loop in Figure 1.1, and as such they may not require any adjustments to references of their basic induction variables. An example of this is when the basic induction variable is updated mid-iteration and its new value is used in that same iteration. This is common in C style *while* loops, where updates of the basic induction variable tend to be within the body instead of at a dedicated position like in *for* loops. The replication of the loop's body copies all updates to the basic induction variable such that, after replication, the number of updates to the variable per iteration reflects the desired unroll factor. Therefore, in these cases, no adjustments to the variable are needed after the replication stage.

The last step in the general strategy is to ensure that the unrolled loop does not execute too many or too few iterations. This step is less straightforward than the previous ones, as it depends on the unrolling strategy being used. We end this section with a high-level overview of what happens in this step, and leave the specifics of each strategy to the next sections of this chapter.

To guarantee the loop does not iterate too many times, we first check if the basic induction variable can exceed the exit value while running the unrolled loop. This is done by calculating the number of iterations the unrolled loop will execute at run time. The number of iterations the loop will run is equal to $|\frac{exit\ value - init\ value}{stride}|$. Taking the absolute value of this calculation is necessary as the initial value, the exit value, and the stride can be positive or negative. We check that the

sign of the stride makes sense when using this formula (i.e., the stride is positive when the exit value is larger than the initial value).

If the starting value and exit value are known at compile time, we may have a stride that causes the basic induction variable to match the exit value at the end of the loop. A good example of this is if the value of $n$ in Figure 1.1b is equal to 20. Since 20 is divisible by 4, $i$ equals 20 after 5 iterations. The unrolled loop thus updates *arr* exactly $n$ times, and ends after the $n$th update. In cases like this, where there is no remainder, no further changes are needed so the unrolling process is complete.

Many unrolled loops do not work out this nicely, so we must deal with the cases where the basic induction variable is off by some amount. If we know the exact start and end values of the loop in advance, we can produce extra iterations before or after the unrolled loop to compensate for those the unrolled loop would miss. Looking at Figure 1.1b again, consider if $n$ equals 21. If we leave the unrolled loop as is, we will execute 3 extra iterations and add 3 extra values to *arr*. To fix this, we can replace $n$ in the unrolled loop with the constant value 20 and replicate one iteration outside of the loop, right after it ends. We then add a new increment of $i$ after the extra iteration.

When the start or end values are unknown at compile time, we are unable to determine whether the loop will execute too many iterations. This situation requires us to do more work in advance to safely unroll the loop. The way *asopt* approaches this problem is shown in Figure 3.1, which has been written in C and slightly modified for the sake of example. As a reminder, *asopt* does not work at the source code level, so the use of C code and constructs such as *goto* are only meant to visualize what is taking place at the assembly level.

Figure 3.1 is similar to Figure 1.1, except that the start and end values are not known until execution time and the unroll factor has changed to 2. In Figure 3.1b, we begin unrolling by pulling out the assignment of *start* to $i$ from the *for* loop's header. This is done so we can apply the *if* statement, which is a safety check guarding against entering the unrolled loop. We check if adding the stride (2 in this case) to $i$ exceeds the limit $n$. If this sum exceeds $n$, then entering the unrolled loop will execute too many iterations. To avoid this, an alternate path is provided for the control flow to take when the unrolled path is not safe.

Our alternate path is a virtually identical copy of the original loop, placed right after the unrolled loop's body. If the unrolled loop is unsafe to enter, the control flow jumps to this copy of the loop and executes it instead. Since the only difference between this copy of the loop and the original loop is the removal of the initial assignment statement, this copy produces the same effect

```
for (i = start; i < n; i++) {
    arr[i] = i;
}
```

(a) A C style *for* loop.

```
    i = start;

    if (i + 2 > n) {
        goto rolled_loop;
    }

unrolled_loop:
    for (; i < n - 1; i += 2) {
        arr[i] = i;
        arr[i+1] = i+1;
    }

rolled_loop:
    for (; i < n; i++)
        arr[i] = i;
```

(b) Same loop after unrolling with *asopt*'s method by an unroll factor of 2. The label above the unrolled loop is for visual purposes only.

Figure 3.1: An example of unrolling a C style *for* loop with bounds only known at execution time.

as the original loop. Conveniently, the positioning of this copy also helps the unrolled loop execute the correct amount of iterations. Notice in Figure 3.1b's unrolled loop that the exit condition exits the loop when $i$ is greater than or equal to $n$ - 1. This change is made to address the case where the basic induction variable is off by some amount if the unrolled loop is executed. Since we do not know whether extra iterations need to be run, we shorten the length of the unrolled loop by the unroll factor minus 1 as a precaution. The control flow then falls into the rolled loop once the unrolled loop ends, which executes any leftover iterations.

This completes the general description of how we perform loop unrolling. We now discuss the details of how unrolling is implemented at the assembly level in *asopt*.

## 3.2 Unrolling with a compile-time number of iterations

The first strategy we use to unroll loops in *asopt* is to try to unroll them with bounds we find at compile time. Before unrolling begins, we run an analysis that checks if the loop meets the various requirements for unrolling and calculates the number of iterations the loop will execute at run time. If a constant initial value and a constant exit value are found by the analysis stage and the

loop satisfies the other requirements for unrolling, then the loop is unrolled using our compile-time number of iterations strategy.

This strategy first determines if any iterations need to be replicated outside of the loop to address the case where the basic induction variable is off by some amount. If iterations need to be added, then the exit value in the unrolled loop is replaced with a smaller value that is calculated to be reached by the basic induction variable during the unrolled loop's execution. The unroll process then continues by replicating the loop's body, fixing the control flow to branch to the correct locations if the unrolling process moves basic blocks around, and adding iterations outside of the loop.

We will work through an example starting from C code and ending with a final unrolled loop at the assembly level. Figure 3.2 shows a loop in C that was compiled by *gcc* into MIPS assembly code. The code before location $L2 in Figure 3.2b (which is called the *loop preheader*) sets up the variables used during the loop, and the code between $L2 and the bottom of the figure is the loop itself. $16 is the register representing the basic induction variable $i$ in Figure 3.2a. $16 first gets assigned the value of 0, and is passed to *printf* as a parameter after the loop begins. $16 is then increased by 1 in the *addiu* instruction toward the end of the figure. It is finally used in the *bne* (branch not equal) instruction at the bottom of the figure, which branches back to $L2 if the value of $16 does not equal the value of $17. $17 is assigned the value of 80 before the start of the loop, so $17 represents the exit value of the loop. The other registers are used to facilitate calling *printf*.

It should be apparent that the code in Figure 3.2b is a few steps removed from a direct translation of Figure 3.2a. During compilation, *gcc* performs a number of optimizations on the source code to transform it into efficient assembly code. One such transformation is the removal of the initial instruction that checks if the control flow should enter the loop. With an initial value known to be smaller than the exit value, it is obvious that the loop should be entered. *gcc* notices this and removes the unnecessary check guarding entry into the loop. Another such transformation is the combination of comparison and branch instructions into one instruction that can do both operations. Because the loop's stride is 1, we can use a branch if not equal instruction to exit the loop when the basic induction variable reaches the limit. In this example, we want to branch back to the top of the loop when the basic induction variables does not equal the limit. To facilitate this, *gcc* replaces the less than comparison and branch instructions with a single *bne*.

The analysis stage of *asopt* reads this code and realizes that 0 is the initial value of the basic induction variable and 80 is the exit value of the loop. It then determines that $16 is a valid

```
for (i = 0; i < 80; i++) {
    printf("%d\n", i);
}
```

(a) A C *for* loop before compilation.

```
        move    $16,$0          # $16 = 0
        la      $18,$LC0        # $18 = &LC0
        li      $17,80          # $17 = 80
$L2:
        move    $5,$16          # $5 = $16
        move    $4,$18          # $4 = $18
        jal     printf          # call printf()
        addiu   $16,$16,1       # $16 += 1
        bne     $16,$17,$L2     # goto $L2 if
                                # $16 != $17
```

(b) Same loop after compilation into MIPS assembly code.

Figure 3.2: A C *for* loop that has been turned into MIPS assembly code by *gcc*.

basic induction variable and $17 is both the exit value of the loop and is loop invariant. Using the formula for a positive stride mentioned previously, it calculates the total number of iterations to be 80 and marks the number of iterations as known at compile time. This satisfies the requirements for unrolling a loop with the compile-time number of iterations strategy, so the analysis stage ends and the algorithm for the compile-time strategy begins.

The next step depends on what unroll factor was specified to *asopt* before unrolling began. If the unroll factor evenly divides 80, then the unrolling process does not create extra iterations as the basic induction variable reaches the value of 80 by running the unrolled loop. If the unroll factor does not evenly divide the limit, however, the process needs to add iterations after the unrolled loop's body. We have chosen an unroll factor of 3 for this example to demonstrate how extra iterations are added after an unrolled loop.

The result of unrolling Figure 3.2b is shown in Figure 3.3. Blank lines are inserted to visually divide the sections of the loop. The main body of the loop is replicated twice, and each copy of the body is placed directly after the previous copy. The branch instruction after the original body is replaced by comparison and branch instructions put at the end of the third copy of the body. After this comparison and branch, the body is replicated two more times in the section labeled $L2_DUP1. These are the extra iterations *asopt* found the loop needed for our chosen unroll factor. After $L2_DUP1's section, the control flow falls through to the next part of the program and ends the loop. While $L2_DUP1's iterations are necessary to ensure proper execution of the loop, the actual unrolled loop is contained between $L2 and $L2_DUP1.

```
        move    $16,$0
        lalui   $18,$LC0
        laori   $18,$18,$LC0
        addiu   $17,$0,78
$L2:
        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        slt     $1,$16,$17
        bnez    $1,$L2
$L2_DUP1:
        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1
```

Figure 3.3: Figure 3.2b unrolled with *asopt* by an unroll factor of 3.

The exit value $17 is changed to 78 instead of 80 in the loop preheader to support the new unrolled loop body. 3 does not evenly divide 80, so *asopt* calculates the last multiple of 3 before 80 to use instead. The unrolled loop iterates 26 times, incrementing the basic induction variable by one 3 times after each iteration and stopping when the basic induction variable equals 78. At iteration 26, the *slt* stores 0 in register $1 causing the branch instruction to fall through and exit the unrolled loop. The control flow then enters $L2_DUP1's section, which contains two more copies of the loop body. These two iterations call *printf* two more times and bring the basic induction variable's value up to 80, completing all of the work the original loop needs to do. When this code is run instead of the original loop's code, it only requires 26 iterations to produce the same effect as 80 iterations of the original loop. That means 54 branch instructions were removed from this loop, which can lead to a savings in CPU cycles used.

