Florida State University Libraries

Electronic Theses, Treatises and Dissertations The Graduate School

2018

Deep: Dependency Elimination Using Early Predictions

Luis G. Penagos

Follow this and additional works at the DigiNole: FSU's Digital Repository. For more information, please contact lib-ir@fsu.edu

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DEEP:

DEPENDENCY ELIMINATION USING EARLY PREDICTIONS

By

LUIS G. PENAGOS

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

2018

Copyright © 2018 Luis G. Penagos. All Rights Reserved.

Luis G. Penagos defended this thesis on July 20, 2018.
The members of the supervisory committee were:

David Whalley

Professor Directing Thesis

Xin Yuan

Committee Member

Weikuan Yu

Committee Member

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

ACKNOWLEDGMENTS

I would like to thank Dr. Whalley for his help, guidance and time. I would not have been able to complete this project without his open-door policy, constant input and guidance throughout the project, even on holidays and weekends. I would also like to extend my gratitude to Ryan Baird, Zhaoxiang Jin and Gorkem Asilioglu at Florida State University and Michigan Technological University for their constant aid and insight on the VPO compiler and FAST superscalar simulator respectively. Lastly, I would like to thank my family for their continued support throughout the project. I could not have accomplished this endeavor without them. This work was supported in part by the US National Science Foundation (NSF) under grants DUE-1259462, DUE-1241525, DGE-1565215, DRL-1640039, IIA-1358147, and CCF-1533846.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Conditional branches have traditionally been a performance bottleneck for most processors. The high frequency of branches in code coupled with expensive pipeline flushes on mispredictions make branches expensive instructions worth optimizing. Conditional branches have historically inhibited compilers from applying optimizations across basic block boundaries due to the forks in control flow that they introduce. This thesis describes a systematic way of generating paths (traces) of branch-free code at compile time by decomposing branching and verification operations to eliminate the dependence of a branch on its preceding compare instruction. This explicit decomposition allows us to move comparison instructions past branches and to merge pre and post branch code. These paths generated at compile time can potentially provide additional opportunities for conventional optimizations such as common subexpression elimination, dead assignment elimination and instruction selection. Moreover, this thesis describes a way of coalescing multiple branch instructions within innermost loops to produce longer basic blocks to provide additional optimization opportunities.

CHAPTER 1

INTRODUCTION

Conditional branches have traditionally been a performance bottleneck for most processors. Despite advances in branch prediction techniques, conditional branches still constitute a large portion of programs, comprising a disproportionate number of execution cycles. Branches introduce discontinuities in program control flows, introducing control dependences at the instruction level, a problem which disrupts sequential instruction fetching by the instruction fetch unit and limits the effectiveness of hardware-based trace caches [3]. Additionally, conditional branches have the potential to trigger expensive pipeline flushes on mispredictions, a problem which is further exacerbated by increasingly deeply pipelined superscalar processors [1].

Extensive work has been conducted to reduce the number of control hazards introduced by conditional branches. Techniques such as reordering branches using profile data and estimated branch execution costs and branch condition merging are among such techniques [1, 2]. Conditional branch reordering however oftentimes introduces additional branches into the control flow of the most beneficial ordering in addition to relying on profile data. Similarly, branch condition merging also relies on profile data to optimize branches along the most frequently executed paths.

We propose to introduce explicit predict and branch instructions to replace traditional branches. There has been prior work on decomposing branches into prediction and resolution instructions. However, this thesis proposes an implicit branch misprediction recovery system, eliminating the need for explicit recovery code and limiting the adverse effects of additional static code size increases. Our technique minimizes the code size increase and does not require profile data, unlike previous efforts. Further, we propose coalescing multiple explicit predict and branch instructions to yield larger superblocks which may enable other compiler optimizations such as common subexpression elimination, dead assignment elimination and instruction selection. Lastly, we propose a way to collapse consecutive explicit predict and branch instructions into 1 instruction within innermost loops to reduce the number of instructions executed by the processor. This novel method of creating and optimizing traces at compile time has been largely unexplored.

CHAPTER 2

DEPENDENCY ELIMINATION USING EARLY PREDICTIONS (DEEP)

The benefits made possible by the DEEP optimization are best understood by first examining how traditional branches are executed on today's out-of-order (OoO) superscalar processors.

2.1 Background

In most processors, traditional branch instructions access the branch predictor (BP) in the instruction fetch (IF) stage to begin speculatively fetching instructions at the branch target. Additionally, all instructions simultaneously access the branch target buffer (BTB) during the IF stage to potentially obtain a branch target in the case that the processor finds this to be a branch instruction during the instruction decode (ID) stage and the BP predicted the branch to be taken. If an instruction hits in the BTB and there is an associated branch target set, the BP can either predict taken or not taken, depending on the global branch history register, and set the PC to the address where the next instruction should be fetched. In the case of a BTB miss, a branch instruction will be predicted as not taken and must wait until the ID stage to calculate its taken branch target as both the instruction type and jump offset are not known to the processor during the IF stage. The actual branch comparison will not occur until the instruction executes.

There are many cases where the branch's corresponding comparison instruction can be delayed due to unavailable register operands or lack of reservation stations in hardware. It is not until the retire (RFU) stage that the processor verifies the branch's prediction with the result calculated by its corresponding comparison instruction when it attempts to commit the branch instruction. If the branch was mispredicted, the prediction is corrected and is then used to update the global branch history register. The branch then signals an exception to flush the pipeline of all instructions not yet committed. Then the processor begins to fetch instructions at the branch's alternate target. Out-of-order processors wait until the retire stage to handle rollbacks caused by branch mispredictions to support precise, in order exceptions. In the case of a correct prediction, the global branch history

register is updated, the BTB is updated with the branch's PC and jump offset and the branch is allowed to commit. Hence, processors are already implicitly decomposing branch instructions into (1) a branch operation and (2) a verification operation over multiple cycles.

2.2 Explicitly Decoupling Branches

In light of what current processors implicitly do with traditional branch instructions, we propose that the compiler explicitly decompose a branch into 2 instructions: a predict and branch (`pb`) instruction and a test and verify (`tv`) instruction. Unlike a traditional branch instruction, the `pb` instruction does not contain the branching condition. A `pb` instruction only contains the label to which it should jump to when it is predicted as taken. In contrast, the `tv` instruction will contain the corresponding branching condition, but no label. The `tv` instruction will verify the branch prediction made by its corresponding `pb` instruction. Hence, the processor no longer needs to implicitly decompose the branch instruction at runtime. Note that by reversing the order of the comparison and branch operations, we eliminate the branch's dependence on a preceding `cmp` instruction.

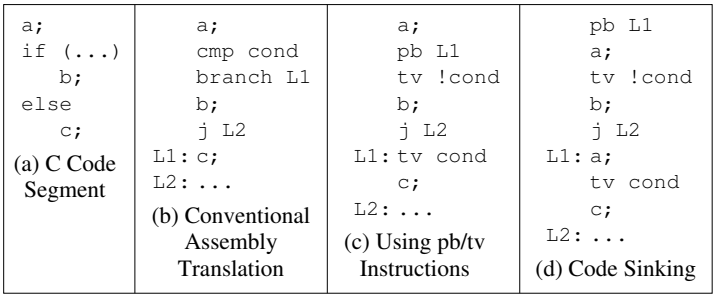

Figure 2.1: Explicitly Decoupling Branches into Prediction and Verification Instructions

Let us call a sequence of straight-line code a DEEP block. Each branch can then be said to have a pair of associated DEEP blocks, one for the branch not taken case and one for the branch taken case. We propose to sink a `pb` instruction's corresponding `tv` instruction to both DEEP blocks to verify the branch as shown in Figure 2.1(c). By pushing down the branch's verification instruction into both paths, we need to issue two different verification instructions: one to verify the original branch condition along the taken path and an inverted comparison for the branch fall thru (not taken) case. Note that because mutually exclusive test and verify conditions are placed in

both immediate successors of a branch, we must convert IF only branches to IF-ELSE branches by introducing an additional unconditional jump to avoid executing two test and verify instructions for the same branch as shown in Figure 2.2(c). We believe that the potential to apply further compiler optimizations within these DEEP blocks can outweigh the negligible code size increase and low cost of the additional unconditional jump.

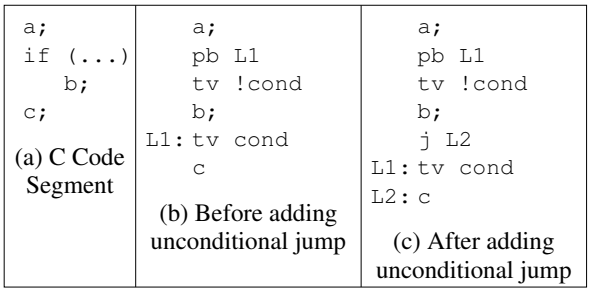

Figure 2.2: Promoting IF-only Constructs to IF-ELSE Constructs

2.3 Code Sinking

As a direct result, the new branch instruction, `pb`, no longer relies on a preceding comparison instruction as in Figure 2.1(b). We can then sink the code preceding the `pb` instruction, namely `a;`, into both successor blocks as shown in Figure 2.1(d) to create longer paths of straight-line code. Note that the semantic meaning of the code in 2.1(d) is equivalent to that shown in Figure 2.1(b) which would have been otherwise produced using traditional branches. However, there are no longer any control dependences between the code executed before the branch (`a;`) and the code executed after the branch (`b;` or `c;`). Instead, both of the branch's immediate successors now consist of longer straight-line paths of code which can potentially allow other compiler optimizations such as common subexpression elimination, dead assignment elimination and instruction selection to be applied. These branch free paths can achieve performance benefits to similar to those achieved by hardware trace caches. However, unlike trace caches, the DEEP technique makes these paths explicitly known to the compiler and does not restrict their length to a pre-set value. As a result, the compiler knows the exact beginning and end locations of these paths so they can be efficiently fetched using a sequential hardware prefetcher.

