THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS & SCIENCES

IMPLEMENTATION OF A POOR MAN'S TRACE CACHE IN AN ASSEMBLY OPTIMIZER

By

SKYLAR SCORCA

A Thesis submitted to the Department of
Computer Science
in partial fulfillment of the requirements for graduation with
Honors in the Major

Degree Awarded:
Summer, 2023

The members of the Defense Committee approve the thesis of Skylar Scorca defended on July 27, 2023.

Dr. David Whalley
Thesis Director

Dr. Linda DeBrunner
Outside Committee Member

Dr. Grigory Fedyukovich
Committee Member

ACKNOWLEDGMENTS

I would like to thank my thesis director, Dr. David Whalley, for his continued support and guidance. I would like to thank the committee members, Dr. Linda DeBrunner and Dr. Grigory Fedyukovich, for their willingness to guide me and determine the result of my defense. I would like to thank Dr. Soner Önder and Tino Moore for their original work on the PMTC architecture, and their guidance during my project. Finally, there are various other people who provided assistance with debugging and testing who I would like to thank: Abby Mortensen, Cory Avrutis, Sarah Larkin, Caleb Swain, and Kieran Young.

TABLE OF CONTENTS

Abstract vi
1 Introduction 1
1.1 Research Problems 1
1.1.1 The Fetch Problem 1
1.1.2 The Branch Prediction Problem 2
1.2 Motivation 2
1.2.1 Anticipated Benefits 2
1.2.2 Research Questions 3
2 Literature Review 5
3 Implementation 8
3.1 Technologies Background 8
3.1.1 Asopt 8
3.1.2 FAST and ADL 8
3.2 Overview of Implementation 9
3.3 The ADL File 9
3.4 The Beginsts File 10
3.5 The Config File 11
3.6 Phase One 13
3.6.1 Delay Slot Insertion and Branch Prediction 13
3.6.2 Code Duplication at Jumps and Branches 15
3.6.3 Write Current Function to Beginsts 17
3.7 Phase Two 18
3.7.1 Read Function Information from Beginsts 18
3.7.2 Code Duplication at Function Calls 18
4 Results 20
4.1 SPEC Benchmarks 20
4.2 Measurements 20
4.3 Data and Analysis 23
5 Discussion 27
5.1 Challenges and Limitations 27
5.2 Future Work 28
5.3 Conclusions 28
Bibliography 30

ABSTRACT

The Poor Man's Trace Cache (PMTC) architecture is a novel fetch technique which intends to address two computer architecture problems: the fetch problem and the branch prediction problem. The fetch problem is the inability of many of today's superscalar processors to fetch to their entire issue width due to transfer of control instructions breaking the sequential ordering of instructions. The branch prediction problem refers to incorrectly issued instructions from a branch misprediction which unnecessarily take up processor resources. The PMTC architecture involves static code duplication following direct transfer of control instructions. The number of instructions copied from the target is variably determined based on the distance of the transfer of control instruction from the nearest cache line boundary. For conditional transfer of control instructions, we select to perform PMTC code duplication only on those instructions which are biased towards taken. Static measurements indicate an increase in path separability, which can improve dynamic branch prediction accuracy since approaching the same branch from distinct paths will likely result in distinct prediction histories. Path separability also has the potential to enable additional static code improving transformations.

CHAPTER 1

INTRODUCTION

The Poor Man's Trace Cache (PMTC) architecture is a novel fetch technique which intends to address the problem of a limited fetch bandwidth as well as low branch prediction accuracy. The expected benefits that will come out of the use of the PMTC architecture are an increased fetch bandwidth, increased branch prediction accuracy, and increased path separability, which may in turn enable additional code improving transformations.

1.1 Research Problems

We introduce two computer architecture problems: the fetch problem and the branch prediction problem.

1.1.1 The Fetch Problem

It has become increasingly difficult to reap the benefits of a wide issue width that is present in many of today's superscalar processors. Ideally, a multi-issue superscalar machine would exploit its maximum issue capabilities. If we consider an 8-issue superscalar, then ideally each issue packet would contain eight instructions from the fetch unit. However, the fetch unit is not always able to supply the maximum number of instructions.The fetch unit relies on a starting address from which to start fetching instructions. A fetch unit can only fetch multiple sequential instructions from within the same cache line, or sometimes across a pair of consecutive cache lines. This implies the need for sequential ordering of instructions. A basic block is a sequential set of instructions with a single entry point and a single exit point. A small average number of instructions within a basic block and a large number of transfer-of-control instructions are the main reasons for limitations on the fetch unit. Jump and taken branch instructions disrupt the

sequential flow of instructions, causing any remaining instructions in the fetch buffer to be rendered invalid for decode and execution. This is a problem because it greatly limits fetch efficiency, which in turn limits the performance of the machine. Even if these machines have the ability to issue eight instructions at once, we only gain as much improvement as we can exploit from the fetch buffer.

1.1.2 The Branch Prediction Problem

Another problem that stems from branch heavy code is the problem of branch prediction. Issuing instructions with a branch that is incorrectly predicted would provide no additional benefit to performance. We may even need to incur a penalty to recover from the branch misprediction. Therefore, it is important to have an accurate branch prediction protocol. We want to limit the number of incorrect predictions that need to be handled by the processor.

1.2 Motivation

The PMTC architecture primarily aims to address the fetch problem and the branch prediction problem. It is important to consider these two problems in tandem. Without accurate branch prediction, multi-issue machines would incur great penalties discarding invalid instructions. On the other hand, accurately predicting branches would only provide as much benefit as the number of instructions from the correct branch path that we can issue. In implementing the PMTC architecture as an optimization on MIPS assembly code, we focus on achieving the benefits of wide issue widths while preventing a large penalty for incorrect branch predictions.