Note that some instructions in this output have changed from one form to another after being run through *asopt*. Instructions like *lalui* (load address load upper immediate) and *laori* (load address or immediate) are SCALE ISA instructions that are direct translations of incompatible instructions from the MIPS ISA. The *slt* and *bnez* instructions are also translated from MIPS to SCALE, however they were translated to be *seq* and *beqz* instructions at first. This is because the loop's original branch instruction is *bne*, which directly translates to a set equal instruction followed by a branch if equal to zero instruction. The reason these instructions changed into *slt* and *bnez* is that we have a restriction in *asopt* that all branches used for exit conditions in loop unrolling must be *slt*. This decision was made early on in the project, as it was deemed necessary at the time. This requirement is no longer necessary

for the compile-time strategy, as using *seq* would work fine. However, for compability reasons with the execution-time method, this condition remains in place. We hope to remove it in the future.

## 3.3 Unrolling with an execution-time number of iterations

The second strategy we use to unroll loops is to try to unroll them with the execution-time number of iterations method shown in Figure 3.1b. During the analysis stage, *asopt* first checks if it can unroll a loop with the compile-time number of iterations method. If the loop does not meet the requirements for unrolling with this method, a second set of checks are run to see if the loop can be unrolled with the execution-time strategy. The requirements for using this method are more relaxed than the compile-time number of iterations strategy, since the initial value or exit value do not have to be known during unrolling.

Figure 3.4a shows a C function that takes an integer $n$ as a parameter and a *for* loop that uses $n$ as its exit value. The resulting MIPS code after compiling with *gcc* is shown in Figure 3.4b. The lines with 3 dots represent load and store instructions modifying the stack, and they are added to the figure to simplify the code. This output is very similar to the one seen in Figure 3.2b, with a couple of exceptions. The first change is the *blez* (branch if less than or equal to zero) instruction at the beginning of the function. This branch jumps to $L6 (which skips running the loop) if $n$, whose value is initially in $4, is less than or equal to zero. The value of $n$ must be in $4 because the MIPS calling convention requires the caller to place the first function argument into $4. The second change is the *move* instruction moving the value of $4 into $17. $4 is overwritten when calling *printf* later in the code, since *printf* is expecting arguments. $17 is not being used to store other values in the function, so it gets used to store the exit value and free up $4. The rest of the code works the same way as in Figure 3.2b.

When *asopt* reads this code, the loop does not pass the checks of the compile-time number of iterations strategy since the exit value is not known at compile time. The loop does, however, pass the checks for the execution-time strategy since the exit value is loop invariant. The basic induction variable is again $16 and, like in the previous example, is usable for loop unrolling. Likewise, $17 is again the exit value for the loop and is usable for unrolling since it is loop invariant. Because the exit value is not known at compile time, the total number of iterations equals $|\frac{exit\ value}{stride}|$, which simplifies to the value in $17 since the stride is 1.

Unlike the compile-time number of iterations method, where the unrolled loop may or may not need additional iterations, the execution-time method always needs additional iterations outside of

```
void printnums(int n)
{
   for (int i = 0; i < n; i++) {
      printf("%d\n", i);
   }
}
```

(a) A C function before compilation.

```
printnums:
        blez    $4,$L6          # goto $L6 if
                                # $4 <= 0
        ...
        move    $17,$4
        move    $16,$0
        la      $18,$LC0
$L3:
        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1
        bne     $17,$16,$L3
        ...
$L6:
        jr      $31             # return
```

(b) Same function after compilation into MIPS assembly code.

Figure 3.4: A C function that is turned into MIPS assembly code by *gcc*.

the unrolled loop. This is because we cannot know whether the unrolled loop by itself covers all iterations. As we saw with Figure 3.1b, this is done by making a copy of the original loop and placing it after the unrolled loop's body.

The result of unrolling Figure 3.4b is shown in Figure 3.5. We use an unroll factor of 2 in this example to keep the figure all on one page. This output is quite different from the one seen in the compile-time number of iterations strategy. The first key difference is that the unrolled loop has an exit value of $19 and not $17. This is seen by looking at the second to last *slt* instruction before the $L3_DUP1 section of the code. Since $17's value is unknown to *asopt* at compile time, guarding instructions are added to the loop's preheader to check if entering the unrolled loop will cause the basic induction variable to exceed $17. The *addiu* instruction after the *lalui*/*laori* pair adds the unroll factor to the initial value of 0, and stores the result in $19. $19 is not used in the original function, so *asopt* allocates and uses it as a temporary storage location for this sum. The following *slt* instruction checks if this sum exceeds the limit. If it does exceed the limit, the control flow jumps over the unrolled loop to $L3_DUP1, which is the copy of the original loop that comes after the unrolled loop. If the sum does not exceed the limit, then the control flow falls through the branch and subtracts 1 from the exit value. This is the precautionary measure that ensures the unrolled loop never executes

too many iterations. We store this result in $19 since its previous value is no longer needed. $19 is now the limit of the unrolled loop, and the loop runs until $16 becomes greater than or equal to $19.

```
printnums:
        slt     $1,$0,$4
        beqz    $1,$L6
        ...
        move    $17,$4
        move    $16,$0
        lalui   $18,$LC0
        laori   $18,$18,$LC0

        addiu   $19,$0,2
        slt     $1,$19,$4
        beqz    $1,$L3_DUP1
        addiu   $19,$17,-1
$L3:
        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        slt     $1,$16,$19
        bnez    $1,$L3

        slt     $1,$16,$17
        beqz    $1,$L_SPLIT2
$L3_DUP1:
        move    $5,$16
        move    $4,$18
        jal     printf
        addiu   $16,$16,1

        slt     $1,$16,$17
        bnez    $1,$L3_DUP1
$L_SPLIT2:
        ...
$L6:
        jr      $31
```

Figure 3.5: Figure 3.4b unrolled with *asopt* by an unroll factor of 2.

The second key difference is the two instructions after the comparison and branch that check $16 against $19. These instructions are not part of the unrolled loop and only execute when the unrolled loop is finished. They check if the unrolled loop has completed all of the required iterations by comparing the basic induction variable to the original exit value $17. If the unrolled loop has completed all required iterations, then we are done executing the loop and jump over the copy of the original loop into $L_SPLIT2's section. This section contains the code that needs to execute after the loop is finished. Before unrolling, it was only accessible from the original loop by falling through the exit branch. The control flow now needs to be able to branch to this code, so *asopt* creates a label for this section and updates the branch instruction after the loop to jump to this new label.

The final difference is the copy of the original loop, which is either jumped into from the branch before the loop or fallen into after the unrolled loop finishes. This loop is virtually identical to the loop in Figure 3.4b, with the exception of the comparison and branch instructions translated from MIPS to SCALE. This loop will either run when the unrolled loop is not safe to enter, or when the unrolled loop has leftover iterations that need to be completed.

Before ending this section, we discuss another solution to unrolling loops with execution-time bounds that requires the use of a modulo operation. This solution first calculates a value from the formula *number of iterations % unroll factor*. If the result is not 0, then there are left-

over iterations the loop will miss if we only run the unrolled loop's body. To address this problem, the code first enters a sub-loop that takes the result of the modulo operation and executes that number of iterations. After this completes, the control flow falls into the unrolled loop and executes the remaining number of iterations.

While this solution requires less instructions than the execution-time number of iterations method previously presented, we believe our solution is superior due to the use of cheaper operations like addition and subtraction and the problems modulo has with negative operands. Modulo operations require dividing the operands, which is an expensive instruction that usually is not fully pipelined. Using a modulo operation would thus take away some of the benefits of performing loop unrolling. The definition of the modulo operation is also ambiguous when one of the operands is negative, which can lead to strange behavior depending on the compiler in use or the underlying architecture. Our solution avoids both of these problems by only using addition and subtraction, which are cheap and fast instructions, and have no issues with negative strides or negative initial and exit values.

## 3.4 Unrolling with an unknown number of iterations (naive loop unrolling)

Up to this point, we have enforced a strict set of requirements to determine what loops are eligible for unrolling. Any loops that do not meet our criteria are rejected and no further attempt is made to unroll them. With traditional unrolling techniques that rely on basic induction variables, not much more can be done to expand the scope of loops that are able to be unrolled. Removing or relaxing the requirements mentioned in previous sections will jeopardize the safety guarantees that ensure loops continue to execute correctly. However, we might be missing an opportunity to further improve performance if we simply dismiss rejected loops as not unrollable.

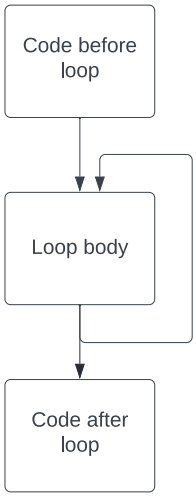
Modern CPUs are designed to fetch and execute groups of sequential instructions instead of one instruction at a time. If the code to be executed is sequential, then the CPU fetches a group of instructions in one operation and begins executing them as soon as possible. For CPUs that support executing multiple instructions during one cycle, this technique keeps the time spent on fetching instructions low. However, transfers of control counteract this by making the code non-sequential. Because the next instruction to be executed following a transfer of control is probably not in the previously fetched group of instructions, the CPU spends a cycle going to the target location and fetching the next set of instructions.

While we cannot avoid using transfers of control in our code, we can decrease the distance between a transfer of control and its target. If the distance between the two is made small enough, then the next instruction to be executed might have already been fetched. The CPU will then be able to skip fetching new instructions and instead continue to execute the instructions it already fetched. For a transfer of control that executes a lot of times (such as a loop branch), moving the target location closer to the preceding code can reduce the number of cycles spent fetching instructions.

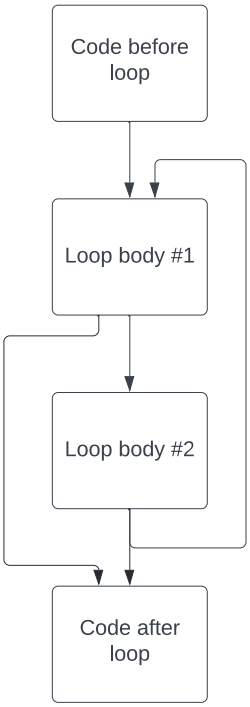
To test this out, we have developed a new loop unrolling strategy that replicates loop bodies sequentially without removing branch instructions. We call this strategy *naive loop unrolling*, due to the relatively simple way in which the loop is unrolled. A diagram showing how naive loop unrolling changes a loop is presented in Figure 3.6. The loop starts out as a contiguous set of basic blocks that has at least one branch instruction deciding whether to exit the loop or jump back to the top. After unrolling with naive loop unrolling using an unroll factor of 2, Figure 3.6b shows that the resulting loop has two copies of the body and two loop branch instructions. Body 1 is identical to the original loop body, except for the exiting branch instruction. The branch is modified to either exit the unrolled loop, or fall into body 2. Body 2 is also an identical copy of the original body, however its exiting branch instruction remains the same as the original loop branch instruction.