2.4 Delayed Branch Verification

Traditionally, a branch instruction is not allowed to commit until its prediction has been verified by its corresponding preceding `cmp` instruction at the retire stage. Note however that in the generated DEEP code in Figure 2.1(c) and in Figure 2.1(d), the verification instruction no longer precedes the branch instruction. Instead, it is placed into both of the branch's immediate successors. In order to preserve program semantics, we cannot commit a `pb` instruction until its corresponding `tv` instruction has executed and verified the branch prediction, regardless of whether or not the prediction was correct or incorrect. This forward data dependence on the `tv` instruction is handled with dedicated hardware prediction and verification queues detailed in the following chapter. We handle this explicit branch prediction verification at the retire phase, similar to traditional branch instructions, to support precise, in-order exceptions on OoO processors. All interleaving instructions between a `pb` instruction and its corresponding `tv` instruction will be executed speculatively by the processor and hence have the possibility of being rolled back on erroneous predictions issued by `pb` instructions. This important implication suggests that we should limit the length of DEEP blocks and that we should not push expensive operations such as `div` instructions between a `pb` instruction and its corresponding `tv` instruction. It is equally important that no exceptions be raised while instructions are speculatively executing (such as a divide by 0 runtime exception).

When the processor enters this speculative state, the compiler must not sink any function calls between a `pb` instruction and its corresponding `tv` instruction. Since both `pb` and `tv` instructions update special hardware queues, a function call may also update these hardware structures and invalidate assumptions about these hardware structures made by the function caller. The compiler must also not sink any partial word stores into this speculative execution state if a processor does not fully support partial load/store forwarding as this can lead to a potential deadlock situation where a test and verify instruction depends on a stalled memory access after a partial word store.

2.5 Instruction Hoisting

In an attempt to mitigate increased misprediction recovery time and limit speculative execution, we can hoist identical instructions up from both immediate successor basic blocks of a `pb` instruction into non-speculative execution. In the DEEP optimization, we attempt to make longer paths of straight-line code by sinking pre-branch code into both immediate successor basic blocks

of a conditional branch to potentially be able to apply other optimizations. If we are unable to successfully apply other code improving optimizations on these straight-line code paths such as common subexpression elimination, dead assignment elimination or instruction selection, we can effectively undo the DEEP transformation. This will yield three benefits: (1) it will potentially enable additional compiler optimization opportunities with the DEEP technique, (2) in the case where no additional compiler optimizations could be applied, we can effectively reduce the misprediction penalty by reducing the number of instructions which are speculatively executed before a `pb` instruction is verified by its corresponding `tv` instruction and (3) we can limit the static code size increase.

CHAPTER 3

GENERATING DEEPER PATHS

3.1 Motivation

The potential performance benefits which can be availed by increased straight-line code can be further exploited if we emit multiple, consecutive `pb` instructions at compile time. In doing so, we can generate longer paths of code that are branch-free and hence, not subject to control dependences. At the expense of additional code duplication, we can eliminate unconditional jumps to further increase basic block sizes. These longer paths will allow the sequential hardware prefetcher to more efficiently fetch instructions. Although the potential for increased optimization opportunities and runtime performance exists, the processor will remain in a speculative state for a longer time and may experience an increased misprediction penalty. Therefore, we limit this technique to loops only as (1) branches in loops have a tendency to be more predictable and (2) we hypothesize that most of the instructions executed in programs are within loops.

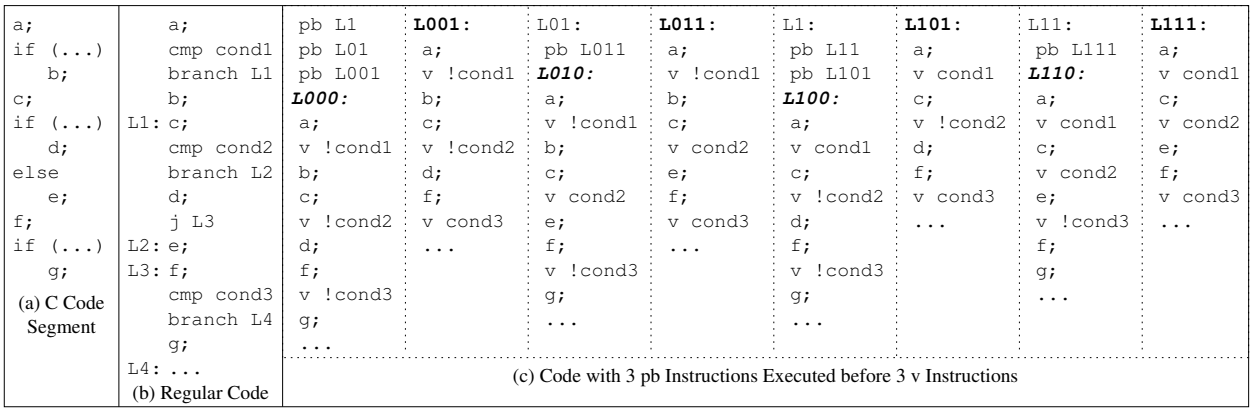

Figure 3.1: Code Generation with Multiple Consecutive `pb` Instructions

A compiler would have traditionally compiled the C code segment in Figure 3.1(a) into something similar to Figure 3.1(b). We propose to identify all possible paths by sinking the code between two consecutive `pb` instructions to the beginning of both immediate successor basic blocks of the second `pb` instruction. Although this idea could be exploited across multiple `pb` instructions, we

presently limit it to 3 consecutive explicit predict and branch instructions to prevent exponential code growth and to limit increased misprediction recovery times. In Figure 3.1(c) we identify all 8 possible paths through the program. We have abbreviated the `tv` instruction as `v`. Each path is preceded by a label in ***bold italics***, indicative of the branch prediction bits required to reach the path where a 0 indicates not taken and a 1 indicates taken. The branch prediction bits are read from left to right, where the first bit corresponds to the first `pb` instruction and the last bit corresponds to the last `pb` instruction. For example, the path starting with label `L000` can only be reached when all 3 branches are predicted as not taken (000). Similarly, the path starting with label `L101` can only be reached when the 3 branches are predicted as taken, not taken and taken respectively.

3.2 The DEEPify Algorithm

We achieve this extended path selection through the DEEPify routine within VPO. The DEEPify routine must meet certain algorithmic requirements, namely (1) it cannot cross loop boundaries, (2) it cannot push down function calls into speculative execution, (3) it cannot place direct or indirect unconditional jumps between interleaving `tv` instructions, (4) it cannot push a partial word store into speculative execution if a `tv` instruction relies on a subsequent memory access and (5) it should not place more than 25 instructions between `tv` instructions. The recursive algorithm relies on loop control flow information to properly execute as we only target explicit branches within innermost loops. Given a set of up to 3 `pb` instructions, the algorithm identifies all possible paths between them by following the control flow graph for both fall thru and taken successors of each branch.

Figure 3.2 shows the context under which the recursive `paths_identify()` algorithm is called. The algorithm will iterate through all basic blocks in the function being optimized, and for each `pb` instruction encountered within an innermost loop, attempt to identify all possible execution paths (up to 3 consecutive `pb` instructions). After all paths have been identified by the `paths_identify()` function, the algorithm proceeds to modify the control flow by performing code duplication through a call to the `make_paths()` function. It is important to first identify all possible paths prior to modifying the control flow to not include any newly introduced basic blocks within identified paths. Lastly, any unreachable code is eliminated and control flow information is checked.

```
FOR each PB DO
     /* split in control flow */
     paths_identify(block, NULL, &paths);

     /* perform code duplication */
     make_paths(&paths);

     /* simplify control flow */
     remv_useless_code();
     do_needed_analysis();
DONE
```

Figure 3.2: The `deepify()` Algorithm

```
FOR each block in path DO
     copy_block(block, path1);
     copy_block(block, path2);
DONE

FOR each successor of blk DO
     WHILE left child exists DO
          add_block_to_path(path);
          IF block has PB THEN
               paths_identify(block, path, &paths);
     DONE
DONE
```

Figure 3.3: The `paths_identify()` Recursive Algorithm

Figure 3.3 shows the recursive algorithm used to identify all possible execution paths. The intuition is to recursively invoke the `paths_identify()` function every time a fork in the control flow is encountered (by a `pb` instruction). When a `pb` instruction is encountered, 2 new paths are created and inherit all blocks in the parent path, as shown by the first `FOR` loop in Figure 3.3. The function then proceeds to follow the control flow graph for both newly created paths, as denoted by the second `FOR` loop in Figure 3.3. Note that the algorithm in Figure 3.3 does not include terminating condition checks, such as function calls, crossing loop boundaries, limiting paths to 3 consecutive `pb` instructions, limiting the total number of instructions within paths or limiting the number of interleaving instructions between test and verify instructions (to prevent possible machine deadlocks from lack of available reorder buffer entries or reservation stations).

The DEEPify algorithm is invoked after all traditional branches have been converted to explicit

predict and branch instructions, but before any additional compiler optimizations are attempted. This maximizes the chance of creating longer straight-line paths of code. To simplify the algorithm, branches which have negative offsets (such as loop back edges) are not considered. To prevent machine deadlocks, we do not allow more than 25 instructions to be placed between consecutive `tv` instructions. Depending on the reorder buffer (ROB) size of an OoO processor, there may not be enough slots to fetch and execute the next `tv` instruction if there are too many interleaving instructions, leading to a deadlock. The algorithm may be further tweaked depending on the benchmarks being simulated. For example, benchmarks with highly predictable branches may be able to chain up to 4 consecutive `pb` instructions at the expense of additional code duplication whereas benchmarks with low prediction rates may not want to invoke the DEEPify routine at all.

CHAPTER 4

HARDWARE SUPPORT FOR DEEP

In order to support the DEEP execution paradigm, we had to introduce changes to both the out-of-order (OoO) superscalar machine description and to the MIPS instruction set architecture (ISA). All hardware changes were made in ADL, the programming language used to write the simulator machine description, which is then compiled into C++ [5].

