FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EXPLOITING GATING TO SELECTIVELY CONVERT CONTROL DEPENDENCIES TO DATA DEPENDENCIES

By

QUAN PHAM

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

2025

Quan Pham defended this thesis on November 5, 2025.
The members of the supervisory committee were:

David Whalley
Professor Directing Thesis

Soner Onder
Committee Member

Xin Yuan
Comittee Member

Gary Tyson
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 am sincerely grateful to Dr.David Whalley for his continued guidance, support, and encouragement throughout the course of this work. I would like to thank Dr.Soner Onder and Dr.Gang-Ryung Uh for their help in setting up the compiler and simulation environment, and for providing valuable feedback for my work. I also want to thank John (Jack) Dewey for helping me to implement the early version of the Gating algorithm and Jacob (Judas) Smith for implementing the block scheduling algorithm. Many thanks also go out to Kieran Young, Caleb Swain, Sarah Larkin, and Scott Pomeville for their help in setting up the testing environment, building the VIS simulator, suggesting improvements to the VIS micro-architecture, and debugging errors. This research was supported in part by the US National Science Foundation (NSF) under the grants CCF-1900788, CCF-1901005, OISE-2103103, OISE-2103105, CCF-2211353, and CCF-2211354. Any opinions, findings, and conclusions or recommendations expressed in this thesis may not reflect the views of the NSF.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Parallelism has been the main method to improve computer performance. Researchers have tried to exploit parallelism at different granularity: instruction-level-parallelism (ILP), thread-level-parallelism (TLP), and data-level-parallelism (DLP). While recent work has shifted focus to DLP and TLP, ILP is still very important for single thread performance. ILP has been exploited by different microarchitectural techniques such as superscalar and very-long-instruction-word (VLIW). Both of these methods have harnessed significant gains in Instruction-Per-Cycle (IPC), but progress has been slowed down because of control dependencies. The Predication technique has been developed to convert control dependencies to data dependencies, but it also has its own disadvantages. This thesis develops, implements and evaluates a program transformation to convert control dependencies using *gating* [11, 10, 2]. This transformation converts control dependencies into data dependencies on selected code regions, as opposed to whole program transformation, by vectorizing the register pool, and using a gate instruction to determine which register instances are used according to the predicate information.

CHAPTER 1

INTRODUCTION

Parallelism has been the main method to improve computer performance. Researchers have tried to exploit parallelism at different granularity: instruction-level-parallelism (ILP), thread-level-parallelism (TLP), data-level-parallelism (DLP). While recent work has shifted focus to DLP and TLP, ILP is still very important for single thread performance. ILP has been utilized by different microarchitectural techniques such as superscalar and very-long-instruction-word (VLIW). Both of these methods has harnessed significant gains in Instruction-Per-Cycle (IPC), but progress has been slowed down because of control dependencies.

1.1 Instruction-Level Parallelism

Instruction-Level-Parallelsim (ILP) is the ability of a processor to execute multiple instructions at the same time. Superscalar processor exploits ILP by issuing and executing multiple instructions in the same clock cycle. They do this by searching and dynamically scheduling independent instructions so that they can execute them at the same time. In superscalar processors, ILP is exploited at run-time, while in the VLIW processors, ILP is exploited at compile-time. In the VLIW method, the compiler determines which instructions are independent and groups them in a pack of fixed number of instructions. The VLIW processor can then issue and execute these pack of instructions without the need for dynamic scheduling.

1.2 Control Dependencies

Control dependencies are dependencies created by branches. When a processor encounters a branch, the next instruction to be executed following that branch is only known after the branch has finished executing. This limits the number of instructions that both superscalar and VLIW processors can execute at the same time.

When superscalar processors encounter a branch, they will predict whether that branch is taken or not using a run-time Branch Predictor. If the branch is predicted to be taken, the target of that branch also needs to be fetched using a Branch Target Buffer (BTB). The superscalar processor

will then assume the predicted path and target and continue issuing instructions on that path. If the branch is later determined to be mispredicted, the processor will need to squash all the wrong speculative instructions and redirect the fetch point to the correct target. The penalty of a branch misprediction depends on the pipeline depth, but on average this type of penalty creates a non-trivial slow down to the proccessor.