The idea behind these changes is that some of the loop's iterations will be located at a fall through block, instead of at the branch target. If the fall through block's instructions have already been fetched and the branch will not be taken, then the CPU can continue executing instructions sequentially and avoid another instruction fetching operation. While the CPU still has to fetch new instructions when branching out of the loop or to the top of the loop, the naively unrolled loop should reduce the overall amount of instruction fetching operations that need to occur.

This strategy is not intended as a replacement for the unrolling strategies previously discussed. Compared to unrolling with compile and execution-time bounds, naive unrolling should result in worse performance since no branch instructions are removed during unrolling. Rather, this is intended for loops that do not pass the previously described unrolling criteria of the analysis stage. Because all branch instructions are kept in the loop, we are guaranteed to always execute the correct number of iterations. This allows us to unroll loops with any exit conditions whatsoever, since we do not have to proactively calculate the number of iterations the loop will run. The exit condition can be testing a value from memory, a basic induction variable, or a value that's recalculated every




(a) A loop before naive loop unrolling.




(b) Same loop after naive loop unrolling with an unroll factor of 2.

Figure 3.6: Structure of a loop before and after naive loop unrolling.

iteration; naive loop unrolling can handle all these cases since all branch instructions are preserved. The only real requirement to unroll a loop using naive loop unrolling is that the loop is contiguous.

We now work through an example starting from C code and ending with the output after unrolling using the naive unrolling method. Figure 3.7a shows a function adapted from a loop found in the *400.perlbench* benchmark of SPEC 06. This loop iterates through a C string and returns a pointer to the position of the null terminator. The number of iterations cannot be determined at either compile time or execution time, since the string is arbitrarily long. If we try unrolling this loop with either of the previously mentioned algorithms, *asopt* will reject it on the basis of being unable to figure out the total number of iterations.

Figure 3.7b shows the resulting MIPS code after compilation with *gcc*. *str* is the only function argument, so it is initially stored in $4. The code copies $4 into $2, despite the fact that $4 is not overwritten anywhere in *find_nullterm*. This instruction was added because *gcc* decided to use $2 as both the basic induction variable and the return value. $2 is a special register in MIPS used to return values from functions, so *find_nullterm* will eventually need to assign its return value

```
char *find_nullterm(char *str)
{
   while(*str) {
      str++;
   }

   return str;
}
```

(a) A C function before compilation.

```
find_nullterm:
        move    $2,$4           # $2 = str
        lb      $3,0($4)        # $3 = *str
        beq     $3,$0,$L2       # goto $L2 if
                                # *str == \0
$L3:
        addiu   $2,$2,1         # str++
        lb      $3,0($2)        # $3 = *str
        bne     $3,$0,$L3       # goto $L3 if
                                # *str != \0
$L2:
        jr      $31
```

(b) Same function after compilation into MIPS assembly code.

Figure 3.7: A second C function that has been turned into MIPS assembly code by *gcc*.

to $2. Using $2 as the basic induction variable allows the function to skip that step and return immediately, since the value is ready to be returned.

The code then checks if the first character of *str* is the null terminator. If it is, we jump to $L2 and exit the function. $2 already contains the first position of *str*, so no extra instructions are needed to set up the return value. If the first character is not the null terminator, we enter the loop. The loop advances one character at a time through the string and falls through to $L2 when the null terminator is reached.

When *asopt* reads this code, it is unable to find an invariant end value that it can use to calculate the number of iterations. If naive loop unrolling is not enabled, then the analysis stage stops here and gives up on unrolling this loop. With naive unrolling enabled, the analysis stage marks this loop as unrollable via naive unrolling. When the actual unrolling algorithms begin, it sees that this loop is marked for naive unrolling so it attempts to unroll it using the naive unrolling strategy.

The result of unrolling the code in Figure 3.7b is shown in Figure 3.8. As we can see, not much changes between the two figures. The loop's body is replicated once, and the original loop branch is modified to either branch out of the loop or fall through to the new body. The second loop branch remains the same as the original branch.

Because naive unrolling is intended for loops that cannot be unrolled with other methods, we designed *asopt* to combine naive unrolling with our compile and execution-time number of iterations

```
find_nullterm:
        move    $2,$4
        lb      $3,($4)
        beqz    $3,$L2
$L3:
        addiu   $2,$2,1
        lb      $3,($2)
        beqz    $3,$L2
$L3_DUP1:
        addiu   $2,$2,1
        lb      $3,($2)
        bnez    $3,$L3
$L2:
        jr      $31
```

Figure 3.8: Figure 3.7b unrolled with *asopt* by an unroll factor of 2.

techniques. *asopt* first tries to unroll all loops using the compile-time method. If it cannot use the compile-time method, it then tries the execution-time method. Finally, if that fails and naive unrolling is enabled, it tries to unroll with naive unrolling.

# CHAPTER 4

# HISTORY OF OUR LOOP UNROLLING IMPLEMENTATION

Loop unrolling was first added to *asopt* by Dr. David Whalley and Dr. Gang-Ryung Uh. Dr. Whalley began writing the loop unrolling code by taking some unrolling algorithms created for a previous project and porting them to *asopt*. His contributions included, but were not limited to, finalizing the design of the compile and execution-time strategies. Dr. Uh worked on multiple features in our loop unrolling algorithms, such as copying a loop body, making a loop contiguous, and rotating a loop's blocks so that the block with the exit branch is placed at the bottom of the loop. Some of Dr. Uh's code was sourced from a previous project, however most of it was changed by him over the course of this study.

I (Joseph Zilonka) joined the SCALE project before the SCALE ISA simulators were working, so my first task was to find and report bugs in *asopt*. I checked the output by hand until the simulators were ready for use, and tested for runtime errors such as segmentation faults. Since I was new to the codebase, I reported all issues I found to Dr. Whalley and Dr. Uh.

Once the simulators were working, I was tasked with running the SPEC benchmarks through *asopt* to see if the code produced could be used with the simulators. Many issues were encountered at this point, so most of my time was spent reporting bugs. The loop unrolling code proved to be particularly problematic as the benchmarks revealed several edge cases we had not yet addressed. Ultimately, *asopt* underwent many changes before reaching a working state.

When we finally got our optimizer to work properly, we checked the results of the unrolling algorithms and found that only a small amount of loops were being unrolled. The performance improvements were marginal for most benchmarks, with a $\leq 0.1\%$ decrease in total cycles taken after enabling loop unrolling. This motivated Dr. Whalley to research how our unrolling methods could be expanded. He drafted up a set of changes we would make to *asopt* that formed the beginnings of this study and the topic of my thesis.

*asopt* was unrolling fewer loops than expected due to small deviations in the structure of excluded loops. These differences were not recognized by our analysis stage, so it flagged the loops as ineligible for unrolling. The cause of these structural differences turned out to be *gcc*'s method

of converting source code into MIPS assembly. Because *gcc* is generating code intended to run on actual MIPS hardware, it is likely producing code that is optimized for that hardware.

One behavior of *gcc* is that it uses branch on not equal instructions as loop exit branches. Despite some source code explicitly using less/greater than operations, *gcc* translates these exit conditions into *bne* instructions whenever it can. This is probably due to a performance gain on MIPS hardware from using these instructions over branch on less/greater than pseudo instructions, since those expand to two instructions when run through the assembler. However, *seq* (the first of two instructions *asopt* transforms *bne* into) may not work for our execution-time method since it is possible for an unrolled loop's basic induction variable to never match the exit value during execution. Since keeping *seq* instructions could allow unrolled loops to execute past their limit, they needed to be replaced with something else.

Initially, *asopt* did not attempt unrolling loops with *seq* exit conditions because of this problem. We assumed *seq* was infrequent and that we could come back later to add support for loops that used it. However, *gcc* introduces *seq* instructions so often that the amount of loops the execution-time method did not unroll was unacceptably high. Upon finding this out, I was tasked with devising a way to replace all *seq* instructions in these loops.

Dr. Whalley proposed that I change *seq* instructions into *slt*, as *slt* was an exit condition for which *asopt* already had unrolling support. The problem with doing this is that the result of the comparison becomes dependent on the ordering of the arguments. When using *seq*, the result is 0 whenever the arguments do not match. When using *slt*, the result can be 1 or 0 depending on which argument comes first. The sign of the loop's stride also comes into play when the instruction is changed.

Take for example Figure 4.1, which is a simple assembly level loop that is supposed to run for 5 iterations. The *seq* instruction stores 0 in $1 until $2 equals 5, so the branch instruction needed to make this loop work is a *beqz*. In its current state, the loop takes 5 iterations for $2 to reach the value of 5. If we swap the arguments of the *seq*, the loop's behavior does not change and we still execute the correct number of iterations. If we also change the value of $3 to -5 and subsequently change the stride to be -1, the loop's behavior remains correct. This illustrates that *seq* is not dependent on either the ordering of the comparison's arguments or the stride of the loop.

Now let us change the comparison instruction to *slt*. If all else remains the same, the loop is only going to execute one iteration. When the first comparison instruction executes, the *slt* stores the value 1 in register $1. This causes the control flow to fall through the branch, rendering the

```
        move    $2,$0                   # $2 = 0
        li      $3,5                    # $3 = 5
$count_to_five:
        addiu   $2,$2,1                 # $2 += 1
        seq     $1,$2,$3                # $1 = ($2 == $3)
        beqz    $1,$count_to_five       # goto $count_to_five
                                        # if $1 == 0
```

Figure 4.1: An example of an assembly level loop using *seq*.

loop meaningless. The way to fix this situation is to *reverse* (change to its inverse) the branch to *bnez*, so the value of 1 causes the control flow to jump back to the top of the loop. No other changes are necessary, as the loop will run for 5 iterations and stop when $2 equals 5. Figure 4.2 shows how Figure 4.1 would need to change to use *slt* as its comparison instruction.

```
        move    $2,$0                   # $2 = 0
        li      $3,5                    # $3 = 5
$count_to_five:
        addiu   $2,$2,1                 # $2 += 1
        slt     $1,$2,$3                # $1 = ($2 < $3)
        bnez    $1,$count_to_five       # goto $count_to_five
                                        # if $1 != 0
```

Figure 4.2: Figure 4.1 modified to use *slt* instead of *seq*.

If we keep the comparison as *slt* and swap its arguments, we can avoid having to reverse the branch. Because $3 is initially greater than $2, *slt* produces the same result as *seq* every iteration by setting $1 to 0. This does not entirely fix the loop, as the *slt* will also set $1 to 0 when $2 equals 5. That causes the loop to execute 6 iterations instead of 5. We can compensate for this by establishing a new exit value that is 1 less than the original, so the *slt* produces a value of 1 when $2 equals 5.