4.1 MIPS ISA Extensions

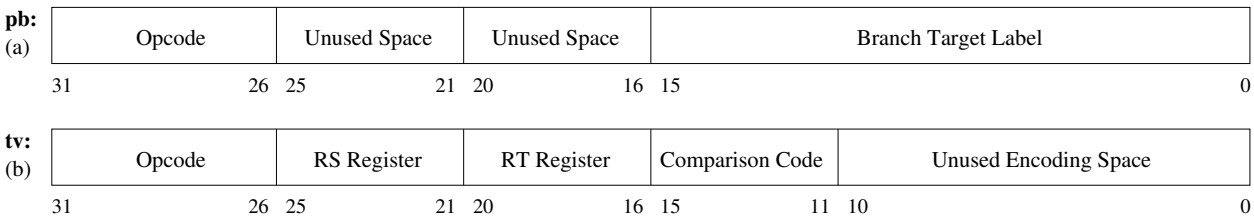

Figure 4.1: `pb` and `tv` Instruction Encodings

4.1.1 The `pb` Instruction

The predict and branch (`pb`) instruction is encoded as an I-Type instruction as shown in Figure 4.1(a). Note that since we have decoupled the branch prediction from verification, we no need to encode the `rs` and `rd` registers as the corresponding branch comparison will be performed by the `tv` instruction. This leaves 10 encoding bits free for future instruction annotations, if needed. Presently, they are all set to be 0.

4.1.2 The `tv` and `tvf` Instructions

It was necessary to introduce two different types of test and verify instructions: `tv` and `tvf`. The `tv` instruction operates on integer register operands and uses the integer unit whereas the `tvf` instruction operates on double and single precision floating point register operands and uses the floating point coprocessor. Both of these instructions will be collectively referred to as the

`tv` instruction throughout this thesis. The test and verify (`tv`) instruction is encoded as an R-Type instruction as shown in Figure 4.1(b). Note that because the test and verify instruction implicitly updates the verification hardware queue, there is no need to encode an `rd` register, which a traditional compare instruction would have encoded using bits 11-15. Instead, the `tv` instruction uses these same 5 bits to encode the type of comparison which it must perform. Tables 4.1 and 4.2 show the different types of comparisons which can be performed by the `tv` and `tvf` instructions respectively. Note that because there are never more than 32 such comparison types, 5 bits suffices to encode this field. We do not emit instructions for the `>` or `>=` operators as they can be derived by switching the register operands and by using the `<` or `<=` operators.

Table 4.1: `tv` Instruction Operations

`tv` instruction		
Mnemonic	Operation	Comparison Code
`tveq`	`==`	1
`tvne`	`!=`	2
`tvlt`	`<`	3
`tvle`	`<=`	4
`tvlu`	`<` (unsigned)	5
`tvleu`	`<=` (unsigned)	6

Table 4.2: `tvf` Instruction Operations

`tvf` instruction			
Mnemonic	Operation	Precision	Comparison Code
`tveqs`	`==`	single	7
`tvnes`	`!=`	single	8
`tlts`	`<`	single	9
`tles`	`<=`	single	10
`tveqd`	`==`	double	11
`tvltd`	`<`	double	13
`tvned`	`!=`	double	12
`tvled`	`<=`	double	14

The modified VPO compiler emits the mnemonics in Tables 4.1 and 4.2 in its MIPS assembly output. However, these mnemonics are implemented as macro expansions to the base `tv` and `tvf`

instructions using the `Condition Code` field in hardware. In doing so, we can avoid utilizing 14 different opcodes and instead only use 2.

4.2 Superscalar Extensions

4.2.1 Hardware Queues

The implicit recovery mechanism proposed in DEEP makes use of 3 special hardware queues. These queues are the mechanism used to communicate between `pb` instruction contexts and their corresponding `tv` instruction contexts. The queues required are:

- A prediction queue
- An alternate target queue
- A verification queue

The prediction queue is only ever accessed by the `pb` instruction and stores a 1 for a taken prediction or a 0 for a not taken prediction. The verification queue is only updated by the `tv` instruction, however it is the `pb` instruction which initially pushes a 0 verification bit onto the verification queue during the ID stage to indicate that it has not had its corresponding `tv` instruction executed. When a test and verify instruction finishes execution, it will update the verification queue with either a 1 to signal the prediction was correct or a -1 to signal that the prediction was incorrect. Since instructions are fetched and decoded in order, we can always guarantee that the same queue index is used to access the prediction queue and its corresponding verification bit in the verification queue. Note that we can easily support OoO execution by letting `tv` instructions index into the verification queue and update their associated verification bit.

Due to the large instruction window size of modern day OoO superscalar processors, these hardware queues need to be sufficiently large to accommodate multiple fetches of `tv` and `pb` instructions. This is especially important when an OoO superscalar processor encounters a tight loop and continues fetching multiple loop bodies worth of instructions. We used a circular buffer with a maximum size of 256 elements in our hardware implementation, a parameter which may vary with a processor's instruction window size. Although we used a queue of integers, we could have compressed our data structure to use 2 bits per entry as we only need to store 3 states: (1) unverified, (2) incorrect and (3) correct.

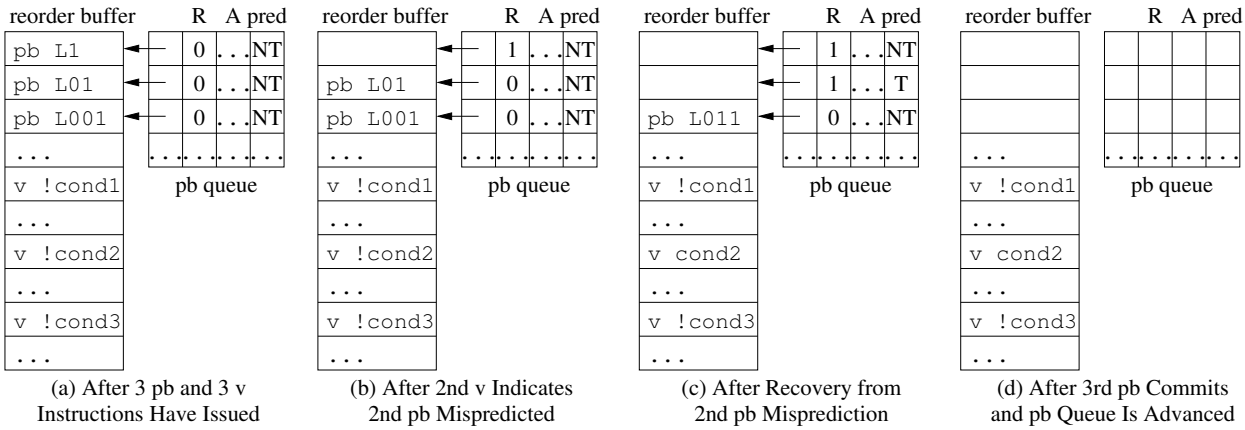

Figure 4.2: Hardware Queues

When a `pb` instruction is decoded by the processor, its prediction is pushed onto the prediction queue, the alternate branch target is pushed onto the alternate target queue and a 0 bit is pushed onto the verification queue. To accommodate the speculative instruction fetching nature of today's superscalar processors, all three hardware queues need to support both a speculative and non-speculative state. This allows the processor to easily rollback to its previous correct state in the case of an exception. Exceptions can arise when an unconditional jump misses in the BTB or an indirect unconditional jump modifies the instruction fetch pointer where subsequent instructions would have been erroneously fetched past either transfer of control instruction.

Figure 4.2 shows how the processor can implicitly recover from explicit branch mispredictions. We have abbreviated the `tv` instruction as `v` in Figure 4.2. Figure 4.2(a) shows the state of the reorder buffer (ROB) and the verification, alternate target and prediction queues after fetching and decoding multiple `pb` and `tv` instructions, represented by R, A and *pred* respectively. Since instructions may execute out of order, each hardware queue entry index corresponds to the same index in the ROB, as denoted by the arrows in Figure 4.2. The verification queue is updated as `tv` instructions execute and `pb` instructions are allowed to commit (and removed from the ROB) when their corresponding `tv` instruction has executed as shown in Figure 4.2(b). Figure 4.2(c) shows the state of the processor after recovering from a misprediction of the second `pb` instruction. Note that the *pred* queue bit is flipped from NT to T, and the `pb` instruction is allowed to commit. Once all predictions have been verified, all 3 hardware queues can be flushed as show in Figure 4.2(d). Both the `pb` and `tv` instructions can only ever transition into their respective non-speculative queue

state at the retire stage, as instructions commit in order. In addition to marking itself as being non-speculative, a test and verify instruction proceeds to remove itself and its corresponding `pb` instruction from all hardware queues as there is no longer a need to communicate across both of these instruction contexts.

4.2.2 Instruction Decode (ID) Stage

When a traditional conditional branch is fetched by a processor, it simultaneously accesses the BP and BTB in the IF stage. If the instruction misses in the BTB, then the BP is forced to issue a not taken prediction as the processor will not know if this is a branch instruction until the instruction decode stage in addition to not knowing the branch target. Today's OoO superscalar processors have very high branch prediction rates [6]. Consequently, we believe it is more beneficial to force every `pb` instruction to hit in the BTB to prevent issuing a not taken prediction when a taken prediction would have otherwise been issued by the BP. To this end, on a BTB miss, the `pb` instruction updates the BTB at the earliest stage possible, the ID stage. By updating the BTB at the ID stage instead of the RFU stage, we can issue an exception much quicker to flush the pipeline and to begin fetching instructions beginning at the same `pb` instruction so that we can now predict taken.

4.2.3 Retire (RFU) Stage

In addition to the 3 specialized hardware queues, we had to modify the retire stage of the OoO superscalar processor. We cannot allow a `pb` instruction to commit without having executed its corresponding `tv` instruction. Since instructions are fetched, decoded and retired in order, regardless of execution order, it suffices to check the head of the verification queue to see if the first `tv` instruction has verified the branch prediction. In the case that the corresponding `tv` instruction has not marked the prediction as being verified, we do not commit the `pb` instruction and stall in the retire stage. Conversely, if the corresponding `tv` instruction has executed, we can mark the `pb` instruction as complete and allow it to retire. Note that even if the `tv` instruction has identified a misprediction as show in Figure 4.2(c), we can still allow the `pb` instruction to retire as a branch can only have one of two outcomes (the opposite of the initial prediction value must be true).