1.2.1 Anticipated Benefits

There are three main intended improvements we expect from the implementation of the PMTC architecture:

a) Increased fetch bandwidth

b) Increased branch prediction accuracy

c) Increased path separability

All of these improvements can potentially provide the benefit of increased performance for any application. The first two directly address the fetch problem and the branch prediction problem, respectively. The last improvement we expect as a result of PMTC code duplication. The redundancy of code which is copied is considered path separability, since two instances of the same copied code can be treated as two separate paths. We expect that this increase in path separability will enable additional code improving transformations.

Since the world and its productivity has grown so dependent on computers, any improvement in performance would be beneficial to our society as a whole. Many of today's desktops, laptops, and servers use superscalar processors which would benefit from this fetch technique. This increase in performance can open up the potential for these processors to be able to handle new and more powerful applications.

1.2.2 Research Questions

There were a number of research questions we asked to guide the research and development of this project:

1. Can we implement a solution that addresses the fetch problem while preventing a large number of performance penalties recovering from branch mispredictions?
2. Can we implement a solution that improves branch prediction accuracy?
3. Does addressing our research problems with this implementation provide substantial performance improvements?
4. Does our solution provide greater path separability?
5. Does our solution enable any additional code improving transformations?

These questions will be revisited at the conclusion of this paper, where we can determine from our results if we achieved our expected benefits and goals. We can also assess if our measurement techniques provide an accurate determination of our success, and discuss any limitations of these techniques.

CHAPTER 2

LITERATURE REVIEW

The concept of a dynamic trace cache as a solution to the fetch problem has been studied for decades [3]. This idea involves storing traces of the dynamic instruction stream so that instructions which are non-contiguous can be sequentially fetched. The PMTC architecture can be thought of as a software trace cache, since it statically predicts the instruction stream and "caches" traces of instructions through code duplication. This method, in contrast to previous methods, has the additional benefit of increasing branch prediction accuracy. Previous work [2] has attempted to increase fetch bandwidth by rearranging the order of basic-blocks to follow the natural sequential fetching of the fetch unit. The drawback of this method is that this rearrangement negatively interacts with branch prediction since branches become more biased towards not taken. In contrast, the PMTC architecture does not alter the control-flow of a program, which allows it to consistently improve branch prediction accuracy.

The PMTC architecture is a novel architecture originally developed by Moore and Önder [1] as part of an assembler. This team has developed numerous algorithms which remain similar for our implementation. Figure 2.1 refers to an algorithm developed by Moore and Önder for inserting delay slots.

Algorithm 1: PMTC Assembler Pass 2 Modifications: Delay Slot Insertion

```
   Result: Insert space for a delay slot after select DCTIs
1  DO: pad text segment start to cache line boundary with no-ops;
2  i =text_begin;
3  while i != text_end do
4      if instruction_stack[i] is a direct jump then
5          while i mod cache line size > 0 do
6              insertNo-op(i, instruction_stack);
7              i++;
8      else if (instruction_stack[i] is a backwards branch) &&
         (primeCandidate(i) == true) then
9          setDelayedOpcode(instruction_stack[i]);
10         while i mod cache line size > 0 do
11             insertNo-op(i, instruction_stack);
12             i++;
13     else
14         i++;
```

Figure 2.1: Delay Slot Insertion Algorithm

This project explores an implementation of the PMTC architecture in an assembly optimizer, rather than part of an assembler. Due to greater knowledge of loop information at the optimization level, we are able to more accurately identify which paths associated with a transfer of control instructions will likely benefit from code duplication. In general, direct transfers of control which could leave a loop are biased towards the successor which stays in the loop. We assume that branches which are biased towards taken will more likely benefit from code duplication. Moore also made this assumption, but did not have the same access to loop information as we do at the assembly optimizer level. Moore's implementation assumes that a backwards branch is most likely to stay within a loop, which can be seen in line 8 of Figure 2.1. Because the assembly optimizer can perform control flow analysis, we know exactly which basic blocks are in any given loop. We expect that the greater knowledge of loop information will provide greater performance improvement. Additionally, the implementation of PMTC at the

assembly optimizer level opens up the opportunity to apply additional code-improving transformations on the duplicated code which can further improve performance.

CHAPTER 3

IMPLEMENTATION

3.1 Technologies Background

3.1.1 Asopt

The terms “asopt” or “opt” are used to refer to the assembly optimizer used in this project. This application performs various code-improving transformations on assembly files. Previously implemented transformations include dead assignment elimination, copy propagation, and loop unrolling. Asopt can keep track of transformation statistics. Command line options are used to signal which optimizations should be applied. The PMTC implementation was split into two phases, which requires two distinct flags. The flag -d is used for phase one. The flag -p is used for phase two.

3.1.2 FAST and ADL

The Flexible Architecture Simulation Tool (FAST) is the simulation system used in this project. This system utilizes an Architecture Description Language (ADL) developed specifically for FAST. The ADL acts as a specification for a given microarchitecture. The output of FAST includes a simulator, assembler, and disassembler for the microarchitecture described by the given ADL file. A researcher studying a microarchitecture may write their own simulator by hand, but can instead write an ADL file for their architecture and pass it as input into FAST. The developers of FAST estimate that it takes a developer about one to three months to describe an architecture in ADL, for which would take 18 to 24 months to write a simulator in C [4]. It is also relatively simple to update an ADL file. In addition to the productivity gains, the resulting simulator can also gather performance data and other statistics.

3.2 Overview of Implementation

PMTC code duplication occurs in two phases. Both phase one and phase two refer to processing transformations which are performed on each function in a given file. If a phase is being performed on an entire file, then this means that the phase is being performed on each function in that file. Likewise, a phase being performed on an entire program implies that the phase is being performed on each file which comprises the program.