Finally, if we make the exit value -5 and the stride -1 while keeping the comparison as *slt*, we run into the same situation as before where the loop executes 1 iteration too many. And if we swap the arguments on top of changing those values, we will run into the situation where the loop becomes useless. This demonstrates that the behavior of *slt* is dependent on the ordering of the arguments and on the stride of the loop.

To address these issues when swapping *seq* for *slt*, we assume that the basic induction variable is always the second argument and the exit value is always the third. Before attempting unrolling, *asopt* checks if the exit's branch instruction will continue to work after changing the comparison instruction or if it needs to be reversed. The branch is reversed if necessary, and the optimizer then moves on to checking if the loop will now execute an extra iteration. If it determines the loop will execute an extra iteration, we create a new exit value that is one less than the original.

Figure 4.3 shows how Figure 4.1 would be modified by *asopt* to work with *slt*, an exit value of -5, and a stride of -1. The basic induction variable is already the second argument and the exit value is already the third, so the arguments do not need to be swapped. However, the loop will execute an extra iteration if we do not modify the exit value to be one less than the original. Since $3's value is not used anywhere else, it can safely get changed to -4. The loop will now execute the correct number of iterations, and end up with the value of -5 in $2. If we did not want to change $3's value, we could instead substitute the *slt* for an *slti* instruction and have -4 be the third argument.

```
        move    $2,$0                # $2 = 0
        li      $3,-4                # $3 = -4
$count_to_five:
        addiu   $2,$2,-1             # $2 += -1
        slt     $1,$2,$3             # $1 = ($2 < $3)
        beqz    $1,$count_to_five    # goto $count_to_five
                                     # if $1 != 0
```

Figure 4.3: Figure 4.1 modified to use *slt*, have a negative exit value, and have a negative stride.

These changes would suffice if *gcc* followed our assumption about the ordering of the comparison instruction's arguments. Unfortunately, it has no rule about how *seq*'s arguments are ordered likely due to the commutative property of the equality operator. Once we found this out, we made it a rule in *asopt* to always order the basic induction variable as the second argument and the limit as the third. During unrolling, we check the ordering of the arguments in the exit condition's comparison instruction and swap them if necessary. The stride is then checked to ensure that the branch instruction will still work when the comparison is modified.

Another behavior of *gcc* is that it sometimes relocate values from their original registers. A relocation happens when a register containing an important value will be overwritten by a future

section of code, so the value is preserved by copying it to another register. This is less of an optimization and more of a consequence of the MIPS calling convention, since a relocation usually happens when a special register needs to be freed. The issue with relocations is that they can introduce complexity into a compiler's optimization algorithms, as many optimizations need to keep track of where values are set and used. Loop unrolling is among these optimizations because it has to find where the initial and exit values are set.

The obvious location to start looking for the initial and exit values is the loop's exit instructions. We can then scan backward through the code to find where the two registers are first set, and retrieve any constant values. The problem is that a constant might be more than one instruction removed from the instruction that sets a register used in the loop. While a human can easily scan the code and see that a constant value has been relocated, an analysis stage must be programmed to expect this and preemptively search backward.

*asopt* originally did not search for values that appear before relocation instances. If a relocation set the initial or exit value register, the value was marked as not known at compile time. Because our compile-time method requires both values to be constant, loops with a known value and a relocated value were being unrolled using the execution-time method. Unrolling with this method is still a potential improvement over not unrolling at all, and the performance difference of choosing one method over the other is minimal. For classification purposes, however, the loop is a compile-time number of iterations loop and should be marked as such. So, we decided to modify our optimizer to look backward and search for constant values.

A large percentage of these relocations were only one instance away from the constant value, so I initially added a quick fix to search backward one relocation for a value. As we looked over more loops, it dawned on us that we needed a more robust solution to track down values that were previously set. Dr. Whalley had already written code that did most of the necessary work, so I adapted it for our implementation of loop unrolling. This replaced my initial quick fix.

One particular benefit brought by Dr. Whalley's solution was the ability to go backward through transfers of control to find where values were coming from. My quick fix could only go back through sequential blocks, which would give incorrect results if the blocks did not sequentially execute. An example of this is having two consecutive blocks, where the first one ends with an unconditional jump to a later section of code. The second block would never be accessed directly after the first one got executed, so searching for a value going up from the second block will skip checking any intervening blocks.

In addition, this change allowed us to expand the set of instructions for which *asopt* could search through to find values. My fix only worked for *move* instructions and assumed that *addiu* and *li* (load immediate) were the only instructions setting registers to constants. The new solution added support for instructions like *ori* (or immediate), *andi* (and immediate), *sll* (shift left logical), and others that could also relocate constants to registers.

These were the most significant changes we made to *asopt* to work around some of the obstacles introduced by *gcc*. There are other edge cases for which we added solutions, but they are so small and appear so rarely that it is not worth describing them in this thesis. Our fixes increased the amount of loops we were able to unroll, and gave us some insight as to how *gcc* generates MIPS assembly.

Despite our considerable progress thus far, we were still not done increasing the scope of loops that could be unrolled. A particular kind of loop ignored by *asopt* had been on our radar since the beginning of the study, and after some of the *gcc* edge cases were resolved we decided it was time to tackle unrolling those loops. Up to that point, *asopt* could only unroll loops that had one exit condition. This covers many of the loops we encounter in the benchmarks, but not all of them. The loops we were ignoring had multiple exit conditions, or more than one instruction that may terminate the loop.

Loops with multiple exits can easily be unrolled by only eliminating one exit branch in the loop. If a single exit meets the requirements for unrolling, we can treat the others as just part of the loop's body and leave them intact. Because they will be replicated along with the rest of the loop's body, unrolling will not add or remove any necessary comparison and branch pairs. If we try to unroll more than one exit, we might run into problems when a basic induction variable is shared among exits. We could also run into some issues if basic blocks shared by multiple exits are replicated more times than intended.

The decision not to unroll these loops was due to the limitations of our initial unrolling implementation. All of our code assumed there would only be one loop exit, so processing a loop with multiple exits could produce unexpected results. This might miss out on loops whose first exit cannot be eliminated yet a later one can. However, it could also break our analysis stage entirely, and require everything to be re-engineered. Consequently, our analysis stage automatically rejected loops with more than one exit. This prevented multiple exit loops from ever reaching the unrolling code or Dr. Uh's code, so those sections of *asopt* were particularly vulnerable to bugs.

To reduce the possibility of having to rewrite everything while adding support for multiple exits, I redesigned the analysis stage. It now starts off by finding all exits in the loop and putting them into a list. It then linearly checks each exit to see if it meets the criteria for unrolling. When an exit does not meet the criteria, the analysis will mark the loop as not unrollable and move on to the next exit. When it finds an exit that can be used for unrolling, it immediately stops and attempts to unroll the loop using that exit. If none of the exits meet the criteria, then it can either give up on unrolling or decide to use naive unrolling if enabled. In this latter case, the analysis stage will use the first exit in the list to unroll the loop.

The benefit from changing the analysis stage in this way was that I did not have to update the unrolling code or any of Dr. Uh's code. Those two sections of *asopt* are still expecting a single exit to be passed to them, which is sufficient if we only want to eliminate one exit per loop. All the analysis stage has to do is select an exit, and tell the other sections of code that the chosen exit is the loop's "main" exit. The other sections will then do the work that needs to be done to that exit, and leave the other exits alone.

This implementation works well for both single exit loops and multiple exit loops. If a loop has one exit, then it is the only exit in the list and it alone determines whether the loop can be unrolled. When a loop has multiple exits, then this process will filter out exits that cannot be used and only attempt to eliminate an exit that is safe to remove. If more than one exit could be eliminated by unrolling, our implementation picks the first one it encounters, ensuring that unrolling is always attempted on the loop.

# CHAPTER 5

# RESULTS

The set of loops we analyze in this chapter comes from the SPEC 06 CPU benchmark suite. We ran our optimizer on all the benchmarks in the suite and collected various results about the characteristics of their innermost loops. We then ran the resulting assembly code through three ADL SCALE simulators to see how a benchmark's performance changes after using our loop unrolling strategies.

## 5.1 Loop characteristics

*asopt* produces various statistics on all loops in an assembly file as the code is being optimized. For this study, we collected the statistics of each benchmark's assembly files and merged them into one comprehensive report for their respective benchmark.

The optimization setting we chose to collect this data with was *u2*, which tells *asopt* to use both the compile-time and execution-time methods to unroll with an unroll factor of 2. We imagine this would be a very common choice, and thus an appropriate setting to give us a general idea of the state of the loops in the benchmarks seeing as the choice of unroll factor and strategy greatly impacts the results obtained by *asopt*.

Consider Figures 5.1 and 5.2. *434.zeusmp* and *470.lbm* stand out from the other benchmarks in that they have a higher average number of instructions per loop. *470.lbm* in particular has a very large amount of instructions in its loops, while also having the smallest number of innermost loops out of all the benchmarks. This could be significant because of the fact that loop unrolling increases code size in exchange for potentially better performance. With programs that have a lot of small loops (such as *403.gcc*), this trade off may be worth it since each unrolled loop can provide a performance improvement to the overall application. In the case of *470.lbm*, because it has so few loops, unrolling would greatly increase code size for a potentially marginal benefit.

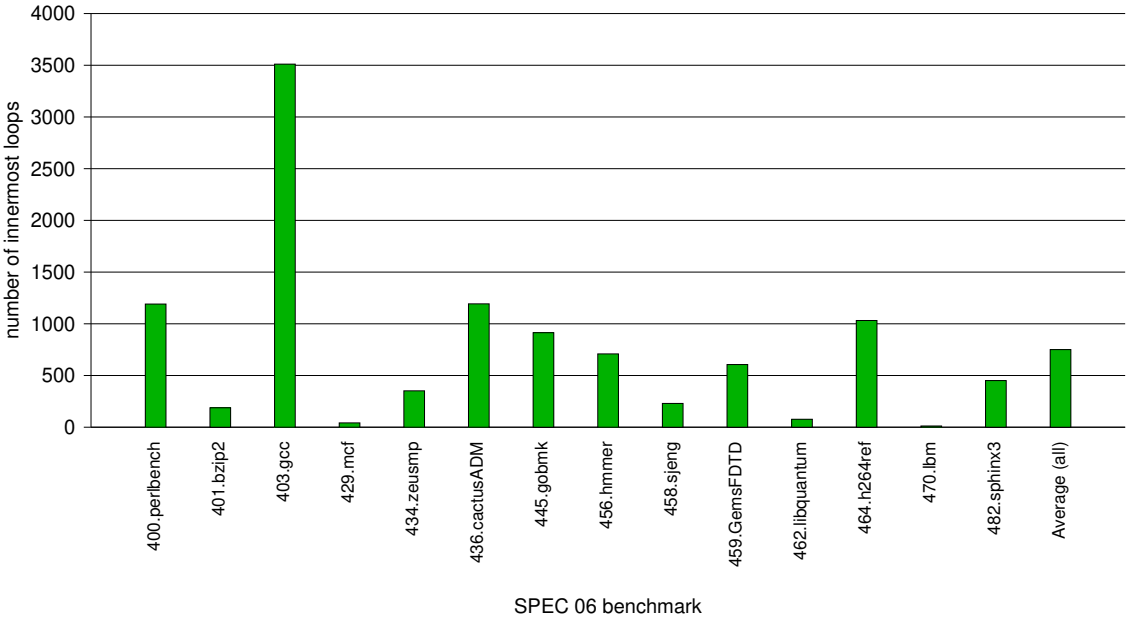



Figure 5.1: Number of innermost loops in each benchmark.