In the case of a misprediction, the `tv` instruction will signal the `pb` instruction to raise an exception through the hardware verification queue akin to that which would have been raised by a

traditional branch instruction on a misprediction. The `pb` instruction will then flush the pipeline and the processor will begin fetching instructions at the alternate target (as given by the alternate target queue). While we intend to convert all traditional branches to `pb` instructions, the proposed hardware and compiler modifications allow `pb` instructions to coexist with traditional branches. This may be advantageous in the case where some functions of benchmark code consist of highly unpredictable branches, such as regions which are heavily reliant on user input. Such functions can be compiled without the DEEP optimization.

CHAPTER 5

EVALUATION

5.1 Experimental Setup

To gather results, we ran 17 benchmarks from the MiBench benchmark suite, all of which are written in C. These benchmarks are representative of a wide variety of embedded applications as shown in Table 5.1. To gather baseline statistics, all 17 benchmarks were compiled using the Very Portable Optimizer (VPO) compiler with all optimizations enabled, except the DEEP optimization. The compiler's back end was configured to target the MIPS ISA. Similarly, all 17 benchmarks were recompiled with VPO, using the same optimizations as the baseline compilation, in addition to enabling the DEEP optimization. The VPO compiler operates on a machine independent intermediate code representation format known as register transfer lists (RTLs). It takes a C input file to the front end and emits RTLs to the back end, where all optimizations are performed. Finally, the back end emits, in this case, MIPS instructions. All statistics were gathered with a simulated MIPS OoO superscalar processor (FAST) [5].

Table 5.1: 17 MiBench Benchmarks

Domain	Benchmarks
automotive	bitcount, qsort, susan
consumer	jpeg, tiff
network	dijkistra, patricia
office	ispell, stringsearch
security	blowfish, rijndael, php, sha
telecom	adpcm, CRC32, FFT, GSM

Only the benchmarks in Table 5.1 were compiled with the DEEP optimization. The standard library was only compiled with all of VPO's optimizations enabled, except DEEP. We believe that most performance benefits are to be found within actual benchmark code as opposed to standard library code. We did not run a larger set of benchmarks, such as SPECINT, to gather our results. All graphs in the following section were normalized as ratios to the baseline results.

5.2 Results

5.2.1 No Consecutive `pb` Instructions

We were able to achieve a decrease in the total number of cycles executed in 10 of the 17 MiBench benchmarks simulated after applying the DEEP optimization. Figure 5.1 shows the total number of cycles executed as a normalized ratio where a ratio of less than 1.0 indicates a performance benefit and a ratio of greater than 1.0 indicates a performance degradation. The ratios in Figure 5.1 were obtained without issuing multiple, consecutive `pb` instructions at compile time. On average, we observed an increase of 0.40% in the total number of cycles executed. In the case of the *bitcount* benchmark, we were able to achieve a decrease of 4.385% in the total number of cycles executed. The *susan* benchmark however witnessed a 9.828% increase in the total number of cycles executed. We believe this performance decrease can be attributed to a doubly nested `for` loop executed within the `susan_smoothing` function in the *susan* benchmark which contains test and verify instructions which are scheduled late by the compiler. Additionally, there are multiple memory accesses which are performed speculatively prior to these test and verify instructions, further increasing the misprediction penalty.

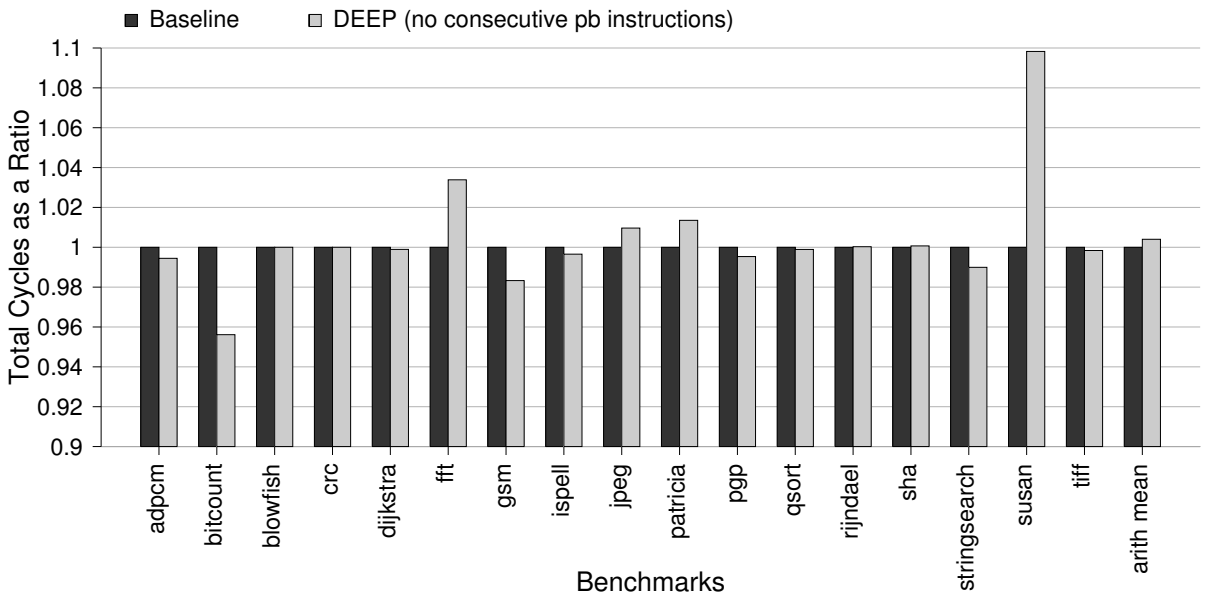

Figure 5.1: Cycles Without Consecutive `pb` Instructions

The limited performance benefit availed by the DEEP optimization can be attributed to multiple reasons, namely: (1) an increase in the number of unconditional jumps introduced as a direct result of promoting IF-only constructs to IF-ELSE constructs, (2) not collapsing `beq` and `bne` instructions at the ISA level like a MIPS processor would with `==` and `!=` relational operations and (3) increased instruction cache (icache) misses as a direct result of static code size increase. We do not believe that the number of increased icache misses is a major contributing reason due to the size of today's large cache sizes. Similarly, we believe that the number of additional execution cycles caused by not collapsing `beq` and `bne` instructions to be responsible for a negligible performance decrease.

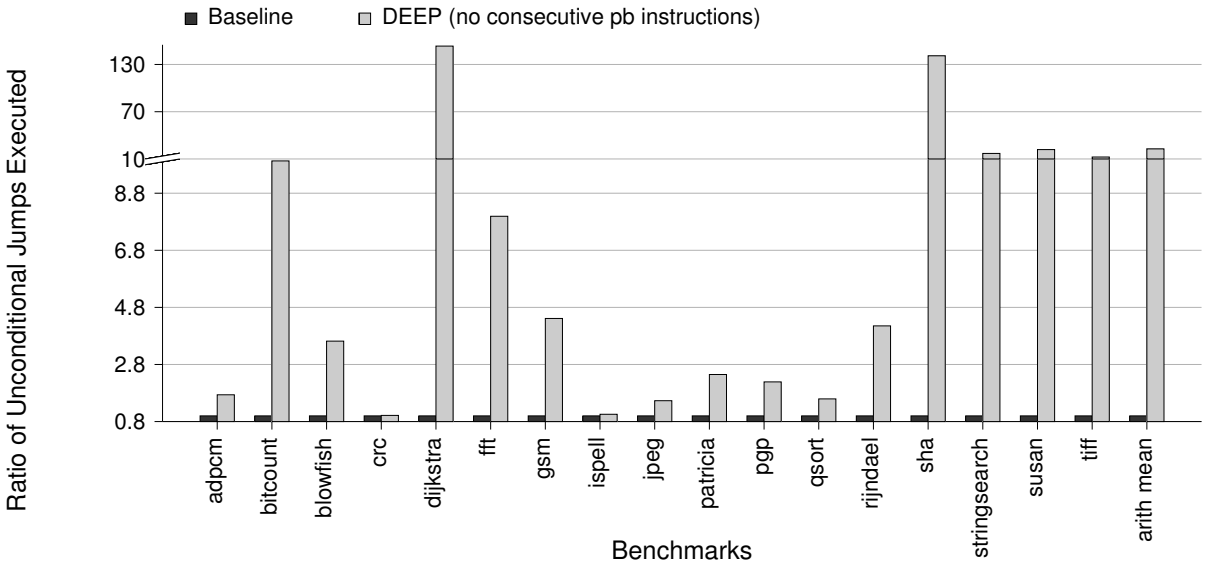

Figure 5.2: Number of Unconditional Jumps Executed Without Consecutive `pb` Instructions

Figure 5.2 shows the normalized ratio of unconditional jumps executed for all 17 simulated MiBench benchmarks without issuing multiple, consecutive `pb` instructions at compile time. The graph's y-axis breaks at 10 to accommodate the large increases in unconditional jumps within several benchmarks (including the arithmetic mean). As hypothesized, the number of unconditional jumps drastically increased across all benchmarks as a direct result of promoting IF-only constructs to IF-ELSE constructs. In some cases, the number of unconditional jumps executed increased by a factor of over 150 as in the case of the `dijkstra` benchmark. We believe that this large increase in unconditional jumps is a direct result of many benchmarks executing IF-only statements within

loops as an additional unconditional jump must be executed. Despite an average increase by a factor of 22.8 in unconditional jumps, we were still able to obtain a decrease in total cycle counts on some benchmarks. This is most evident in the *bitcount* benchmark where we were able to achieve a 4.385% decrease in the total number of cycles executed but increased the number of unconditional jumps by a factor of 9.93.