Phase one handles the insertion of all delay slots for branches, jumps, and function calls. PMTC code duplication for jump and branch instructions occurs in the first phase, while PMTC code duplication for call instructions must be handled in the second phase. The reason for this is that in many cases the caller function and the callee function are defined in separate files. This requires a first pass through every function, in every file, in every directory of the program. We compile a list of functions in a file called the “beginsts” file. To assist phase two, we generate the beginsts files when performing phase one on the entire program. Phase two reads from all the beginsts files generated and uses this information to perform PMTC code duplication at function calls.

3.3 The ADL File

The ADL file used for the PMTC architecture is a modified version of the ADL file used for the MIPS architecture. The PMTC file includes three additional types of instructions. First, we needed a delayed version for jumps, branches, and function calls. A delayed transfer of control signals that instructions sequentially after the transfer of control should be fetched and executed before the transfer of control takes place, given that the transfer will be taken. The delayed version includes an additional ‘d’ at the end of the mnemonic. For example, the delayed version for “beq” would be “beqd.” Second, we needed to recognize branch and jump instructions which

utilize an address offset to locate the target address. For this, we add a double underscore to the end of the mnemonic. For example, “beq” would expect an absolute address or label for its argument, while “beq__” would expect an address offset relative to its location. Finally, we needed a third type of instruction which is simply a combination of the two modifications. This type of instruction would end with “d__” and would both be delayed and expect an address offset. Referring back to our example, the modified version of “beq” would be “beqd__”. The ADL file for the PMTC architecture includes descriptions for the three additional types of instructions.

3.4 The Beginsts File

Code duplication for branches and jumps can be locally executed within one assembly file. On the other hand, code duplication for function calls often cannot be locally executed. There are many cases where functions are declared and defined in files separate from where they are called, and in some cases in separate directories. The beginsts file serves as a repository of all the functions across all files in a given directory. Note that it would be excessive to store all of the instructions of a function. We only store the first N instructions of a function, where N = instructions per cache line. We generate the beginsts file in the first iteration of PMTC, which we refer to as phase one. We use the beginsts file in the second iteration of PMTC in order to perform code duplication for function calls. This is the main purpose of splitting the PMTC implementation into two phases.

The first line of the beginsts file contains the total number of functions currently being stored in the file. Each function lists the file name, function name, and at most the first N lines of the function, where N = instructions per cache line. Some functions may have less than N instructions. For this reason, the keyword “end” is used to signal the end of a function listing.

Two consecutive function listings from the SPEC benchmark 462.libquantum are shown in Figure 3.1. For more information on SPEC benchmarks, refer to Section 4.1 of this paper.

```
...
classic.s quantum_inverse_mod
        div     $0,$5,$4
        mfhi    $2
        addiu   $3,$0,1
        beq__   $2,$3,13
        sll     $3,$5,1
        addiu   $2,$0,1
        addiu   $7,$0,1
        addiu   $2,$2,1
        end
measure.s quantum_real
        mtc1    $4,$f0
        jr      $31
        end
...
```

Figure 3.1: Beginsts File Function Listing Example

3.5 The Config File

The configuration file contains user-specified values and options for PMTC, and a list of directories that comprise a program. Opt assumes a particular name, location, and structure of the configuration file. The name of the configuration file must be pmtc.config and it must be located in the directory from which opt is being executed. The config file used for the SPEC benchmark 999.specrand is shown in Figure 3.2. For more information on SPEC benchmarks, refer to Section 4.1 of this paper.

```
cache line size: 8
function info file name: beginsts.txt
treat function not found as error (y/n): n
program directories:
~/SPEC/libs/adl-support.a
~/SPEC/libs/adl-support-fortran.a
~/SPEC/libs/diet-libgcc
~/SPEC/libs/libc.a
~/SPEC/libs/libgfortran.a
~/SPEC/libs/libm.a
~/SPEC/libs/lib-pthread.a
~/SPEC/06/integer/999.specrand
```

Figure 3.2: PMTC Configuration File

The cache line size specifies how many instructions fit between two cache line boundaries. The function info file name is the name of the beginsts file. For this project, “beginsts.txt” was always used as the name of this file. However, the implementation allows for any user-specified name. At a function call, the program attempts PMTC code duplication by locating the function information in a beginsts file. If this function cannot be found in the beginsts file, then opt can do one of two things: report an error or ignore the function call. The next line of the config file specifies an option which either treats “function not found” as an error when the value is ‘y’, or ignores this entirely if the value is ‘n’. It may be necessary to turn off this option when some of the program directories contain an incomplete beginsts file. This would occur if not all program files have been processed with the -d flag, which may be needed to isolate asopt errors. Finally, the config file lists the directories which comprise the entire program. This is essential for opt to locate all the beginsts files during phase two.

3.6 Phase One

PMTC phase one can be broken up into three distinct parts which occur once per function:

1. Delay Slot Insertion and Branch Prediction – "nop" instructions are inserted up to the cache line boundary after function calls, jump instructions, and branches which are biased towards taken, which is also determined in this step.
2. Code Duplication at Jumps and Branches – delay slots are filled with copied instructions from the target address for jumps and branches.
3. Write Current Function to Beginsts – at most a cache line's worth of instructions at the beginning of the function is written to the beginsts file.