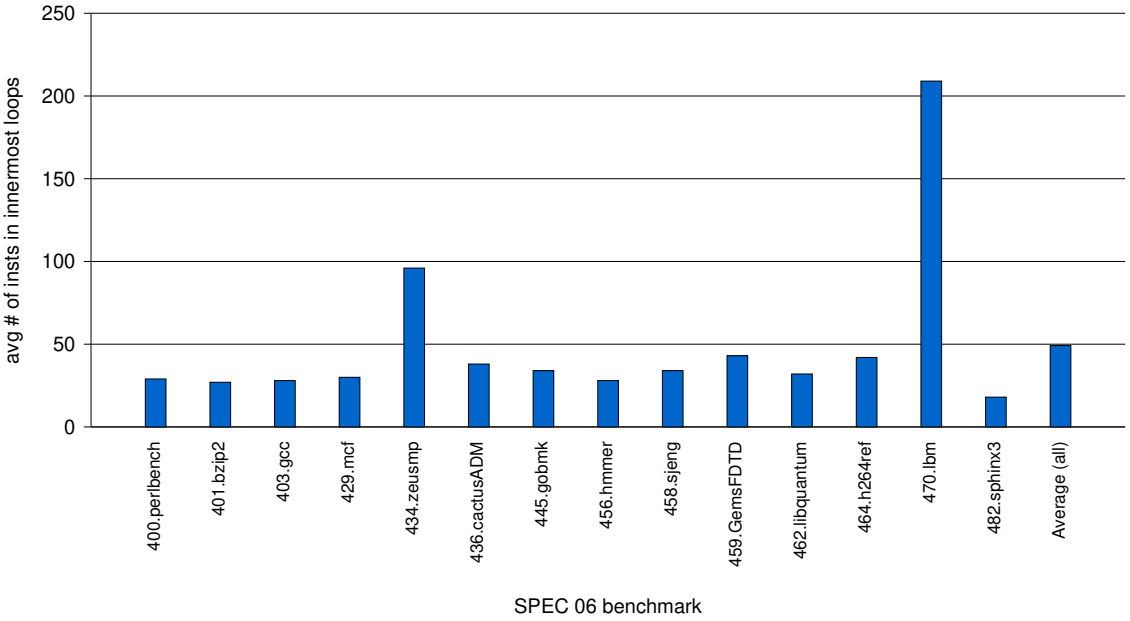



Figure 5.2: Average number of instructions per innermost loop.

The next set of results are represented as a percentage of the total number of innermost loops, as opposed to an absolute number. Figure 5.3 shows the percentage of innermost loops containing a constant initial value, a constant exit value, and a constant stride. Overall, there seems to be a small amount of loops that contain all three values. Our compile-time unrolling strategy requires all three of these values to be constant, so these results represent a maximum percentage of loops that could be unrolled using that method. For benchmarks like *470.lbm* and *445.gobmk*, we expect the compile-time unrolling strategy to deliver some performance benefit after unrolling. It should be noted that while these loops meet the criteria for unrolling with the compile-time method, they may fail our unrolling checks at some other step. Thus Figure 5.3 does not represent the exact percentage of loops successfully unrolled using the compile-time strategy.

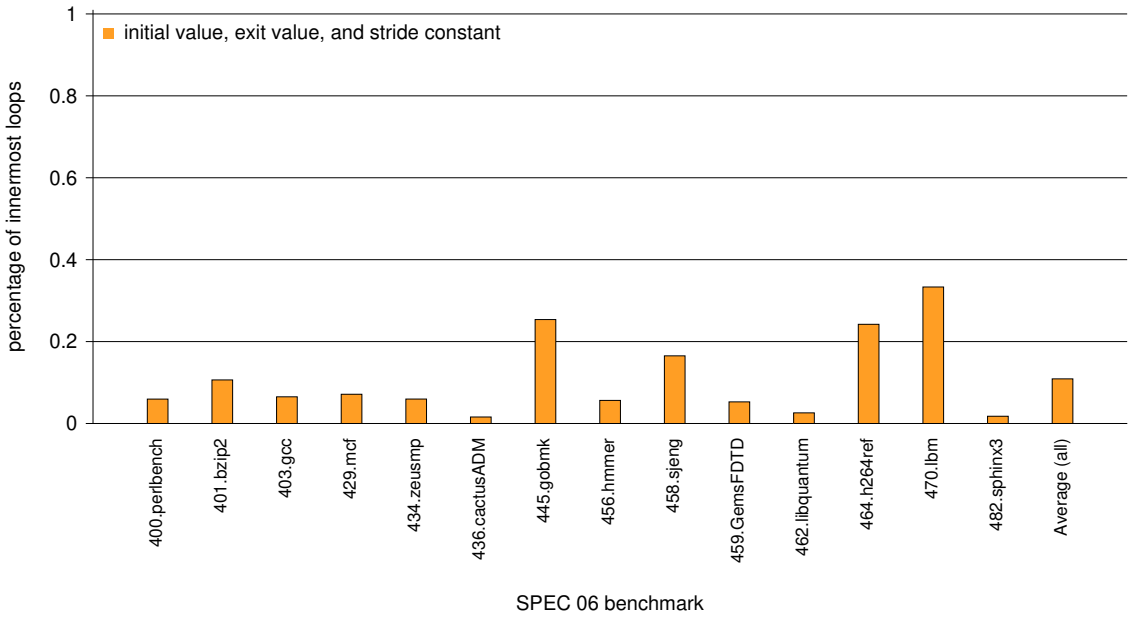



Figure 5.3: Percentage of innermost loops with a constant initial value, a constant exit value, and a constant stride.

The left bar of each benchmark in Figure 5.4 represents how *asopt* classified a given loop during the unrolling process in terms of number of iterations. Unlike the compile and execution-time classifications, which signify *asopt* unrolled those loops using one of these two strategies, loops in the "unknown number of iterations" category are not guaranteed to have undergone naive unrolling. In theory, naive unrolling should be able to unroll all loops, not excluding those under the "unknown number" category. However, there is a small number of loops that still do not get naively unrolled due to corner cases not handled by our optimizer.

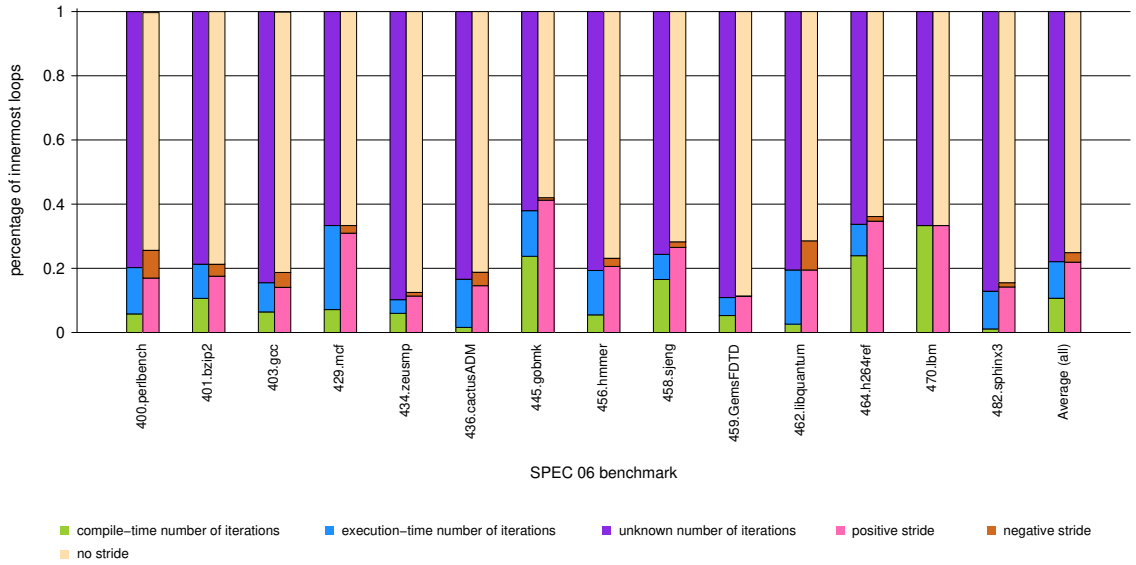



Figure 5.4: Classifications of number of iterations, and different kinds of strides.

The percentage of innermost loops that are marked as having an "unknown number" of iterations is significant. Without naive loop unrolling, there would be no way to potentially gain performance from these loops since applying the compile and execution-time methods requires knowing the number of iterations in advance (or that the number of iterations is some constant).

The right bar of each benchmark, which shows the percentage of a type of stride present in that benchmark's loops, also reflects the usefulness of naive unrolling seeing as the majority of loops have no detectable stride. The compile and execution-time methods require a known stride to unroll a loop, so these loops could not be unrolled without naive unrolling. Having a known stride enables us to calculate the number of iterations a loop will run, so there is a large overlap where loops with no stride iterate for an unknown number of times.

Interestingly, positive strides constitute the majority of loops that have a known stride. This makes supporting negative strides less important as the number of loops we could unroll that have negative strides is likely small. During this study, we added support for unrolling negative strides to our optimizer in the hopes that a larger portion of loops would be unrolled. While this change did allow more loops to be unrolled, it did not have much impact on the benchmarks as loops with negative strides are relatively rare.

Figure 5.5 shows whether there is more than one exit on a benchmark's left bar and whether the exit value is invariant on the right bar. Statistically, innermost loops are likely to have a single exit, and this trend holds true in all tested benchmarks. Some benchmarks, such as *400.perlbench*,

still have a relatively high number of loops with multiple exits. Because our optimizer could not unroll loops with multiple exits prior to this study, we ignored many potential opportunities for unrolling. Unlike supporting negative strides, adding support for multiple exit loops seems to have been a meaningful optimization to make due to the large number of loops we would have missed had we not done so.