In the VLIW method, the compiler will schedule the branch instruction to be in the same pack with path-independent instructions. If the compiler can not find enough path-independent instructions to fully fill that pack, it will use nop instructions to fill those slots. When many packs need to be filled with nop instructions, the IPC of that proccessor will show significant reduction.

1.3 Gating Transformation

Gating [11, 2] is a mechanism which merges multiple versions of data values across control-flow paths, hence allowing conversion of control dependencies into data dependencies. Gated Single Assignment (GSA) and Future Gated Single Assignment (FGSA) represent the program in single assignment form, and in case of executable FGSA form an entire program can be represented without branches. In order to facilitate direct execution of single assignment form a vectorized instruction space processor model was developed by Prof. Soner Onder being investigated under the NSF grants supporting this thesis. In an instruction space vectorized architecture, each regular register is associated with a vector of `n` instances. A branch instruction can be converted into a compare instruction that will set a register instance of a predicate vector representing possible paths of execution. Control-dependent paths can be transformed into control-independent paths that write to different instance registers. After these new control-independent instructions, a gate instruction will be inserted to act as a multiplexer to select the instance register that matches the executed path represented in the predicate vector. When the control dependencies are converted to data dependencies, the compiler can reschedule the control-independent instructions.

CHAPTER 2

RELATED WORK

There were already many attempts to increase IPC in processor. Each method has its own strategy to sidestep the overhead presented by control dependencies, but each of them has its own trade-offs.

2.1 Trace Scheduling

Trace Scheduling was an optimization technique for scheduling instructions across basic block boundaries [3]. This method was built upon an earlier local compaction technique which focused on sheduling instructions within a basic block[5]. The main goal of the trace scheduling technique is to optimize the critical trace of a program so that processor's resources are maximally utilized along that path. The critical path of instructions, consisting of one or many blocks, is determined by profiling and heuristics, and a DAG is built to reflect the data-dependence of those instructions. Code motions rules would then be applied to move the instructions along the hot trace so that data dependencies are minimized and resource utilization is maximized. Because the trace scheduling algorithm moves instructions across branches, compensation code need to be generated to ensure data integrity when the program needs to execute the uncommon path. According to the author, this compensation code can be complicated to generate [3]. This scheduling technique helps the common path runs with minimum delays which noticeably increase IPC, but it does not efficiently handle the case when the processor needs to execute the uncommon path. Additionally, the common trace is determined by heuristics and profiling, so optimality is not guaranteed. Finally, this technique does not remove control dependencies, so the superscalar and VLIW processor still suffered from the control dependence bottleneck.

2.2 Super Block Formation

Super Block Formation was another scheduling technique attempting to improve the existing Trace Scheduling method [4]. A superblock can be viewed as a hot trace with no side entrances. By ensuring that there is only one main entrance and one or more exits, the corrective (bookkeeping) phase of this technique is less complex than Trace Scheduling. This method constructs the super

block by picking the hot trace of one or more basic blocks by heuristics and removing side entrances by using a tail duplication technique. A DAG will then be built from the instructions of the super block, and the scheduler will use that DAG to apply code motion rules to the instructions. This method does help noticeably improve IPC, but control dependencies caused by the side exits still exists and creates overhead. The hot trace is still determined by profiling, so there is no guarantee of optimal results.