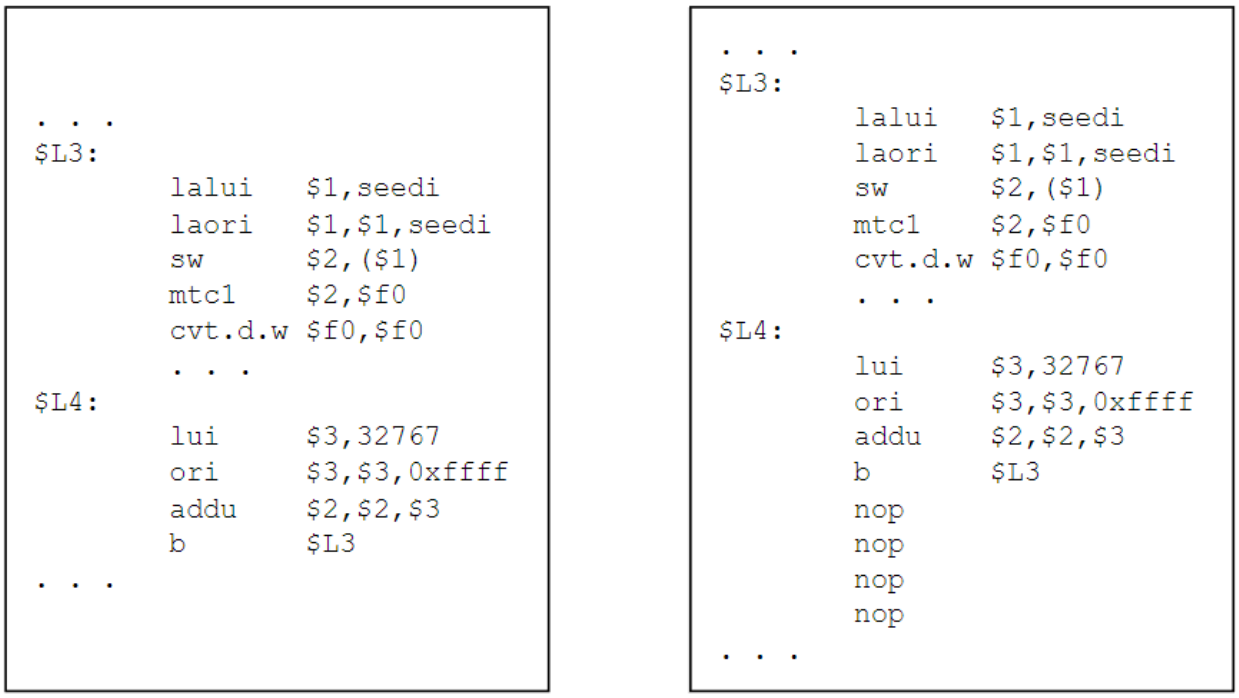
3.6.1 Delay Slot Insertion and Branch Prediction

The algorithm for inserting delay slots remains very similar to the one developed by Moore and Önder, which is shown in Figure 2.1. For any jump, call, or biased branch, we insert "nop" instructions until the end of the cache line. The main difference is the logic used to determine which branches should be delayed. Figure 2.1 includes a check for backwards branches. There is no such check in this implementation. The assembly optimizer keeps track of all control flow loops, giving us a better insight to which branches are biased towards taken. Loops typically branch back on themselves at least once, but only leave the loop exactly once. Thus, we can assume that branches which could leave a loop are biased towards the successor which stays in the loop. This check could not have been implemented in the assembler developed by Moore and Önder, since it is unknown which branches stay within a loop at the assembler level. Figure 3.3 shows how the program determines whether or not a branch is biased. A return value of "TRUE" signifies that the branch is biased towards taken, and thus will be transformed to a delayed branch. Otherwise, the function returns "FALSE".

```
if(current block has no fall-through){
    return TRUE;
}
else{

  //if branch leaves any loop, biased not taken
  for(all loops in function){
    if(loop contains current block){
      if(loop does not contain target){
          return FALSE;
      }
    }
  }

  //if branch stays in any loop, biased taken
  for(all loops in function){
    if(loop contains current block){
      if(loop contains target){
          return TRUE;
      }
    }
  }

  //otherwise, not biased one way or another
  return FALSE; //choose to predict not taken
```

Figure 3.3: Function for Determining Biased Branches

Figure 3.4 shows the effects of delay slot insertion on the source code. Opt determined that there is room for four delay slots following the jump instruction in the block starting at $L4. It is necessary to apply delay slot insertion on the entire function before performing code duplication, since copied instructions may also include their own delay slots.



a. original code

b. after delay slot insertion

Figure 3.4: Delay Slot Insertion Example

3.6.2 Code Duplication at Jumps and Branches

After all delay slots have been inserted with nops across the entire function, we begin PMTC code duplication at jumps and biased branches. For any jump or biased branch, we identify the target of the control transfer. Once located, we copy instructions from the target until we either run out of delay slots (nop instructions) or run out of instructions to copy. If we run out of instructions to copy before we run out of delay slots, then the remaining "nop" instructions are changed to "trash" instructions. The "trash" instruction has the same behavior as the "nop" instruction, but is used to represent a filled delay slot instead of an unfilled delay slot. The branch or jump instruction must be replaced with a delayed version of the instruction. This is done by appending a 'd' to the end of the mnemonic. A delayed instruction signals to the

simulator that instructions sequentially after the transfer of control, up to the cache line boundary, should be fetched and executed. After the cache line boundary is reached, the simulator starts fetching and executing from the target, skipping past the instructions which were already fetched. For details on the implementation of this instruction type, refer to Section 3.3 of this paper.