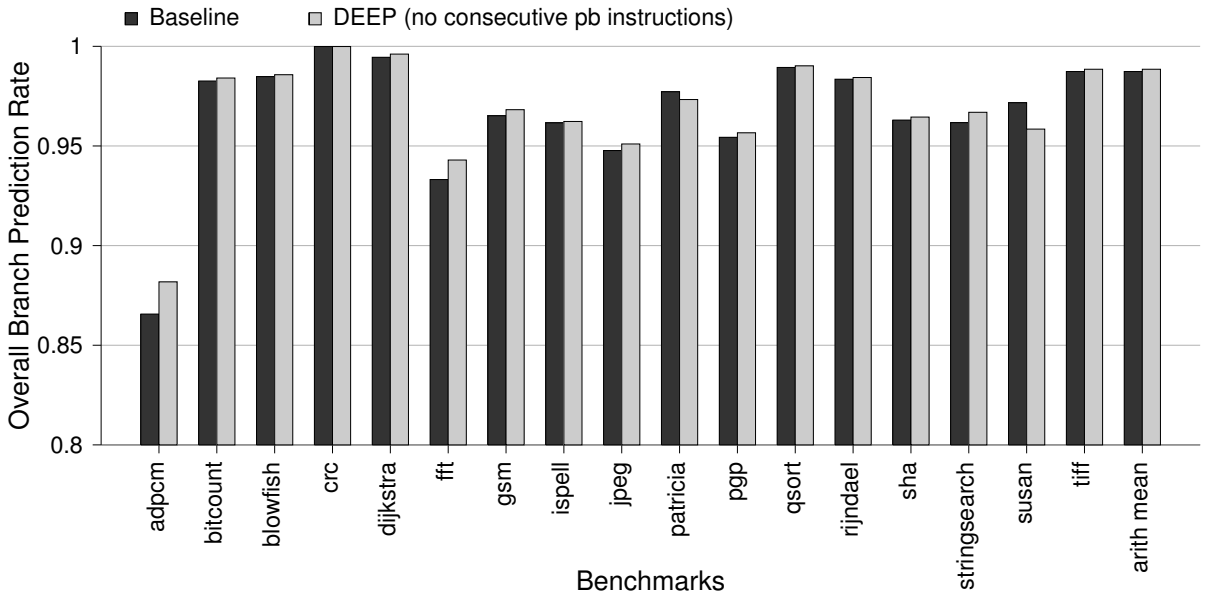

Figure 5.3: Overall Branch Prediction Rates Without Consecutive `pb` Instructions

We observed that overall branch prediction rates improved as a direct result of the number of additional unconditional jumps introduced. Figure 5.3 shows the overall branch prediction rates without issuing multiple, consecutive `pb` instructions at compile time. The average overall branch prediction rate increased by 0.115%. With the exception of the *crc*, *patricia* and *susan* benchmarks, all benchmarks saw an increase in overall branch prediction rates. Note that we consider both conditional and unconditional jumps when calculating the overall branch prediction rate. Simulation results showed that overall conditional branch prediction rates slightly decreased. We believe the conditional branch prediction rate decreased as the branch predictor state may take longer to update at the retire stage as `pb` instructions may have to stall additional cycles to allow their corresponding `tv` instruction to verify branch predictions. In comparing overall branch

prediction rates to the total number of cycles lost, we observed an inverse relationship between better prediction rates and total number of cycles executed.

As a direct result of increased overall branch prediction rates, we observed that most benchmarks saw a decrease in the total number of cycles lost. We define cycles lost as the number of cycles spent performing useless operations, such as a `pb` instruction stalling at the retire stage or wasted cycles as a result of recovering from branch mispredictions. Figure 5.4 shows the number of cycles lost as a normalized ratio. On average, we were able to achieve a decrease in the number of lost cycles by 3.814%. In the case of the *bitcount* and *patricia* benchmarks, we witnessed a decrease of 29.704% and 26.904% in the total number of cycles lost, respectively. The significant overall branch prediction rate decrease in the *bitcount* benchmark helps to explain its overall decrease in total cycle counts. However, the *ispell*, *susan*, *fft*, *jpeg* and *sha* benchmarks witnessed an increase in the number of cycles lost by up to 7.303%.

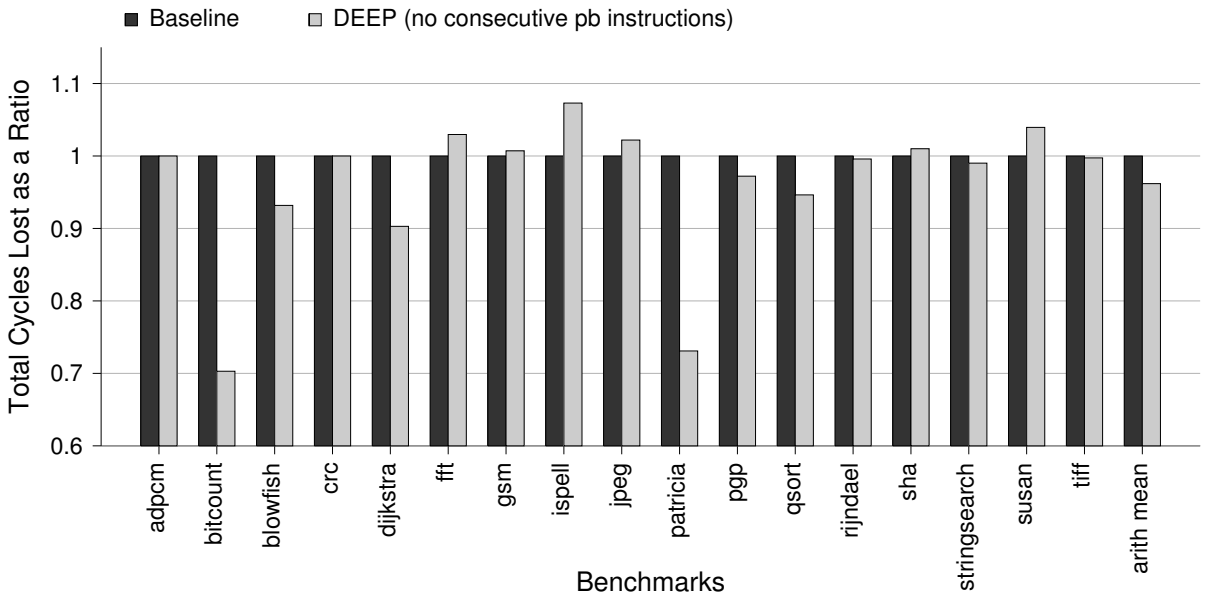

Figure 5.4: Branch Misprediction Stall Cycles Without Consecutive `pb` Instructions

The increase in static code size is shown in Figure 5.5. The increase in code size is expressed as a ratio of the number of RTLs in benchmarks after applying the DEEP optimization (without consecutive `pb` instructions) to the number of RTLs in benchmarks before applying the DEEP optimization. These statistics were gathered at compile time and do not account for pseudo instruction

macro expansions (such as expanding the *la* instruction into the *ori* and *lui* instructions). On average, the static code size increased by 18.105%. Interestingly, there does not appear to be a direct correlation between the total number of execution cycles avoided and static code size increase as both the *bitcount* and *susan* benchmarks saw a similar static code size increase (12.864% and 13.469%) yet exhibited a performance benefit and degradation respectively.

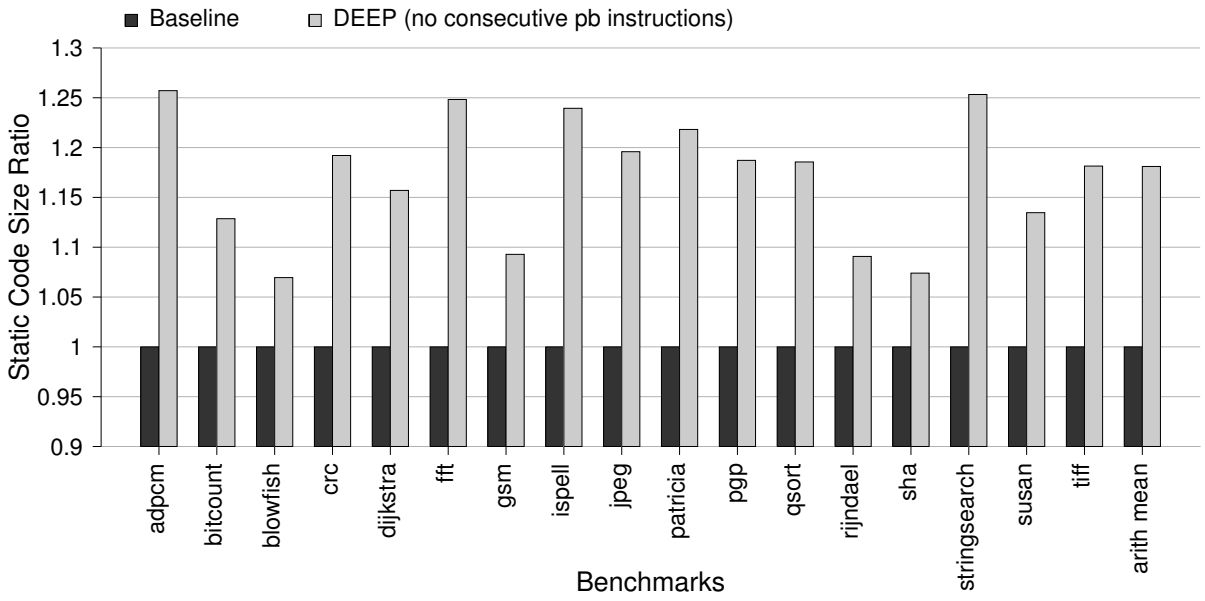

Figure 5.5: Static Code Size Without Consecutive `pb` Instructions

5.2.2 Up to 3 Consecutive `pb` Instructions

We observed that when we allowed the compiler to issue multiple, consecutive `pb` instructions, nearly all performance benefits were diminished. Figure 5.6 shows the total number of cycles executed as a normalized ratio. On average, the total number of cycles executed increased by a factor of 0.570%. Only the *gsm* and *patricia* benchmarks saw a reduction in the total number of cycles executed over the case of issuing non consecutive `pb` instructions. The remaining 15 benchmarks witnessed minor performance decreases. We believe that these performance decreases can be resolved by collapsing multiple explicit predict and branch instructions into a single instruction and through the use of a dedicated hardware paths array discussed in Chapter 7.