2.3 Predication

Predication is an optimization technique that attempts to transform control dependencies into data dependencies [12]. The most general method to implement predication is to add an extra register field to every instruction. That extra field will hold a predicate register whose values can only be true (1) or false (0). The branch instruction will be transformed into a predicate-setting instruction. If the predicate register is true, the associated instruction will be executed and committed to change the processor states. Otherwise, the instruction will be executed, but not committed. This technique allows the branch to be eliminated, and the taken and non-taken paths of the branch can be merged into one basic block. This allows more freedom for the compiler to do scheduling without the need for compensation code when the processor goes off-trace such as in trace scheduling or super block formation. The overhead of predication was apparent from the fact that the processor needs to execute both paths of the branch. This overhead will be exaggerated if a branch is unbalanced - one of the two paths has much fewer instructions compared to the other. When an unbalanced branch is transformed into predicated instructions and the shorter path is the correct path, the processor will have to waste computing resources on the longer wrong path. The execution of the longer non-taken path also causes delays which hurts IPC. Compared to predication, the gating technique will provide a more effective way to handle unbalanced branches. Another overhead created by the predication technique is the true dependence between the predicate-setting instructions and the predicated instructions. Hence, the scheduler cannot move the predicated instructions above the predicate-setting instructions because the predicated instructions need the result of the predicate-setting instructions. Both of these limitations are later addressed by the Gating technique.

CHAPTER 3

INFRASTRUCTURE

This research is part of NSF Project entitled Vectorized Instruction Space (VIS). As part of the project, a 64 bit instruction word MIPS ISA, and a 64 bit instruction word Vectorized Instruction Space (VIS) ISA have been developed by Prof. Soner Onder. ADL simulators, including a functional and superscalar simulators, have been developed by Michigan Tech PhD students Caleb Swain, Kieran Young & Sarah Larkin. I implemented the Gating Transformation on this VIS ISA using the Assembly Optimizer (asopt) [7, 6], and simulated on the ADL simulator [8].

3.1 The Instruction Set Architecture (ISA)

In VIS ISA, each regular register is associated with a vector of 32 instances, each register field needs extra 5 bits to represent the instance number and 1 more bit represents whether that register is a regular register or a vector instance register. Every instruction supports the capability to read from or write to a vector instance register. An example of an `add` instruction using vector instance registers in the VIS ISA is `add $2[1],$4[2],$5[2]`. Every instruction can still use the regular register just like in original MIPS: `add $2,$4,$5`. Besides `seq, sne, slt, slti, sltiu` instructions, three more compare instructions will be added to help set the predicate register: `sle, sltei, sleiu`. The `gate` and `pgate` instructions are created to support the Gating Transformation. Besides the `and` instruction, the `cand` instruction is added to support exception handling for the Gating Transformation. Because each instruction now has 64 bits for encoding, load instructions can load directly from a global label without the need to construct the address using `lui` and `ori` instructions.

3.2 Compilation Infrastructure

The MIPS assembly is transformed into VIS assembly using our Assembly Optimizer (asopt) [7, 6]. The asopt program reads each function in MIPS and builds a control-flow graph based on the control flow of that function. Data-flow analysis is performed on each function, and the `sets`, `uses`, `deads` and `reaching-definitions` of each instruction are also calculated [1]. Optimization

techniques such as dead assignment elimination, reverse branches, useless jump elimination, etc. are also implemented in the asopt system. Each of these optimizations can be enabled or disabled by compilation flags. The Gating Transformation is also implemented within asopt to leverage its existing functionality.

3.3 Simulation Infrastructure

The VIS ISA is tested on an ADL simulator generated by Flexible Architecture Simulation Tool (FAST) [8, 9]. Microarchitecture simulators can either be hand-coded or auto-generated using a higher-level description language, such as Architecture Description Language (ADL). In this project, we chose the second option due to its ease of use and the convenience of the FAST toolchain. Kieran Young, Sarah Larkin, and Caleb Swain at Michigan Tech University have implemented the 64-bit VIS simulators. In ADL, microarchitectural constructs such as the register pool, functional units, and cache hierarchies can be composed to specify the desired processor. The ADL program would then be compiled via the FAST toolchain to generate an assembler, disassembler, and simulator written in C program. Due to the FAST toolchain, different features of the VIS ISA can be quickly prototyped along side many modifications to the microarchitectures in the simulators.