Consider our example which was first presented in Figure 3.4. Opt knows to copy exactly four instructions from the target, $L3, because that is how many delay slots are listed after the jump instruction. Figure 3.5 shows the result of this transformation.

```
. . .
$L3:
        lalui   $1,seedi
        laori   $1,$1,seedi
        sw      $2,($1)
        mtc1    $2,$f0
        cvt.d.w $f0,$f0
        . . .
$L4:
        lui     $3,32767
        ori     $3,$3,0xffff
        addu    $2,$2,$3
        b       $L3
        nop
        nop
        nop
        nop
. . .
```

a. before PMTC code duplication

```
. . .
$L3:
        lalui   $1,seedi
        laori   $1,$1,seedi
        sw      $2,($1)
        mtc1    $2,$f0
        cvt.d.w $f0,$f0
        . . .
$L4:
        lui     $3,32767
        ori     $3,$3,0xffff
        addu    $2,$2,$3
        jd      $L3
        lalui   $1,seedi
        laori   $1,$1,seedi
        sw      $2,($1)
        mtc1    $2,$f0
. . .
```

b. after PMTC code duplication

Figure 3.5: PMTC Code Duplication Example

3.6.3 Write Current Function to Beginsts

At this point, all delay slots for branch and jump instructions have been filled, and call instructions still have unfilled delay slots. This last step involves adding a function listing for the current function to the beginsts file. We add the filename, function name, and a list of the first N lines of the function, where N = instructions per cache line. For more information on the format of this function listing, refer to Section 3.4 of this paper.

There is one type of transformation on the code source which can still happen at this step. If we encounter a branch or jump instruction, we must replace the label for the target with the address offset of the target from the control flow instruction in question. We must also append a double underscore to the end of the mnemonic. For details on the implementation of this instruction type, refer to Section 3.3 of this paper. This transformation must be done before the function is written to the beginsts file. In the case that we must copy a branch or jump instruction into the delay slot for a function call, the control flow instruction would be referencing a label which is defined in the callee function, not the caller. If the target were to live in a separate file, then the assembler would either not understand how to translate the label into an offset, or would translate the label into the incorrect offset. This transformation is essentially performing this translation so that there is no ambiguity at the assembler level.

Figure 3.6 shows the effect of this transformation. We replace the mnemonic “bne” with “bne__” so that the assembler knows to expect an address offset. We replace the target of the branch instruction, $L3, with an offset of -3 because the first instruction of $L3 is “mult $2,$4”, which is three instructions before the branch instruction.

a. before replacing target with offset

b. after replacing target with offset

Figure 3.6: Replacing Branch Target with Address Offset

3.7 Phase Two

PMTC phase two can be broken up into two distinct parts which:

1. Read Function Information from Beginsts – all functions are read from all the beginsts files and stored in a data structure. This is done before asopt processes the assembly file.
2. Code Duplication at Function Calls – delay slots at function calls are filled with the function's instruction listing which was read from the beginsts file.

3.7.1 Read Function Information from Beginsts

There exists one beginsts file per directory of the program, with its own function count in the first line of the file. Opt reads the list of directories from the configuration file and stores this list in a data structure. Opt then uses this list to add up all the function counts for each file, and allocates a dynamic array of structs with this size. We then iterate through the list of directories again to read all the function listings into this array.

3.7.2 Code Duplication at Function Calls

This step is similar to the step described in Section 3.6.2 of this paper. For any function call, we search for the function in the array of functions generated in the last step. Once located, we copy

instructions from the callee function into the delay slots after the function call. Similarly to code duplication at branches and jumps, we do this until we either run out of delays slots or run out of instructions to copy. If there are extra delay slots remaining, they are replaced with “trash” instructions.

CHAPTER 4

RESULTS

4.1 SPEC Benchmarks

The Standard Performance Evaluation Corporation (SPEC) benchmarks are standardized test programs used to evaluate machine performance. These benchmarks are used widely in computing research because they are written to accurately mimic real-world applications. Some examples include a video compression program, a path finding algorithm used in games, and quantum computing simulations [5]. There exist both integer and floating point benchmarks. Any given benchmark contains various source files. The name of a SPEC benchmark is of the form <benchmark_id_number>.<title>. Some examples include 999.specrand, 462.libquantum, and 026.compress. The SPEC CPU2006 benchmarks are the programs which will be used for our evaluations.

4.2 Measurements

Opt keeps track of various static measurements while processing a file. Upon completion, opt dumps the statistics to a file with a .pmtc extension. Running opt on the file “decoherence.s” from the benchmark 462.libquantum results in a statistics file called “decoherence.pmtc”, which is shown in Figure 4.1 as an example.

```
delay slots inserted: 70
delay slots filled: 13
jumps with delay slots: 4
branches with delay slots: 2
calls with delay slots: 11
paths duplicated: 3
code size before (insts): 208
code size after (insts): 278
```

Figure 4.1: decoherence.pmtc

The “delay slots inserted” value is the total number of delay slots inserted in the decoherence.s file. The “delay slots filled” value is the total number of those delay slots which were replaced with a duplicated instruction. The next three values are the total number of instructions of a given type which were given delay slots. Whenever a jump or branch instruction was copied into a delay slot, the “paths duplicated” value was incremented. This value is used to measure the effect that the PMTC transformations have on path separability. The last two values show the total code size, in number of instructions, before and after the PMTC optimization has been performed. These values are used to measure the code size growth.

A separate program was developed to gather data from all the .pmtc files for a given benchmark, and produce statistics for the entire program. The resulting statistics file for the benchmark 462.libquantum is shown in Figure 4.2.

```
delay slots inserted: 3765
delay slots filled: 1418
jumps with delay slots: 248
branches with delay slots: 173
calls with delay slots: 498
paths duplicated: 142
code size before (insts): 7943
code size after (insts): 11708

STATISTICS:
percent of delay slots filled: 37.662682
total delayed instructions: 919
   percent jumps: 26.985855
   percent branches: 18.824810
   percent calls: 54.189335
code size growth (insts): 3765
percent growth of code size: 47.400227
percent of delayed instructions with a duplicated path: 15.451578
```

Figure 4.2: Benchmark Statistics Example

The first section is the result of adding up all the values from all the .pmtc files in the benchmark. The next section consists of calculations made from the data in the first section. Assume that every value that says “percent” was multiplied by 100 to represent the percent instead of the fraction. The value for “percent of delay slots filled” is obtained by taking delay slots filled over the total number of delay slots inserted. The values for percent jumps, branches, and calls are obtained by taking the individual count over the total number of delayed instructions. The percent growth is obtained by taking the total growth over the original size of the code. The final value is obtained by taking the paths duplicated over the total number of delayed instructions.