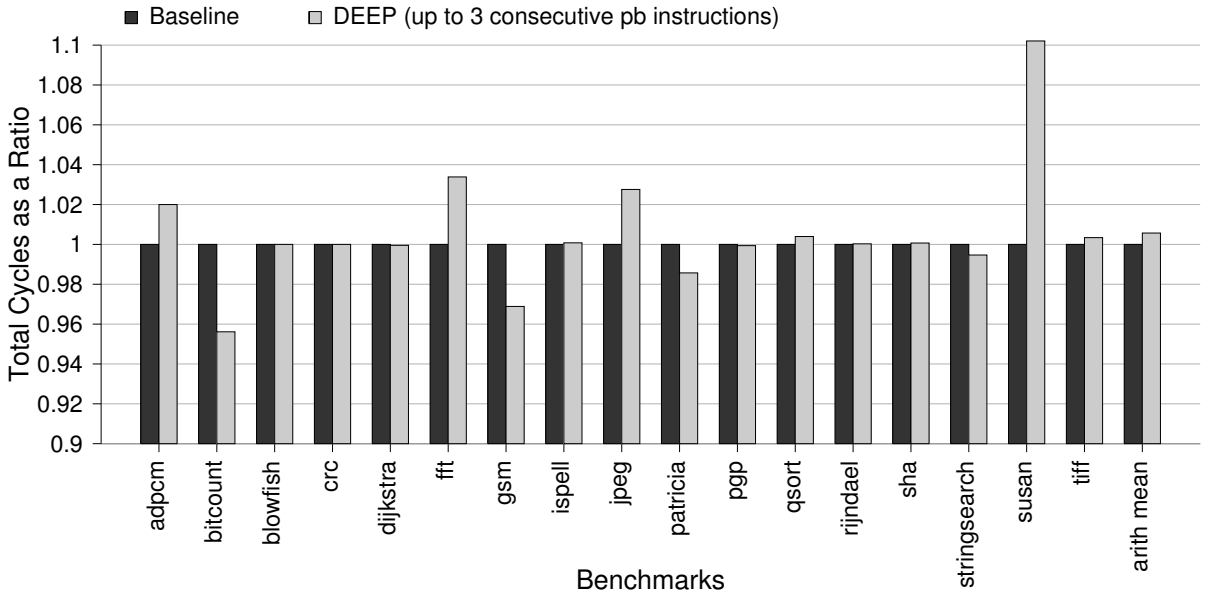

Figure 5.6: Cycles with Consecutive `pb` Instructions

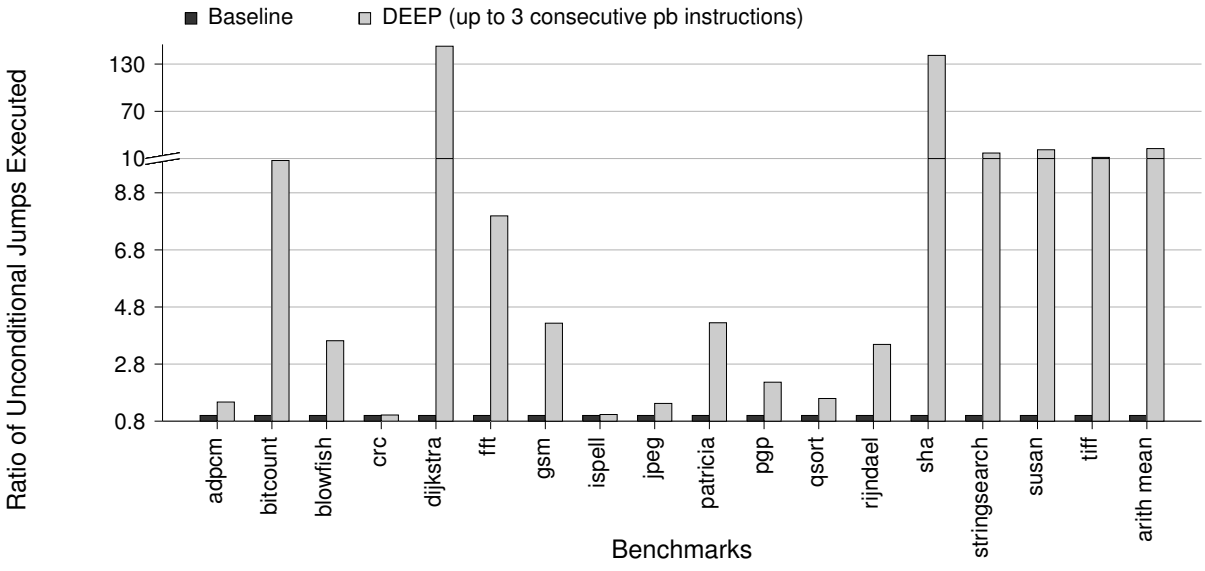

Figure 5.7: Number of Unconditional Jumps Executed with Consecutive `pb` Instructions

Figure 5.7 shows the number of unconditional jumps executed with up to 3 consecutive `pb` instructions. We believe that the additional costs associated with these unconditional jumps can

be mitigated by the dedicated hardware paths array discussed in Chapter 7. Figure 5.8 shows the overall branch prediction rates for executions with up to 3 consecutive `pb` instructions. These measurements closely resemble those obtained without consecutive `pb` instructions. On average, branch prediction rates improved slightly likely due to the number of additional unconditional jumps introduced by the compiler.

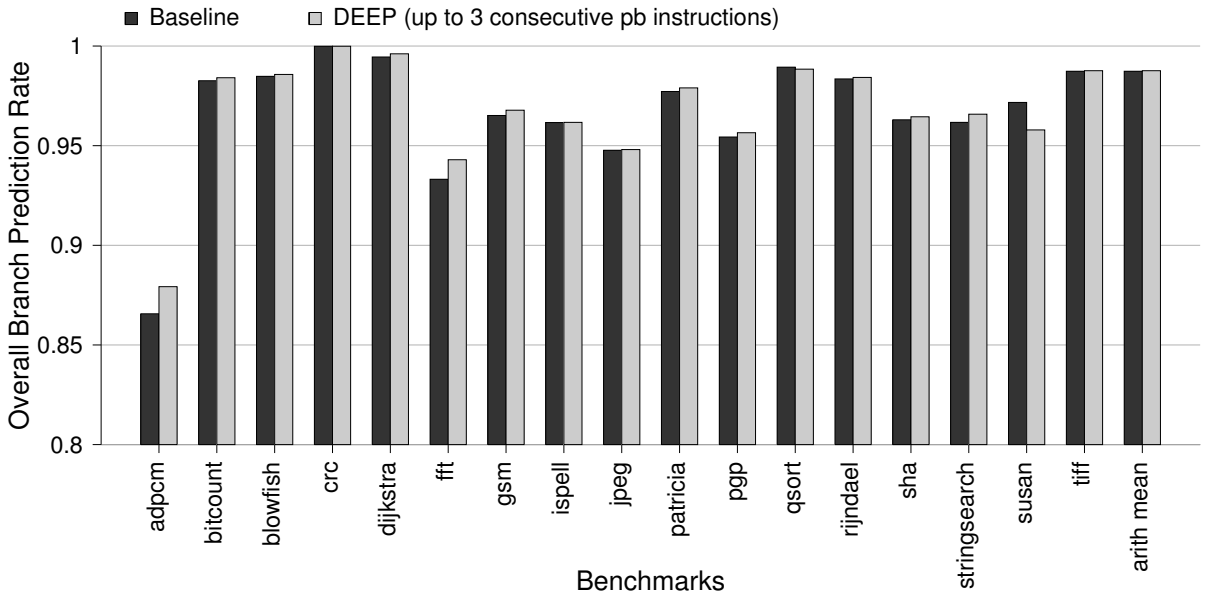

Figure 5.8: Overall Branch Prediction Rates with Consecutive `pb` Instructions

Figure 5.9 shows the number of cycles lost as a normalized ratio with consecutive `pb` instructions. These results are nearly identical to those obtained without issuing consecutive `pb` instructions. Figure 5.10 shows the increase in static code size after applying the DEEP optimization with up to 3 consecutive `pb` instructions. The average static code size increased by 44.453%, whereas the increase was only 18.105% without consecutive `pb` instructions as shown in Figure 5.5. The *susan* benchmark experienced the largest static code size increase, increasing by a factor of 2.752. This large increase can be attributed to the large basic blocks within the *susan* benchmark which are copied by the DEEPify algorithm. Although we did not apply the DEEP optimization to a larger test suite such as SPECINT, we believe the potential performance benefits can be much larger as there were not many opportunities within the MiBench benchmark suite where we could apply the DEEPify algorithm to issue consecutive `pb` instructions within innermost loops at compile time.