CHAPTER 4

GATING TRANSFORMATION OVERVIEW

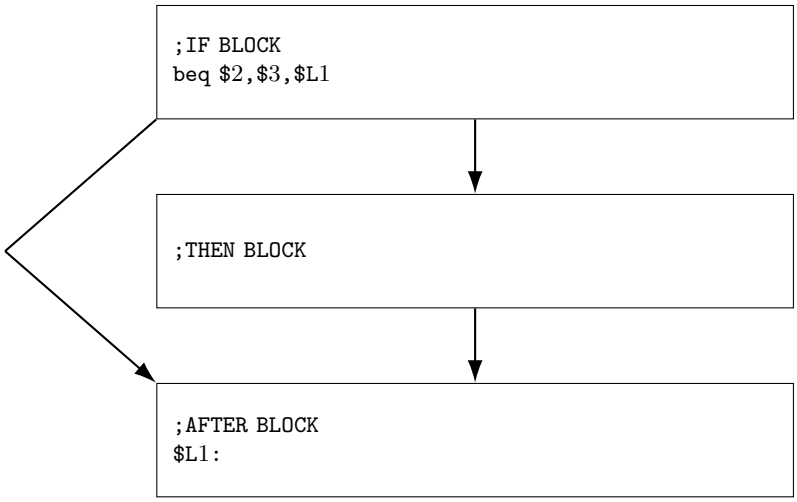

Figure 4.1: IF-THEN-AFTER structure

The main goal the of Gating Transformation is to eliminate conditional branches. A region of assembly instructions controlled by a branch will be called a branching region. The Gating Transformation algorithm will first categorize whether a branching region has the If-Then-After structure (Fig. 4.1), If-Then-Else-After structure (Fig. 4.2), or other structures. The current transformation algorithm can only work with If-Then-After and If-Then-Else-After structures. The branch instruction is converted into two predicate-setting instructions. In the traditional Predication technique, these two instructions set two completely different predicate registers, but in the Gating technique these two instructions set two instance registers associated with the same regular register. For example, instruction `beq $2,$3,$L1` would be transformed into `seq $1[0],$2,$3` and `sne $1[1],$2,$3` instructions. `$1[0]` will guard the jump path, and `$1[0]` will guard the fall-through path.