4.3 Data and Analysis

All data was taken from the same thirteen benchmarks. These benchmarks consist of a mix of both integer and floating point programs from the SPEC CPU2006 benchmarks.

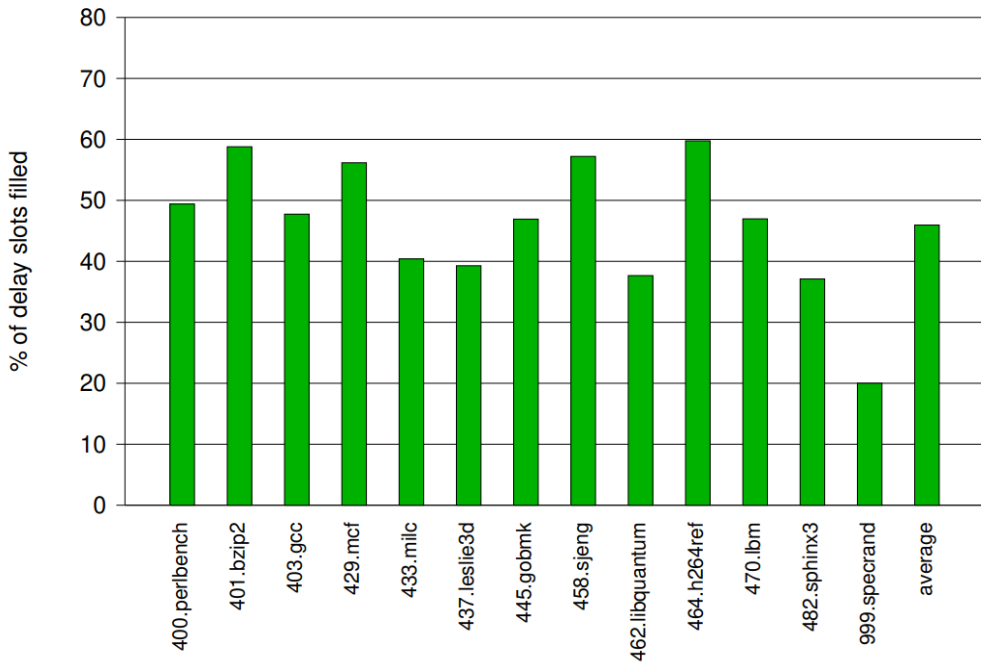


Figure 4.3: Percent of Delay Slots Filled after Phase One

Figure 4.3 shows the percent of delay slots filled for each of the benchmarks. Note that we were only able to gather statistics on this for phase one, so there are still delay slots for call instructions which have not yet been filled. Most of the benchmarks lie around 40% to 60%, averaging around 45%. This shows a significant amount of delay slots being filled in phase one. The data suggests that we may find similar results for the delay slots being filled in phase two. In other words, we expect the results for phase two to also lie around 40% to 60% to offset the results from phase one. Note that the two values would likely not add up to 100%, since there are some cases where not all delay slots can be filled. In these cases, they are replaced with “trash” instructions.