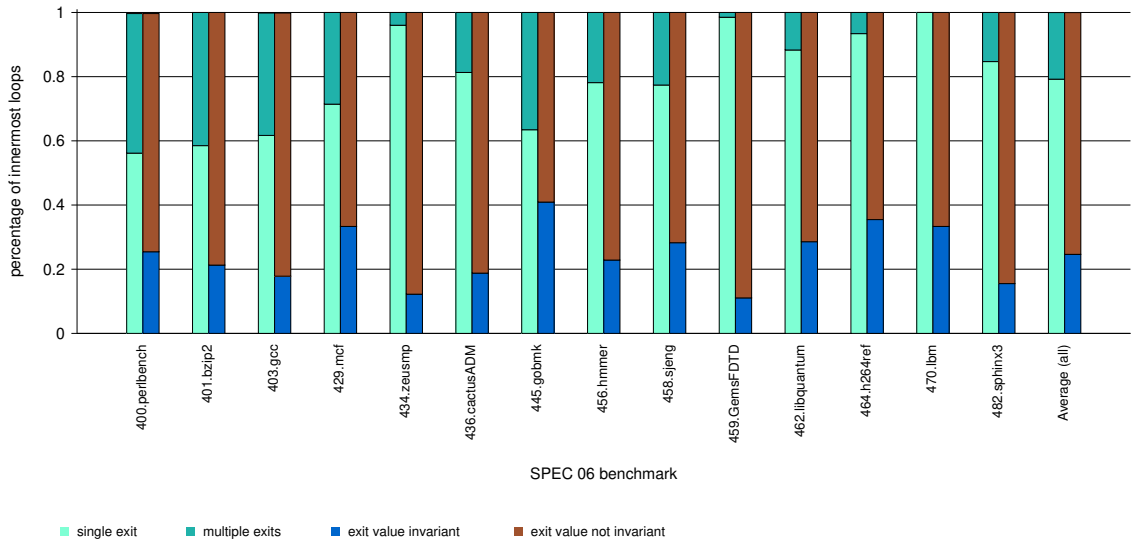



Figure 5.5: Percentage of loop exit amounts and whether an exit value is invariant.

Unfortunately, the majority of loops seem to have non-invariant exit values. This is a problem for the compile and execution-time unrolling strategies, as they need a loop's exit value to be invariant in order for them to unroll the loop. Naive unrolling is a good solution to this problem, since it does not care whether an exit value is invariant. Something to note is that, if an exit value is not known (usually due to it being loaded from a memory location during the loop), we mark it as "not invariant," which slightly inflates that number. It might be possible to add unrolling support for loops where an exit value's memory location is loop invariant, but we leave that for future work.

Finally, we analyze the "unrolling statuses" *asopt* provides after a file is processed. When a loop is detected by the optimizer and unrolling is enabled, *asopt* will always attempt to unroll it. If unrolling is successful, then the unroll status of the loop is set to "unrolled." If it is not successful, the status is set to a corresponding reason depending on which point in the unrolling process the optimizer encountered an issue. These statuses can be specific, such as "no basic induction variable in exit," while others are more general and encompass a variety of related issues. Something to

note is that a loop could be unfit for unrolling for more than one reason. In those cases, the reason that gets assigned is the first one *asopt* encounters.

The hierarchy of reasons that can be assigned is shown from bottom to top in the bars of Figure 5.6, excluding the "unrolled" reason. What this means is that "irreducible" can get assigned first, followed by "improper exit value," "problem with branch over loop," and so on. If a reason other than "unrolled" is assigned, *asopt* gives up on unrolling the loop unless naive unrolling is enabled.

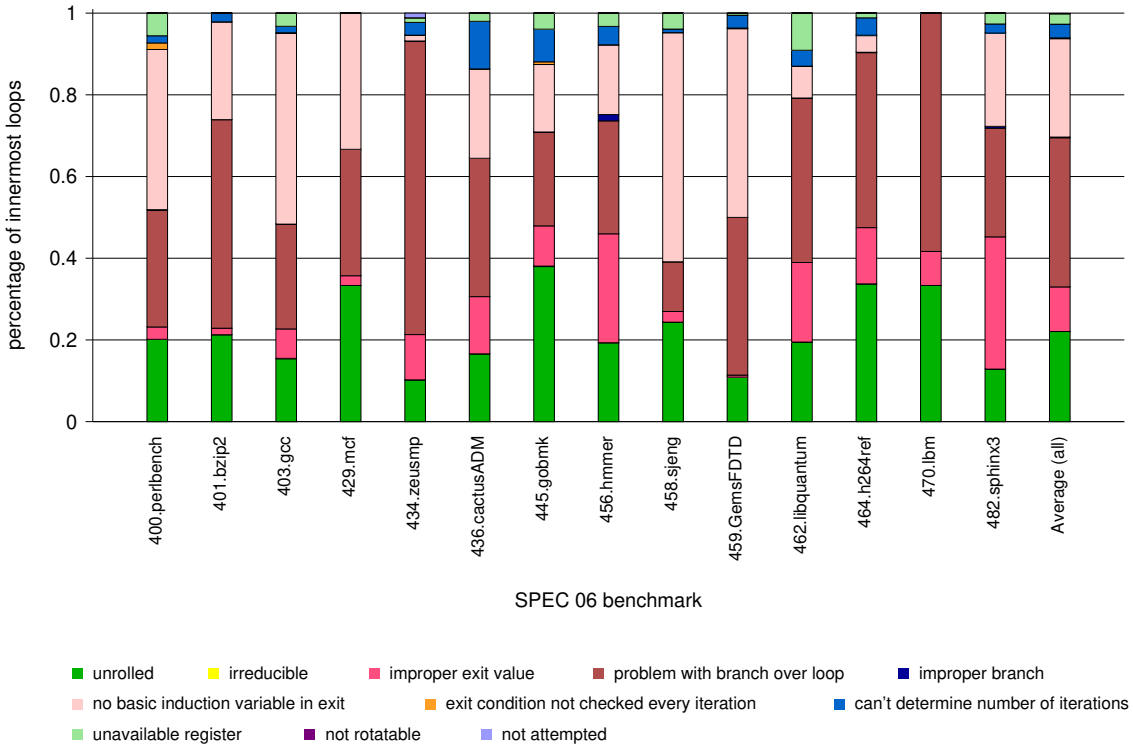



Figure 5.6: Percentage of unroll statuses after loop unrolling.

Figure 5.6 shows the aggregated percentages of these statuses for the innermost loops of each benchmark. On average, around 20% of loops are successfully unrolled while the rest run into issues. By far the largest status encountered when unrolling is the "problem with branch over loop" status. We will discuss what this status means after we briefly overview the other statuses.

The "irreducible" status is necessary since *asopt* can only unroll natural loops with a single entry point (the loop header). A reducible loop is a loop where the edges can be partitioned between backedges and the remaining edges form a directed acyclic graph (DAG). We check if the blocks in the innermost loop are reducible, and if they are not we assign the status of "irreducible".

"No basic induction variable in exit," "exit condition not checked every iteration," and "can't determine number of iterations" are self-explanatory. "Unavailable register" gets assigned when we have run out of registers and we need to allocate a new one. "Not rotatable" and "not attempted" get assigned when the code that makes the loop body contiguous or the code that attempts to rotate a loop run into a corner case we have not yet handled. Most of these issues have been resolved, which is why they rarely show up.

"Improper exit value" is assigned when an exit value is not invariant and not explicitly constant. "Improper branch" gets assigned when *asopt* could not find a valid relational operator (less than, greater than, etc.) in the loop's exit branch or the loop's exit comparison instruction.

"Problem with branch over loop" is the most complicated unroll status. It has become a catch-all we use for a variety of statuses that share the same underlying problem. The issue boils down to an arbitrary restriction we had in *asopt* that required it to check for a branch instruction that branches past the loop. Issues such as a loop not having a preheader, a loop preheader having multiple predecessors, not being able to find an initial or exit value in the branch over the loop instruction, or not finding a branch over the loop instruction at all are tied to this assumption. If we removed the need for this assumption, those issues would no longer be important and could safely be ignored by our optimizer. Removing this restriction will be a significant undertaking, so we decided not to do this for the current study. We expect this restriction will be removed in future work. Because it is the most common unroll status, successfully removing it is a priority and doing so should increase the number of successfully unrolled loops.

Because these statistics were collected using *u2*, Figure 5.6 will look very different when naive loop unrolling is enabled. Most of the unroll statuses would change into "unrolled" since naive unrolling is not bound by the same restrictions as the compile and execution-time strategies. The only statuses that will remain are "irreducible," "not rotatable," and "not attempted".

## 5.2 Dynamic results

Here we discuss the performance results obtained from using our unrolling strategies on the SPEC benchmarks. We are interested in how many cycles a simulation of a benchmark takes, and how the unrolled version of a benchmark stacks up to the non-unrolled version. Our results are presented as a percentage of cycles relative to the amount of cycles taken to run the non-unrolled instance, since each benchmark can vary greatly in terms of absolute cycle counts. Since we must

run the benchmarks through the optimizer to switch to the SCALE ISA, our baseline metric will be running the benchmark through the optimizer with no unrolling enabled.

There were three different sets of unrolling strategies under test, and each combination was tested using unroll factors of 2 and 4. The first set was to only unroll loops that could be unrolled with the compile-time method. The second set was to unroll using both the compile and execution-time methods, with the optimizer attempting the compile-time method first. The final set was to unroll using the compile-time method, the execution-time method, and naive unrolling (in this order).

All three sets of tests were run by passing the *gcc*-compiled MIPS assembly files from a benchmark through our optimizer to produce optimized code. The code was first changed from MIPS to SCALE, and then was unrolled with the respective unroll factor and strategies for each test. If there were any specific minor optimizations that could be done (or in the case of the VLIW simulator, VLIW instruction scheduling), those were performed after unrolling was completed. This process of running files through the optimizer was also done for the simulator's system library files, so that all code under test had been run through the optimizer.

In the interest of time, we used the *test* input set for running the benchmarks. While this is not the preferred method of running SPEC, it was sufficient for our study to get a general idea of how *asopt*'s unrolling algorithms affect the performance of the benchmarks. We anticipate the performance of loop unrolling will improve further using the *reference* input set, as loops in that set will contain more iterations.

Figures 5.7 and 5.8 show the performance results from running the optimized code on the SCALE pipeline simulator. This simulator represents a standard 5-stage pipeline SCALE processor. In general, the benchmarks required fewer cycles to complete after being unrolled with an unroll factor of 4. Combining all three methods produces the best average results for an unroll factor of 4, while only using the compile and execution-time methods gives the best average results for an unroll factor of 2.