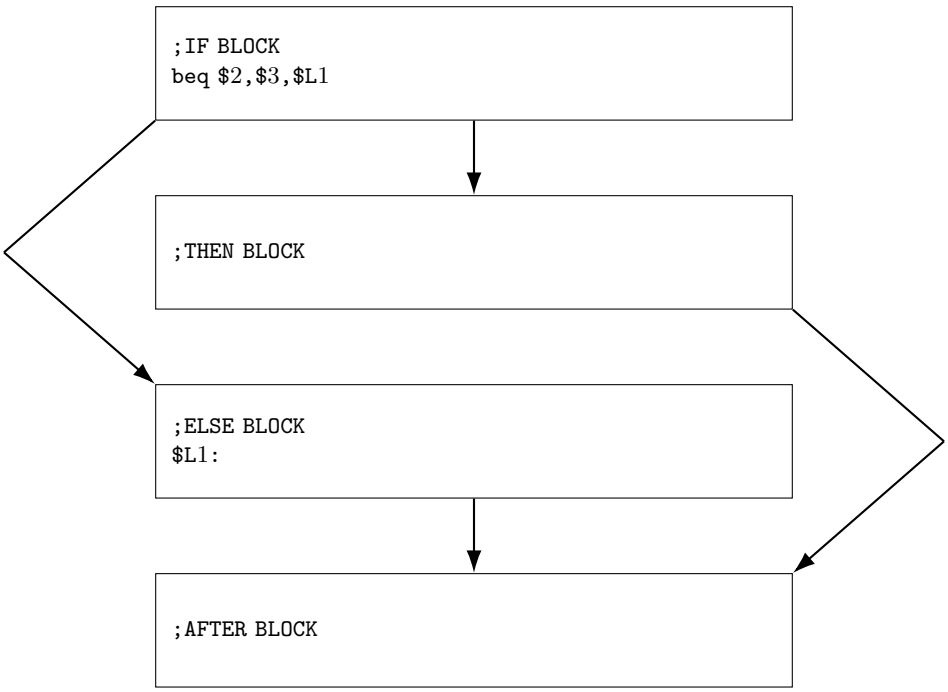

Figure 4.2: IF-THEN-ELSE-AFTER structure

4.1 Single-Level Gating Transformation

If a definition of a register is live outside the branching region, that definition will need to be gated. In Fig. 4.3, before the transformation, the definitions of register $4 and $5 are live outside the branching region - both of those registers are used in the `L1: add $7,$4,$5` instruction. Hence, those registers have to be gated. On the fall-through path which is guarded by vector instance register `$1[0]`, `addi $4,$6,1` instruction is transformed into `addi $4[1],$rename1,1` instruction, and `addi $5,$2,1` instruction is transformed into `addi $5[1],$2,1` instruction. On the jump path which is guarded by vector instance register `$1[1]`, `li $4,5` instruction is transformed into `li $4[0],5` instruction, and `li $5,6` instruction is transformed into `li $5[0],6` instruction. At the bottom of the branching region, the gate instructions will be inserted to choose the definition that is on the executed path. The structure of the `gate` instruction is as follows: `gate dst_reg,src_vect,pred_vect,start_idx,length`. The semantic of the `gate` instruction is described by the pseudo code below:

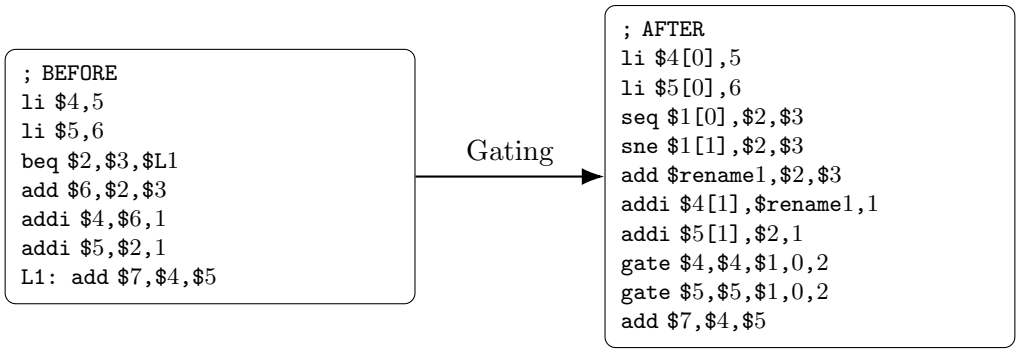

Figure 4.3: Single-Level Gate Transformation

```
for (int i = start_idx; i < (start_idx + length); i++) {
    if (pred_vect[i] == TRUE) {
        dst_reg = src_vect[i];
        break;
    }
}
```

The pseudo code might give the impression that the gate instruction sequentially checks the value of each predicate instance, but these checks are executed in parallel. In our superscalar simulator, the gate instruction will be dynamically split into `cmov` (condtional move) micro-operations, and each of these `cmov` instructions will be independently executed in parallel. On the example shown in Fig. 4.3, the `gate $4,$4,$1,0,2` instruction will select between the register instances `$4[0]` and `$4[1]`, and the `gate $5,$5,$1,0,2` instruction will select between the register instances `$5[0]` and `$5[1]`. The gate instruction is immediately executed when the `true` predicate and its corresponding register instance are known. If the original control flow follows the shorter path of a branch, then the gate instruction needs not wait for the instructions along the longer path to complete.

4.2 Single-Level PGating Tranformation

A store instruction cannot be gated like a regular instruction because a store instruction that is executed on the non-taken path would still update data in memory. We use a new instruction called `pgate` to handle store instructions. The structure of `pgate` instruction is as follows: `pgate dst_vec,src_reg,pred_vect,start_idx,length`. Each register will have a poison bit. The `pgate` instruction will set the poison bit of the instance register associated with the executed path to FALSE, and the poison bits of other instance registers will be set to TRUE. If an instruction