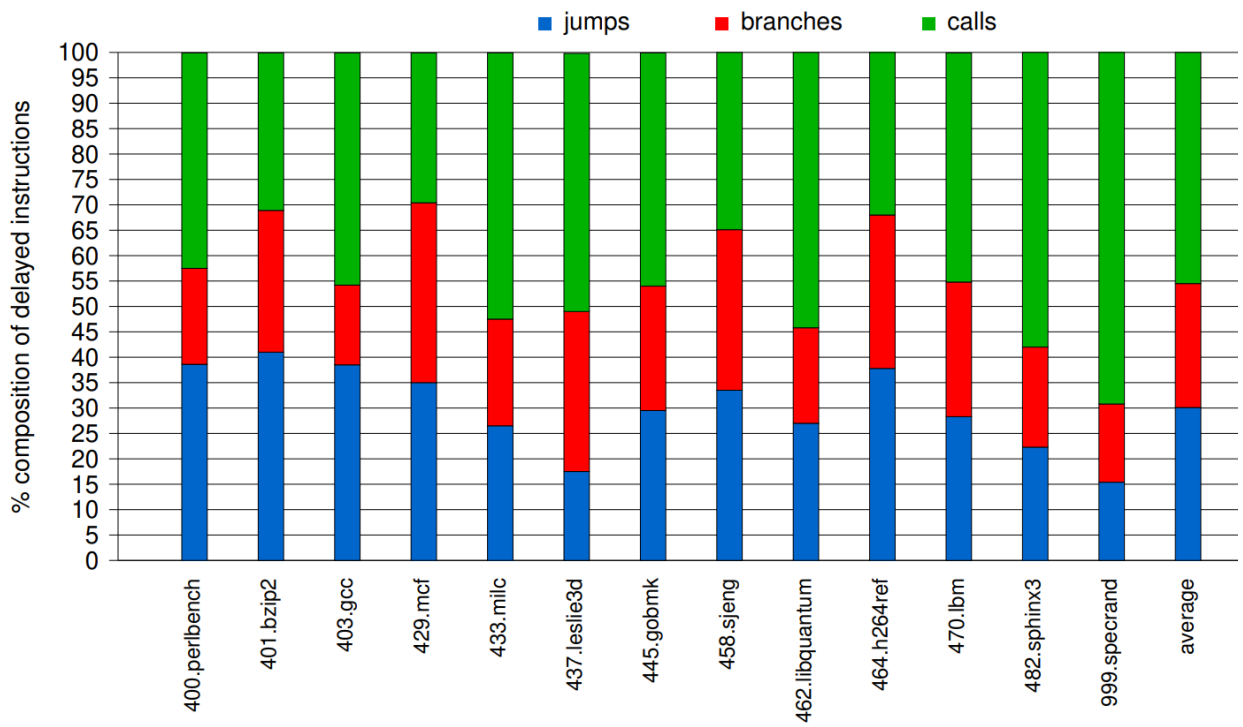


Figure 4.4: Composition of Instructions Given Delay Slots

Figure 4.4 shows the composition of instructions given delay slots by instruction type. Note that the percentage of jumps and branches taken together roughly matches the graph in Figure 4.3. These are the results we expect if a majority of the delay slots for jump and branch instructions are being filled. This also supports the suggestion that the remaining delay slots will be mostly filled during phase two.

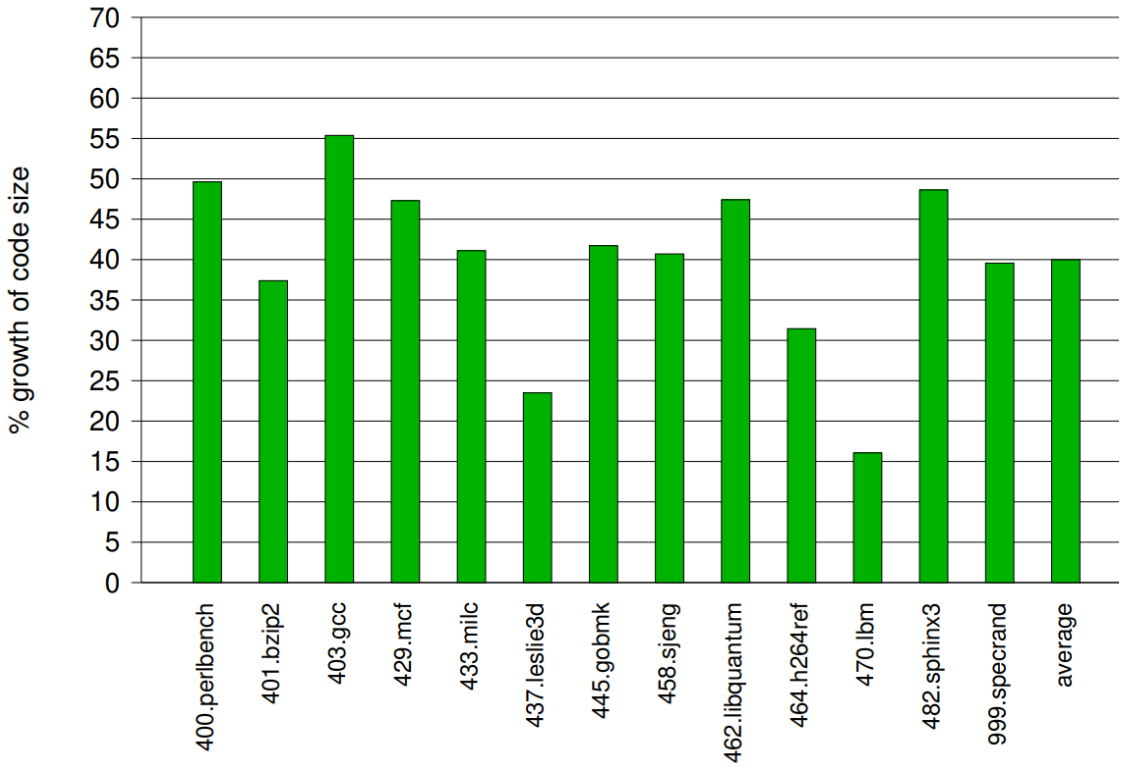


Figure 4.5: Percent Growth of Code Size

Figure 4.5 shows the percent growth of the code size for each of the benchmarks. Most of the benchmarks lie around 40% to 50%, which is a significant increase in code size. The average lies at 40% due to a few benchmarks which saw little code growth. The large increase in code size is considered one of the drawbacks of this implementation. This data will help guide our future research to focus on methods to decrease unnecessary growth. There does not seem to be any correlation between percent growth and the size of the original program. In other words, nothing in this data suggests that percent growth increases or decreases with the size of the program.