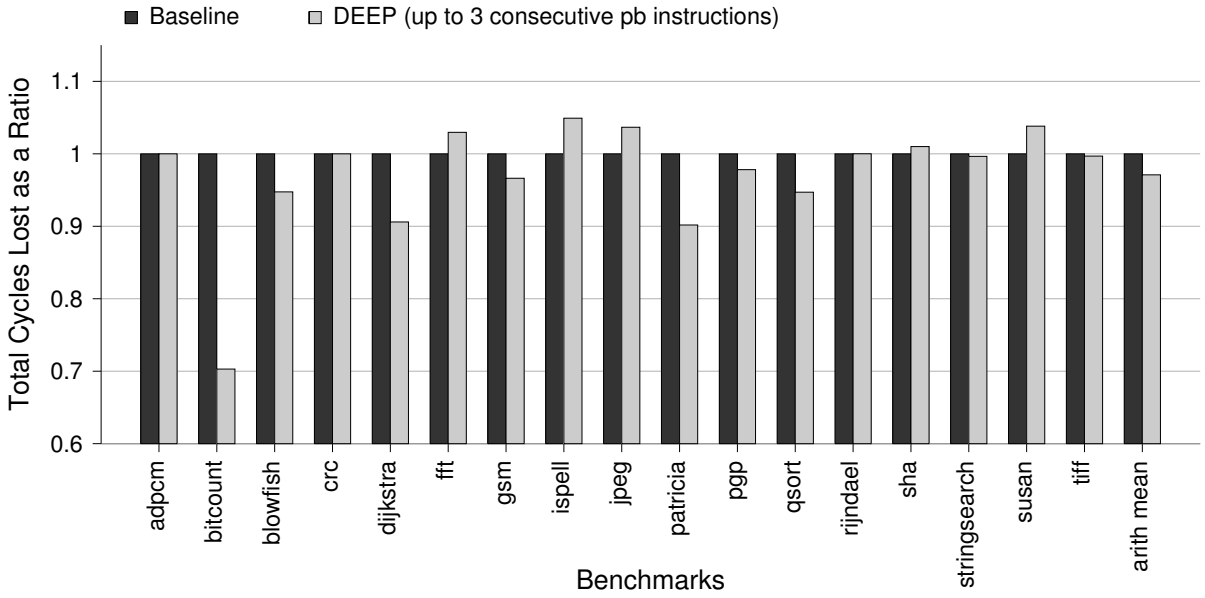

Figure 5.9: Branch Misprediction Stall Cycles with Consecutive `pb` Instructions

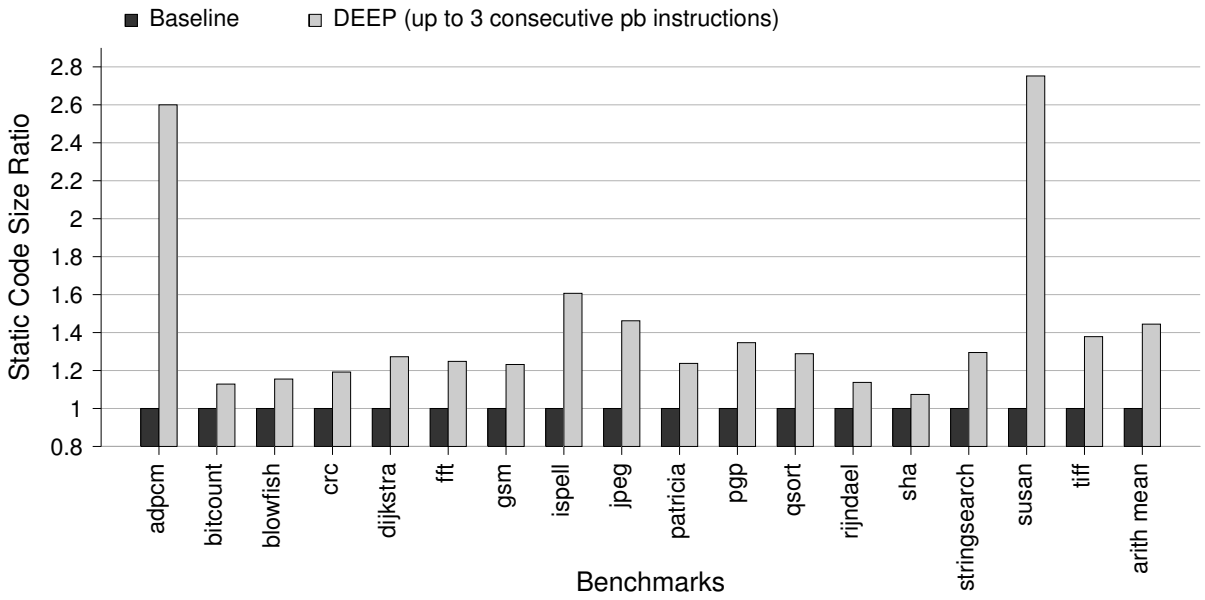

Figure 5.10: Static Code Size with Consecutive `pb` Instructions

CHAPTER 6

RELATED WORK

To the best of our knowledge, an extremely limited amount of work has been conducted into decomposing branch instructions to explicitly perform a predict and branch operation and to perform a verification with a later instruction. One such work that we are aware of is *Branch Vanguard*. McFarlin *et al.* [4] propose a similar method to the DEEP technique to eliminate control dependences. However, *Branch Vanguard* limits the code transformation to only highly predictable branches as identified by profile data whereas we propose to apply the DEEP optimization to all branches as we think the additional opportunities for optimizations will outweigh the slightly increased misprediction recovery time. Further, in *Branch Vanguard*, the authors require the use of an in-order processor and propose the generation of explicit recovery code, which further increases static code size. In contrast, our recovery mechanism seeks to emulate the implicit recovery mechanism of traditional branches within an OoO processor. By using an implicit recovery scheme, we do not need to generate additional recovery paths, further simplifying our optimization and limiting code growth. Moreover, we propose the creation of paths with multiple consecutive `pb` instructions to potentially give rise to additional opportunities to apply compiler optimizations within loops. *Branch Vanguard* did not support this feature likely due to the complexity associated with their explicit recovery code.

There have been other proposed approaches to decouple the prediction and verification operations of conditional branches. One approach uses a prepare-to-branch instruction in addition to a conventional branch instruction that requires an extra instruction to be executed for each conditional branch, which impacts code size and possibly performance [7]. Another approach proposed to use a set of branch registers to hold branch target addresses and a set of instruction registers to hold branch target instructions, allowing branch target address calculations to be hoisted out of loops and to be eliminated by common subexpression elimination [8]. This approach requires a delay slot that is hard to fill with multi-issue processors.

Multiple compiler optimizations which make use of code duplication to eliminate conditional branches have also been explored in the past. Such compilation techniques include duplicating code when there is a path to a nonloop branch where the branch result is known [9, 10]. These techniques result in even greater code size increase for eliminating a small number of branches. Our explicit branch prediction approach limits code size increase by hoisting identical instructions from both successors of a conditional branch after applying the DEEP optimization, common subexpression elimination, dead assignment elimination and instruction selection. In hoisting identical instructions from speculative execution to non-speculative execution, we effectively reduce static code size increase and decrease the branch misprediction recovery time.

CHAPTER 7

FUTURE WORK

7.1 Paths Array

7.1.1 Overview

We can further increase the potential performance benefits availed by multiple, consecutive `pb` instructions inside innermost loops by collapsing consecutive `pb` instructions into a singular branch instruction and by making use of a dedicated hardware paths array. The paths array is a special hardware queue to associate path predictions with target path addresses. Prior to entering a loop, we propose to populate the paths array queue with all of its potential target path addresses through a loop. We can then perform an efficient indirect jump to addresses stored in the paths array each loop iteration based on the predictions issued by a multibit branch predictor.

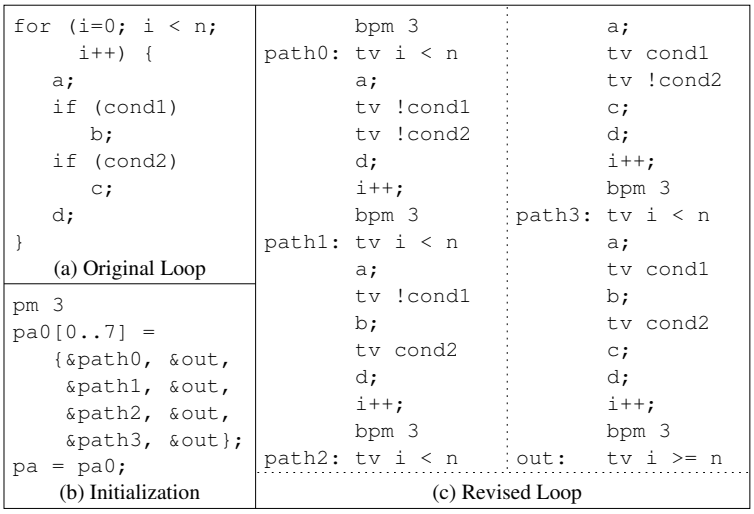

Figure 7.1: Path Selection in Innermost Loops

Figure 7.1(b) shows the paths array initialization in the pre-loop basic block. This serves to associate a PC-offset at which each path through the loop begins. The *prediction register* is represented by the `pa` variable and is updated with multiple prediction bits issued by the `bpm` and `pm` instructions. Figure 7.1(c) shows the modified loop after all 4 possible execution paths

through the loop have been identified. The first path prediction is issued by the `pm` instruction prior to assigning path target addresses to the paths array. Note that each path ends with a `bpm` instruction. This instruction will branch to the path target address stored in the paths array as indexed by the *prediction register* and then issue 3 new prediction bits. Further, note that every path also must exit the loop through the `out` label to ensure the correct number of loop iterations have been executed. We believe the potential performance benefits of path selection using prediction can be significant if innermost loop branches are highly predictable.

Due to the associated overhead with creating and populating the paths array hardware structure, we only propose that this method be applied on innermost loops as the preloop overhead will be far outweighed by the loop body execution. We believe this technique has the potential to significantly increase performance as a significant portion of branches execute within innermost loops as Figure 7.2 shows. We anticipate the most performance benefit will be seen by paths which contain multiple taken branches such as path `L111` in figure 3.1. Instead of requiring two jumps before reaching the desired path, we can immediately jump to the beginning of this path with the paths array approach.