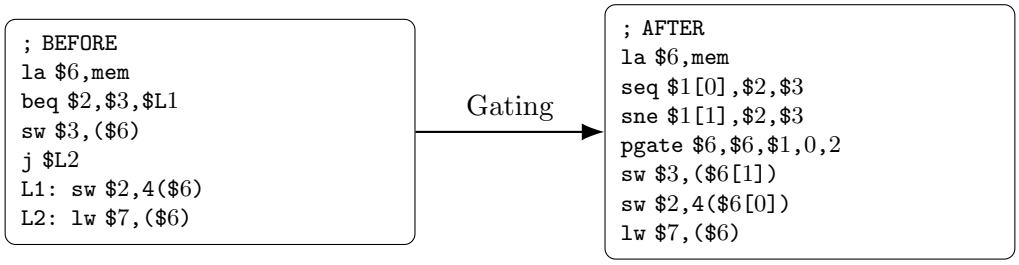

Figure 4.4: Single-Level PGate Transformation

reads from a `poisoned register`, it will not execute that instruction. If that instruction has a `destination register`, the poison bit of that register will also be set to TRUE. If an instruction reads from an `unpoisoned register` (poison bit = FALSE) and references an invalid memory address, an exception will be raised. An instruction that has a `poisoned register` and writes to a regular register will also raise an exception. The behavior of the pgate instruction is described by the pseudo code below:

```
for (int i = start_idx; i < (start_idx + length); i++) {
    if (pred_vect[i] == TRUE) {
        dst_vec[i].poison_bit = FALSE;
        dst_vec[i] = src_reg;
    }
    else {
        dst_vec[i].poison_bit = TRUE;
    }
}
```

In Fig. 4.4, the `sw $3,($6[1])` instruction is only executed if the poison bit of register $6[1] is FALSE. If the tested predicate has not yet been set, then the `pgate` instruction will delay until the predicate is known.

CHAPTER 5

ITERATIVE GATING TRANSFORMATION OVERVIEW

```
; BEFORE
li $4,5
beq $2,$3,$L2
add $6,$2,$3
addi $4,$6,1
beq $7,$8,$L3
li $4,6
$L3: j $L2:
$L2: addi $7,$4,5
```

Figure 5.1: Nested Branching Region

As mentioned before, the current Gating algorithm can only work with IF-THEN-AFTER and IF-THEN-ELSE-AFTER structures. Hence, we need to use an iterative approach to handle nested branching regions such as in Fig. 5.1 (BEFORE Gating). That same branching region can also be represented by basic blocks in Fig. 5.2.

5.1 Nested Multi-Level Gating Transformation

We can transform these nested regions by iteratively transforming the innermost branching region, and then proceeding to the outer branching regions. In Fig. 5.2, the THEN BLOCK, THEN_2 BLOCK and AFTER_2 BLOCK can form an IF-THEN-AFTER structure. Hence, we can apply gating and collapse this branching region into a basic block (Fig. 5.3). After that first transformation, the IF BLOCK, THEN BLOCK and AFTER BLOCK can form another IF-THEN-AFTER structure. Thus, we perform another gating transformation as shown in Fig. 5.4. We can see that there are two definitions of `$4` that reach the AFTER BLOCK: `li $4,5` and `gate $4,$4,$1,0,2` (Fig. 5.3). When a definition is a `gate` instruction, we perform Gate Chaining on that definition. That gate instruction would be removed, and the `sets` of that `gate` instruction will be renamed to be used for the new outer gate instruction (`gate $4,$4,$1,0,3`) which is gating the