*456.hmmer* seems to have benefited a lot from naive unrolling with an unroll factor of 4, given the large difference between only using the compile and execution-time strategies and combining them with naive unrolling. *464.h264ref*, *458.sjeng*, and *429.mcf* benefited the most from any kind of unrolling compared to the other benchmarks. On the other hand, *470.lbm* and *436.cactusADM* seemed to reap no performance gains from any of the unrolling strategies, and *462.libquantum* had no change when using an unroll factor of 4.

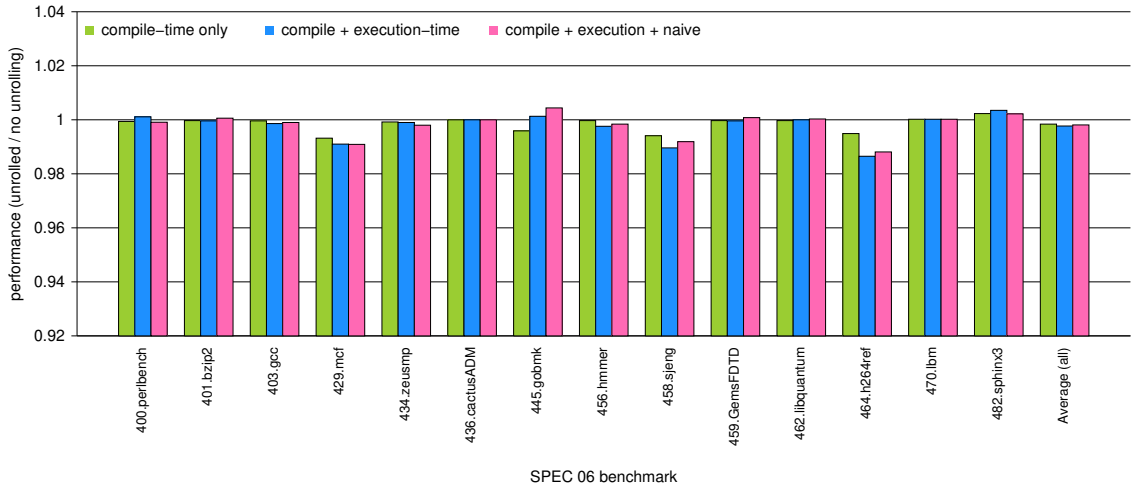



Figure 5.7: Performance results from the SCALE pipeline simulator after unrolling with *asopt* using an unroll factor of 2.

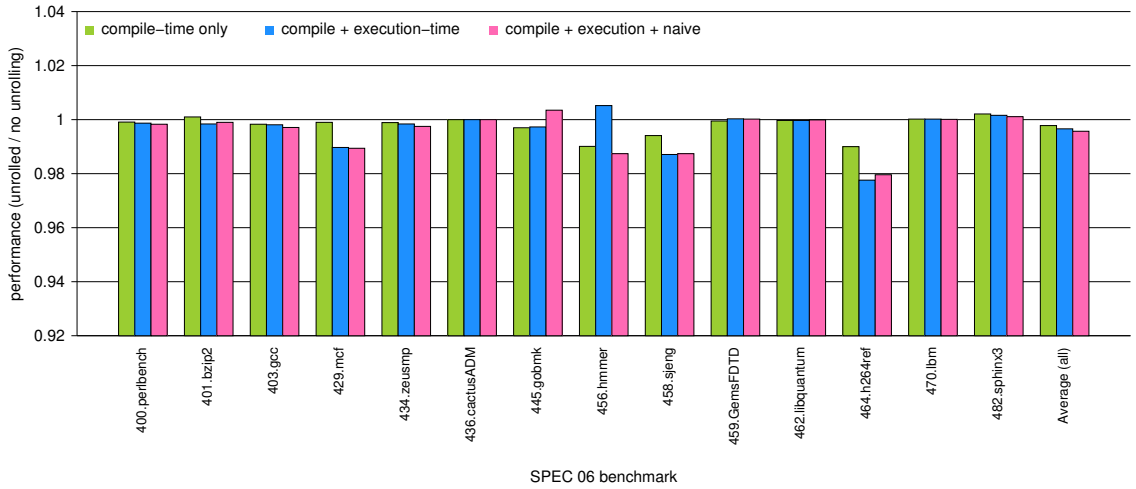



Figure 5.8: Performance results from the SCALE pipeline simulator after unrolling with *asopt* using an unroll factor of 4.

Figures 5.9 and 5.10 show the results from running the same code through the SCALE superscalar simulator. The superscalar simulator is a 4-issue superscalar SCALE processor that does not use memory speculation.

On average, performance is worse using our unrolling techniques on the superscalar processor. Certain benchmarks still get benefits from it, such as *464.h264ref* and *458.sjeng*, but *429.mcf* and *456.hmmer* are now considerably worse after unrolling using either an unroll factor of 2 or 4. While mostly beneficial when used on the pipeline simulator, naive unrolling seems to make performance

worse on the superscalar simulator. This result makes sense if the loops that were naively unrolled are loading and storing values to memory, as instructions dependent on those values cannot be executed in parallel and must stall the CPU. The effect of naively unrolling these loops would then cause more instruction cache misses, seeing as instructions would need to be fetched again once the necessary data is available.

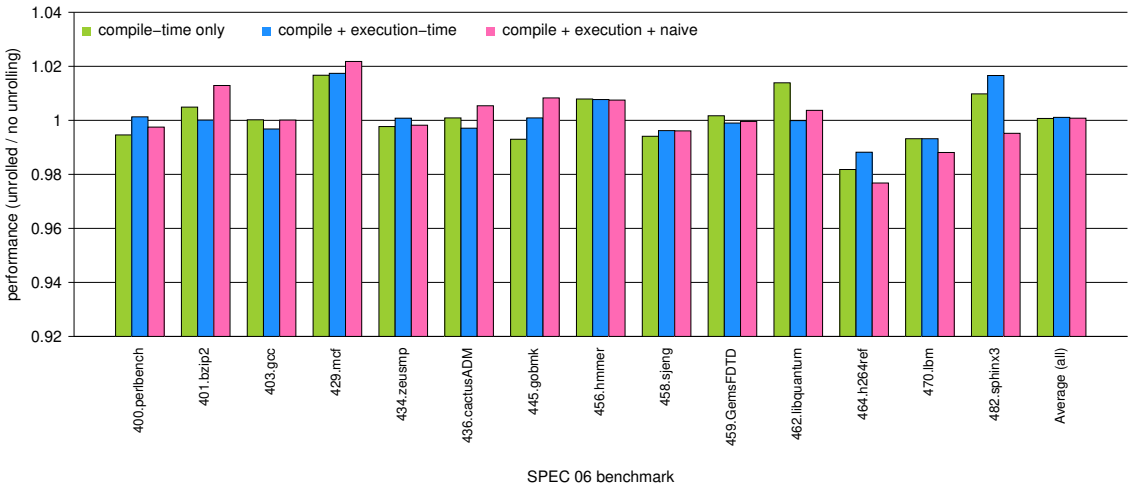



Figure 5.9: Performance results from the SCALE superscalar simulator after unrolling with *asopt* using an unroll factor of 2.

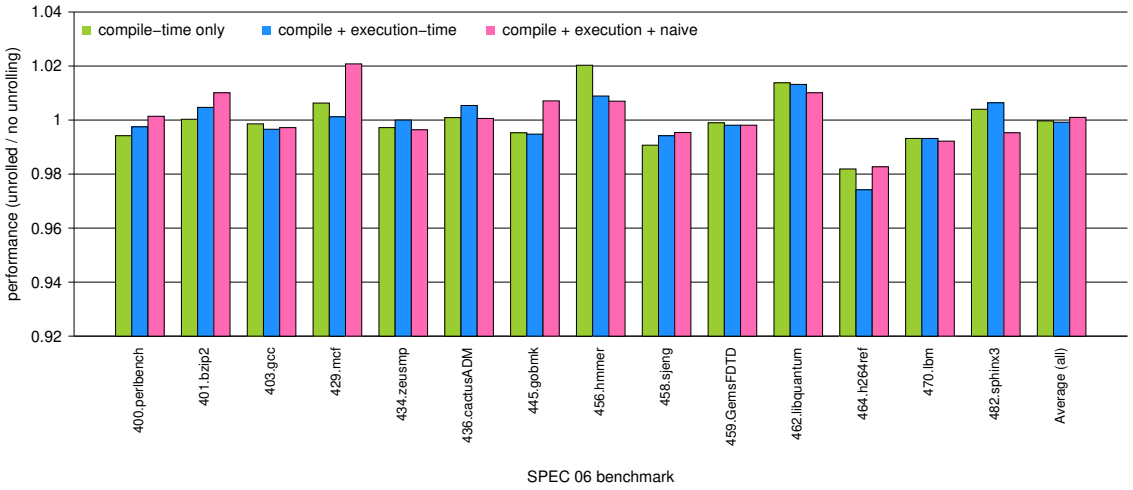



Figure 5.10: Performance results from the SCALE superscalar simulator after unrolling with *asopt* using an unroll factor of 4.

Lastly, Figures 5.11 and 5.12 show the results after converting the code into the SCALE VLIW ISA. The VLIW simulator we used simulates a 4-wide SCALE VLIW processor with no lane

restrictions on memory operations. This means that a memory instruction can go in any of the 4 lanes of a VLIW pack. The code is first unrolled by *asopt*, similar to how the code for the previous two benchmarks is produced. After unrolling, the code is converted into VLIW packs and VLIW scheduling is performed. The results we observe from this simulator are influenced by both our unrolling algorithms and any scheduling optimizations done by the VLIW scheduler.

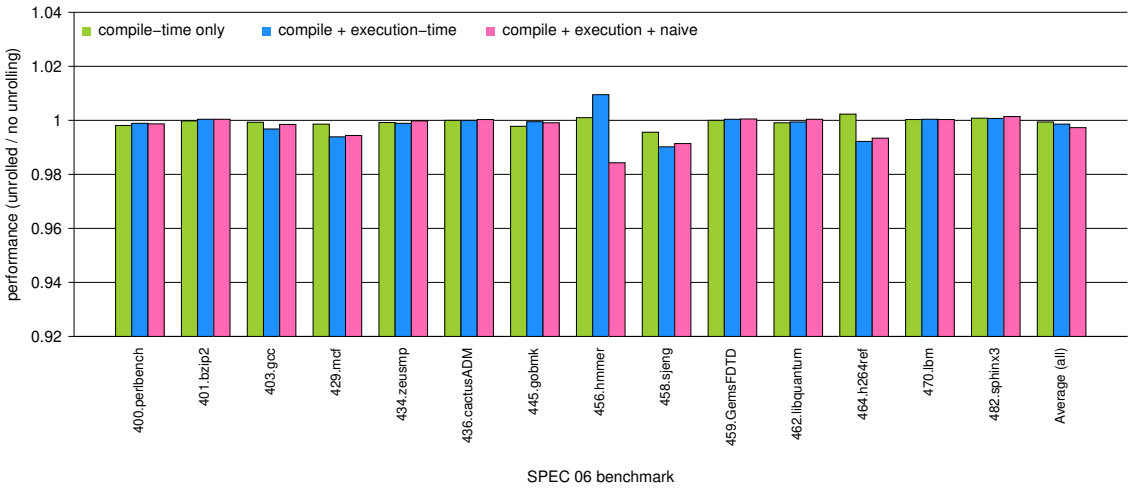



Figure 5.11: Performance results from the SCALE VLIW simulator after unrolling with *asopt* using an unroll factor of 2.