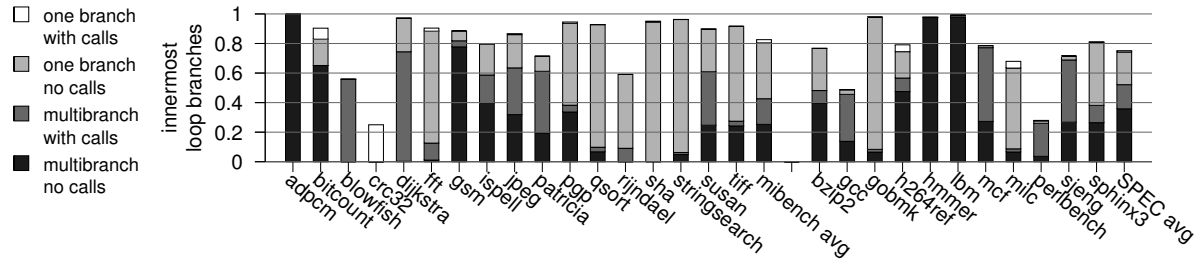

Figure 7.2: Fractions of Branches Executing in Innermost Loops

7.1.2 MIPS ISA Extensions

We propose some new instructions to support this approach: (1) the branch and predict multiple (`bpm`) instruction, (2) the predict multiple (`pm`) instruction, (3) the test and verify clear (`tvc`) instruction and (4) the assign path (`ap`) instruction. Unlike the `pb` instruction, the `bpm` instruction does not need to store a target label, freeing up additional encoding bits for future instruction annotations. Potential encodings of these instructions are shown in Figure 7.3. The difference between the `bpm` and `pm` instructions is that a `bpm` instruction will first branch based on the value

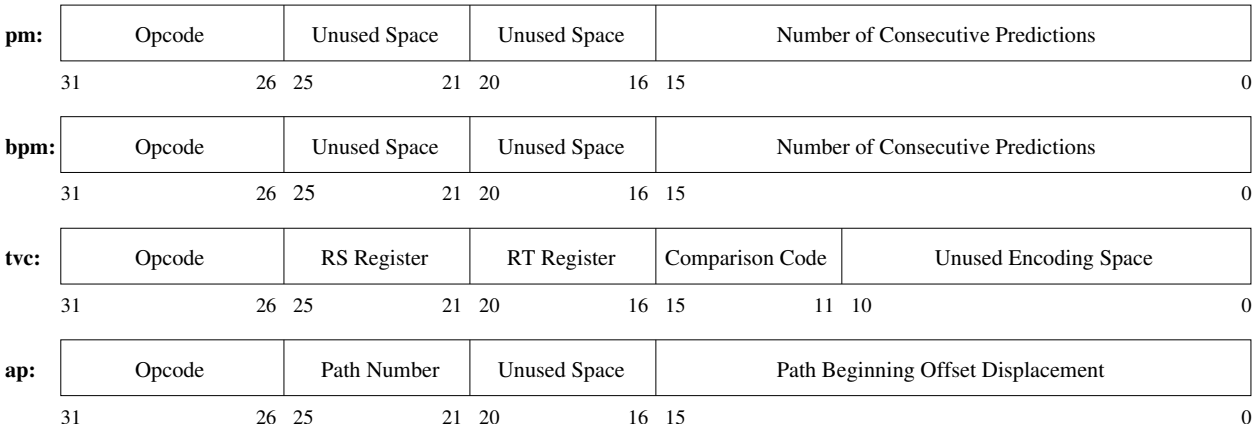

Figure 7.3: Potential `pm`, `bpm`, `tvc` and `ap` Instruction Encodings

stored in the *prediction register*, access the BP and finally update the *prediction register*, whereas the `pm` instruction will only access the BP and update the *prediction register*. Since both of these instructions only need to store a prediction count, they can be encoded as I-Type instructions. Note that since we limit the number of consecutive branch predictions which a `pm` or `bpm` instruction can make to be a small number (3), these instructions could also use an R-Type instruction encoding and encode the number of consecutive predictions where the RS, RT or RD register index would have been traditionally encoded. However, we think it would make most sense to use an I-type encoding to simplify instruction decoding in hardware.

The test and verify clear (`tvc`) instruction is needed to support cases where there are a variable amount of `pb` instructions amongst the identified paths. This case can arise when there are interleaving function calls within one path but not among another path. Much like the `tv` instruction, the `tvc` instruction will verify the first bit(s) in the *prediction register* but it will also mark the remaining prediction bit(s) as verified, effectively nullifying any outstanding prediction bit(s). This allows us to issue a `pm 3` instruction and support a path with only 1 or 2 verification instructions. The `ap` instruction will be used to associate path indexes with target path addresses and will be inserted into the preheader basic block of innermost loops. Loop invariant code motion can hoist these `ap` instructions out of an entire loop nest.

7.1.3 Superscalar Extensions

The paths array hardware structure is implemented as a simple queue in hardware where the queue index is the path index (as determined by multiple prediction bits) and the value is the PC displacement of the beginning of the path relative to the `ap` instruction as shown in Figure 7.4. Since we apply this transformation after the DEEPify routine, we are guaranteed that there are at most 3 consecutive `pb` instructions in any given path. If we then interpret the path prediction bits as a binary number (i.e. the path where all 3 predictions are taken would be 111 or 7), we will at most need 8 entries in the paths array hardware structure. Whenever a `bpm` instruction is encountered, it uses the *prediction register* to index into the paths array to jump to the appropriate path address target as indicated by the last multibit prediction issued by the BP. The `bpm` instruction then proceeds to shift in a new set of prediction bits into the *prediction register*.

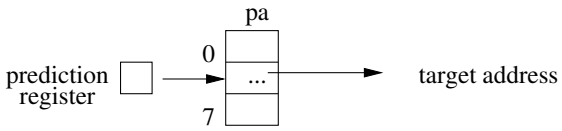

Figure 7.4: Hardware Support for Path Selection

Since both the `bpm` and `pm` instructions signal hardware mechanisms (i.e. branch and set the new instruction fetch pointer or issue multiple branch predictions), they need not perform any calculations during the EX stage. Instead, once they are decoded, they can be marked as complete, contingent on the their multiple issued predictions being verified by their corresponding test and verify instructions. To our knowledge, the proper implementation of a single cycle, multibit branch predictor requires complicated hardware support. Rather than relying on additional hardware support, we propose to issue branch predictions over multiple instructions. For example, if a `pm 3` or `bpm 3` instruction is fetched we can access the BP for the first prediction with the `pm` or `bpm` instruction. We can issue the second branch prediction with the next instruction fetched. Similarly, we can issue the last prediction with the instruction fetched at PC+8 of the original `pm` or `bpm` instruction. This approach allows us to use the conventional BP without introducing additional hardware complexities at the expense of a couple of additional cycles. The predictions can be made as long as the target paths are at least two instructions, a requirement which the compiler can satisfy at compile time.

CHAPTER 8

CONCLUSIONS

Conditional branches can cause expensive pipeline flushes and occur with high frequency in programs. Hence, they are expensive instructions worth optimizing. The DEEP optimization eliminates control dependences in programs by decomposing branch instructions into an explicit predict and branch instruction and a test and verify instruction at compile time. This explicit branch decomposition eliminates the dependence between a branch instruction and its preceding compare instruction and allows instructions preceding branches to be sunk into both successors of conditional branches creating longer straight-line blocks of code. In addition, longer paths can be generated within innermost loops by issuing multiple, consecutive explicit predict and branch instructions before multiple test and verify instructions at compile time. In some cases we were able to successfully apply additional compiler optimizations on these straight-line paths of code. In the case where no additional compiler optimizations can be applied, identical instructions from both branch successor basic blocks can be hoisted before an explicit predict and branch instruction to limit the misprediction recovery penalty and to limit static code size increases. In this thesis, we have shown that we can achieve up to a 4.385% decrease in total cycle counts by applying the DEEP optimization.

BIBLIOGRAPHY

[1] Kreahling W., Whalley D., Bailey M., Yuan X., Uh GR., van Engelen R. (2003) Branch Elimination via Multi-variable Condition Merging. In: Kosch H., Bszrmnyi L., Hellwagner H. (eds) Euro-Par 2003 Parallel Processing. Euro-Par 2003. Lecture Notes in Computer Science, vol 2790. Springer, Berlin, Heidelberg.

[2] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. 2002. Efficient and effective branch reordering using profile data. ACM Trans. Program. Lang. Syst. 24, 6 (November 2002), 667-697.

[3] B. Black, B. Rychlik and J. P. Shen, "The block-based trace cache," *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, Atlanta, GA, 1999, pp. 196-207.

[4] D. S. McFarlin and C. Zilles, "Branch vanguard: Decomposing branch functionality into prediction and resolution instructions," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015, pp. 323-335.

[5] S. Onder. An introduction to Flexible Architecture Simulation Tool (FAST) and Architecture Description Language ADL. Technical report, Michigan Technological University Department of Computer Science, Michigan.

[6] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Fifth Edition.

[7] A. Bright, J. Fritts, and M. Gschwind. Decoupled fetch-execute engine with static branch prediction support. Technical Report RC23261, IBM Research Report, 1999.

[8] Jack W. Davidson and David B. Whalley. Reducing the cost of branches by using registers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 182-191, Seattle, Washington, May 28-31, 1990.

[9] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 56-66, La Jolla, California, June 18-21, 1995. *SIGPLAN Notices*, 30(6), June 1995.

[10] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 146-158, Las Vegas, Nevada, June 15-18, 1997. *SIGPLAN Notices*, 32(6), June 1997.

BIOGRAPHICAL SKETCH

Luis Penagos was born in Bucaramanga, Colombia on September 2, 1994. He received a B.S. in Computer Science from Florida State University (FSU) in May of 2017, graduating with the *summa cum laude* honors. As an undergraduate student, Luis worked under Dr. Whalley as an undergraduate research assistant gathering statistics for effective memoization opportunities within the VPO compiler and the FAST functional simulator. Additionally, Luis worked as a web technician for Florida State University for 4 years designing multiple departmental websites and applications. Luis briefly worked with Dr. Zhenghao Zhang in the Department of Computer Science to help translate a MATLAB active RFID identification algorithm to an equivalent C program to run on embedded systems.

He then proceeded to continue his education by pursuing an M.S. in Computer Science at Florida State University in May of 2017. Luis spent the summer of 2017 working under Dr. Whalley as a graduate research assistant and assisted a 10 week research trip to Gothenburg, Sweden to collaborate with Chalmers University of Technology faculty on compiler optimization research. In the summer of 2018, Luis was employed as a graduate teaching assistant, leading a recitation for Computer Organization II (CDA3101) and grading course projects. His research dealt with eliminating control dependences in code through explicit branch predictions and verifications. Luis maintains an active interest in the areas of compiler optimizations, computer architecture and wireless networks.