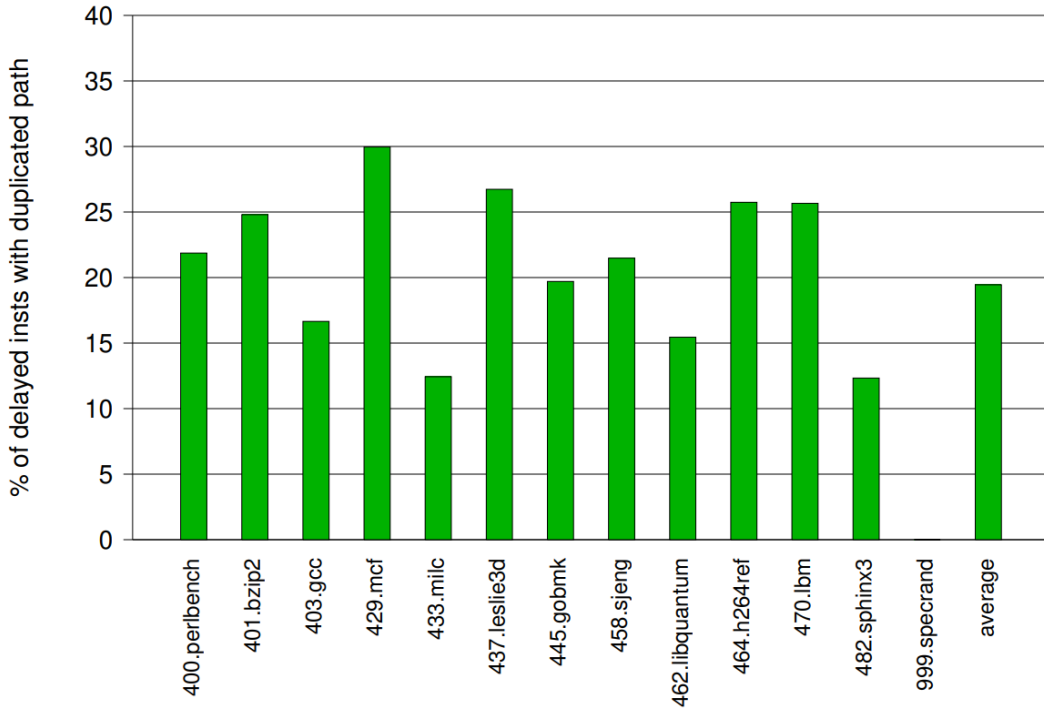


Figure 4.6: Percent of Delayed Instructions with a Duplicated Path

Figure 4.6 shows an estimate of the percent of delayed instructions with a duplicated path. This tells us about what percent of our transformations result in an increase in path separability. Most of the benchmarks lie around 20% to 30%. Note that we were only able to gather statistics on this for phase one, so there are still delay slots for call instructions which have not yet been filled. We expect to see a similar increase in phase two, proportional to the percent of delay instructions which are call instructions. After both phase one and phase two, it is likely that about 50% of our transformations result in an increase in path separability. This is a significant increase, and suggests that the resulting code would benefit from additional optimizations. This data will help guide the future work to be more involved with applying additional optimization and measuring how the effects on path separability enable certain transformations.

CHAPTER 5

DISCUSSION

5.1 Challenges and Limitations

While working on the project, we discovered an issue with copying jump and branch instructions into the delay slots for a function call. The jump or branch instruction being copied would have a target label which is defined in the callee function, not the caller. This would result in unexpected behavior, as the assembler would attempt to locate the label only within the same file. In response to this issue, we brainstormed various possible solutions. One possibility was to relabel every label so they would all be unique and global, thus removing any ambiguity. Another possibility was to replace the target label with the target address offset from the branch. The latter was implemented. For more information on this transformation, refer to Section 3.6.3.

In order for the PMTC architecture to correctly work, the instructions must be aligned on the cache line boundary. At the beginning of every function, we add an align directive which agrees with the cache line size assumed by opt. For example, we use the ".align 5" directive so the simulator knows to align instructions so that there are eight instructions, or 2^5 bytes, per cache line. During testing, we discovered that the PMTC simulator produced by FAST does not correctly align the instructions. We notified one of the developers of FAST and ADL, and he discovered that the linker is changing the alignment. He told us that this will be fixed in the next version release. Without the ability to simulate the code produced, we were unable to get any of the dynamic measurements that we had planned. Due to this issue, our testing and measurements were limited to be only static at this time.

In order to test phase two, it must be re-applied to the resulting program from phase one. There was no infrastructure in place to be able to re-apply opt to the resulting code, since opt relies on an existing setup of the benchmark directories. This setup would need to be reconstructed during phase one in order for opt to test the modified program. Addressing this obstacle was not sufficiently considered in the scope of this project, thus testing phase two will be postponed to future work. Due to this unforeseen challenge, our testing and measurements were limited to phase one results at this time.

5.2 Future Work

We will be able to begin gathering static measurements for phase two after building the necessary framework. Once the new version of FAST is released, we will be able to begin testing the resulting code and obtaining dynamic measurements. We may discover that these measurements indicate insignificant speedup, or inefficient use of delay slots, or inaccurate branch prediction strategies. Based on this, we can modify our implementation and experiment with different delay slot insertion algorithms and branch prediction algorithms. We may also want to explore the potential for additional assembly optimizations that PMTC code duplication may enable.

5.3 Conclusions

While the results may have been more limited than we originally intended, they proved to be a promising sign that the PMTC architecture can sufficiently address the fetch problem and the branch prediction problem, both introduced in Section 1.1. The results encourage further development of the PMTC architecture as an assembly optimization.

Let us revisit our research questions:

1. Can we implement a solution that addresses the fetch problem while preventing a large number of performance penalties recovering from branch mispredictions?
2. Can we implement a solution that improves branch prediction accuracy?
3. Does addressing our research problems with this implementation provide substantial performance improvements?
4. Does our solution provide greater path separability?
5. Does our solution enable any additional code improving transformations?

Questions 1 through 3 cannot be sufficiently answered without being able to simulate the architecture, thus will have to be revisited in future work. In response to question 4, our results do suggest that the solution provides greater path separability. This implies that the solution has the potential to enable additional code improving transformations. While this is promising, question 5 cannot be sufficiently answered with our current knowledge. The results are an encouraging sign to continue building upon the PMTC implementation in our future work.

BIBLIOGRAPHY

[1] Moore, Tino C. “Poor Man's Trace Cache: A Variable Delay Slot Architecture.” *Michigan Technological University*, 2022.

[2] Ramírez, Alex, et al. “Software Trace Cache.” *Proceedings of the 13th International Conference on Supercomputing*, 1999. *ICS '99*, https://doi.org/10.1145/305138.305178.

[3] E. Rotenberg, S. Bennett and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Paris, France, 1996, pp. 24-34, doi: 10.1109/MICRO.1996.566447.

[4] Soner Onder and Rajiv Gupta. “Automatic generation of microarchitecture simulators”. In: ¨ IEEE International Conference on Computer Languages. Chicago, May 1998, pp. 80–89.

[5] SPEC CPU Subcommittee and by the original program authors. *SPEC CPU2006 Benchmark Descriptions*.