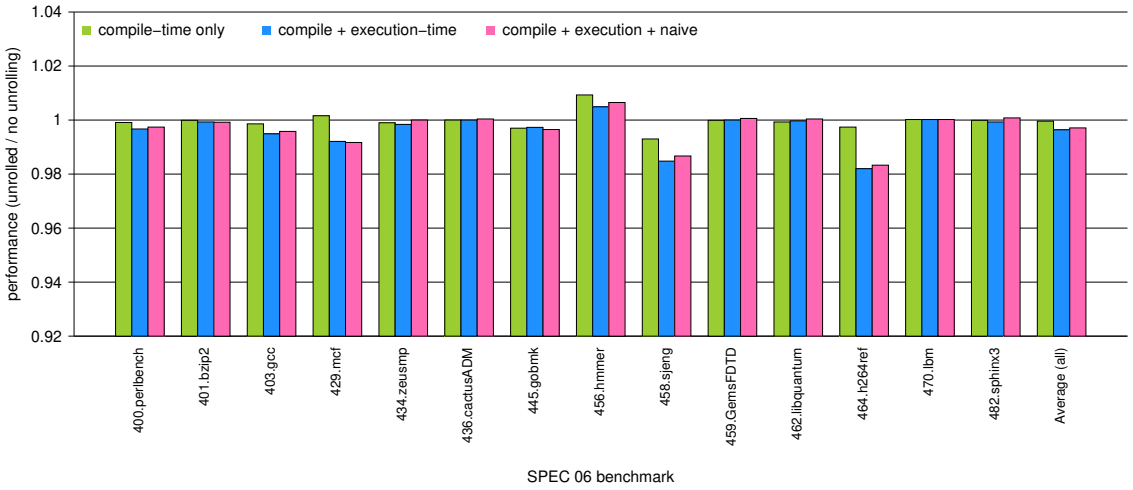



Figure 5.12: Performance results from the SCALE VLIW simulator after unrolling with *asopt* using an unroll factor of 4.

The results here are much less erratic than the superscalar simulator, and akin to the results of the pipeline simulator. On average, the benchmarks perform slightly better than the baseline

for both unroll factors. Like with the pipeline simulator, the inclusion of naive unrolling improved performance in some of the benchmarks.

*456.hmmer* has the worst performance out of all the benchmarks, with increased cycle counts after using all combinations of unrolling strategies with an unroll factor of 4. Peculiarly, this was not the case when we unrolled *456.hmmer* using all three strategies with an unroll factor of 2. In fact, that result is the best performance result from Figure 5.11, which is interesting considering its compile and execution-time result is the worst in the figure.

*464.h264ref* and *458.sjeng* yet again enjoy performance benefits for the most part, with the exception of *464.h264ref*'s compile-time only result using an unroll factor of 2.

# CHAPTER 6

# RELATED WORKS

There have been several studies done on loop unrolling as this optimization has been known for a number of decades. Most of these studies performed loop unrolling at a high level, typically at the source or intermediate code levels.

[1] studied more aggressive loop unrolling techniques and their impacts on application performance. They used similar compile and execution-time methods in their study, and they also collected characteristics of unrolled innermost loops. Where our study differs from [1] is the exploration of naive loop unrolling and our work at the assembly level. Naive unrolling has been done before in some compilers. While the compile and execution-time methods are similar conceptually, they differ in their implementations as our optimizer only optimizes assembly level code.

[2] studied how loop unrolling can provide performance benefits when paired with static or dynamic disambiguation of memory references. Loop unrolling is not the main focus of that study, as they are interested in how loop optimizations in general interact with dynamic disambiguation of memory references. Our study is entirely focused on loop unrolling, and like mentioned previously features naive loop unrolling and optimizations at the assembly level.

[3] explored the basics of unrolling loops in Fortran. They looked into how unrolling at the source code level can be beneficial to performance and tested their ideas on a variety of Fortran compilers. They are interested in obtaining performance gains without touching assembly code, which is the opposite approach of this study.

[7] explored automatically selecting useful unroll factors and generating compact code for those unroll factors. Their contributions include a new code generation algorithm for unrolling nested loops, and a new algorithm that can efficiently enumerate feasible unroll vectors. Our study does not discuss the selection of unroll factors, as we leave that up to the user. We have also developed our own new code generation algorithm for unrolling innermost loops in the form of naive loop unrolling.

[8] studied using AI techniques to predict the best loop unroll factors for different loops. They demonstrate how supervised learning techniques can determine the appropriateness of unrolling a program's loops. Like mentioned previously, we leave the choice of unroll factor up to the user. We

also leave the choice of whether to unroll loops up to an application writer by providing statistics about that application's loops.

Finally, [5] studied re-applying loop unrolling on already unrolled loops to determine the impact on other optimizations. While similar to this study in attempting to obtain optimal code after the compilation process, it relies on an iterative approach to find an optimal way to unroll a loop. We do not use an iterative approach, as our optimizer only attempts unrolling once on each loop in an input file. They also focus on the phase-ordering problem at the compiler level, which is a concern for our optimizer but somewhat different in that phase ordering might be more flexible at the assembly level.

# CHAPTER 7

# CONCLUSIONS

The results show that *asopt*'s unrolling strategies are a promising start. As discussed throughout the study, our optimizer falls short when attempting to unroll specific configurations of loops that would otherwise be unrollable. This was due to incorrect assumptions we made about our unrolling criteria and difficulties in implementing the necessary logic to handle various corner cases. While the current optimizer can successfully unroll many loops, more can be done to expand our strategies to cover more loops.

In particular, the "problem with branch over loop" unroll status is our next focus for future work. There are still many loops we should be able to unroll that get flagged with this status, leaving potential performance benefits on the table. While we know it is possible to unroll these loops, getting the optimizer to the point where it can actually unroll them will be a challenge. The assumptions we made about unrolling criteria are hard-coded into various parts of the unrolling algorithms, and it will take a significant amount of re-engineering to remove those criteria from the code.

As far as loop characteristics are concerned, *asopt*'s statistic-collection abilities give us a unique look into the state of SPEC 06's innermost loops. Raw performance metrics are important to measure, but they can take a long time to collect. Code characteristics such as average number of instructions per loop and whether an exit value is invariant can give us fast insight into a program's structure without needing to run the code. This can quickly inform application writers what optimizations would be beneficial to their code, allowing them to bypass the potentially costly process of running their applications to completion.

As we expected, the effectiveness of unrolling varies by benchmark. Benchmarks such as *464.h264ref* had performance improvements virtually across the board, while other benchmarks like *459.GemsFDTD* saw little to no improvement. The types of loops in a benchmark, as well as the overall quantity of them, are better predictors of potential performance improvements and should be considered before attempting to unroll the loops in an application.

The most surprising result was the effectiveness of naive loop unrolling. While not a typical method of unrolling loops, most simulations run with naively unrolled code performed better than

the baseline. This is a promising result since, without naive unrolling, there would be no way to unroll a large class of loops. The fact that naive unrolling gave us a positive result at all demonstrates that there is still performance to be gained from unrolling loops in certain applications. We do not suggest blindly running naive unrolling on all code, as it might still degrade performance depending on the application. If a custom approach is taken where naive unrolling is exclusively used on loops whose performance is known to improve against their baseline's, we can reduce an application's cycle count with minimal effort.

We conclude by stating that our compile-time, execution-time, and naive unrolling strategies definitively improve performance for a sizeable amount of applications. While not every application will benefit from our strategies (or loop unrolling in general), our results suggest it is worthwhile to run applications through *asopt* to check if their code could be improved. We believe there is still plenty of room to expand the scope of our unrolling algorithms, and that the results seen in this study will only improve with the further development of *asopt*.

# BIBLIOGRAPHY


[1] Jack W. Davidson and Sanjay Jinturkar. *An Aggressive Approach to Loop Unrolling*. Tech. rep. USA, 1995, p. 36.

[2] Jack W. Davidson and Sanjay Jinturkar. "Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation." In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 1995, pp. 125–132. DOI: 10.1109/MICRO.1995.476820.

[3] J. J. Dongarra and A. R. Hinds. "Unrolling loops in Fortran." In: *Software: Practice and Experience* 9.3 (1979), pp. 219–226. DOI: https://doi.org/10.1002/spe.4380090307.

[4] Arthur Karapateas. "Retargeting an Assembly Optimizer for the MIPS/SCALE Assembly Language." MA thesis. Tallahassee, FL: Florida State University, July 2021.

[5] Nicholas Nethercote, Doug Burger, and Kathryn McKinley. "Convergent Compilation Applied to Loop Unrolling." In: *T. HiPEAC* 1 (Jan. 2007), pp. 140–158. DOI: 10.1007/978-3-540-71528-3_10.

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

[7] Vivek Sarkar. "Optimized Unrolling of Nested Loops." In: *Proceedings of the 14th International Conference on Supercomputing*. ICS '00. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2000, pp. 153–166. ISBN: 1581132700. URL: https://doi.org/10.1145/335231.335246.

[8] M. Stephenson and S. Amarasinghe. "Predicting unroll factors using supervised classification." In: *International Symposium on Code Generation and Optimization*. 2005, pp. 123–134. DOI: 10.1109/CGO.2005.29.

# BIOGRAPHICAL SKETCH

Joseph Zilonka is a Masters student studying Computer Science at Florida State University. Zilonka received his Bachelor of Science in Computer Science from FSU in 2020, and will be working as a RISC-V Compiler Developer for Advanced Micro Devices (AMD) starting in August 2022. His time as a Masters student was spent researching compilers and compiler optimizations under the supervision of Dr. David Whalley and Dr. Gang-Ryung Uh, and contributing to their research in the NSF SCALE project. He also held a position as a Teaching Assistant for Professor Robert Myers in FSU's Object Oriented Programming (OOP) C++ class. Before his time at FSU and throughout his studies, Zilonka held technology focused internship positions at Carmel 6000 and Donna Klein Jewish Academy.