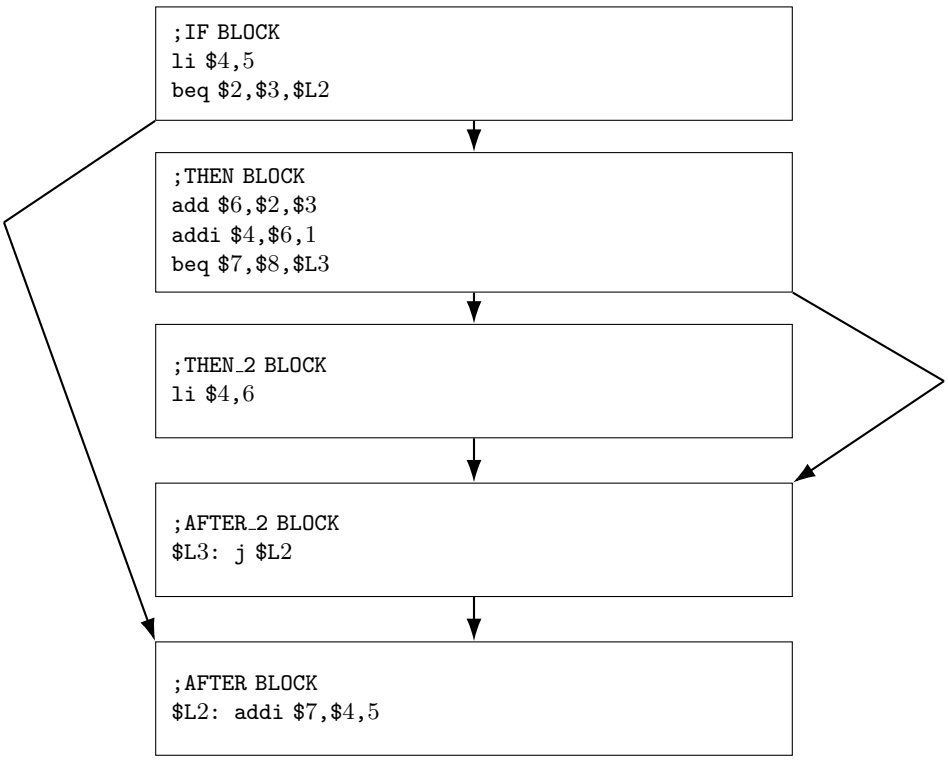

Figure 5.2: Nested Branching Graph

whole branching region. The corresponding predicate instances of the two definitions in the THEN BLOCK will also be chained using the `cand` instructions:

```
sne    $rename2,2$,$3
<other assembly lines>
seq    $rename3,$7,$8
cand   $1[1],$rename2,$rename3
sne    $rename4,$7,$8
cand   $1[2],$rename2,$rename4
```

The predicate `$1[1]` now represents the boolean expression `($2 != $3)` $\wedge$ `($7 == $8)`, and the predicate `$1[2]` now represents the boolean expression `($2 != $3)` $\wedge$ `($7 != $8)`.

There are also cases where the THEN BLOCK or the ELSE BLOCK have gate instructions in it, but Gate Chaining is not applied. The details of the Gate Chaining technique will be discussed in Section 6.5.2. A simple example where Gate Chaining is not performed because the inner `gate` instruction is not a definition of the outer `gate` instruction is shown in Fig. 5.5. We can see that

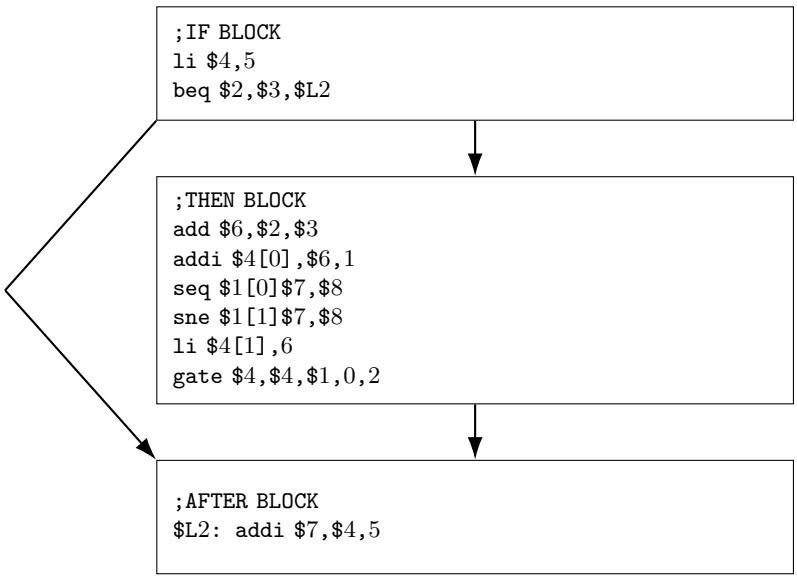

Figure 5.3: Nested Branching Graph After First Transformation

the predicate instances are not chained in this case, and the inner gate instruction is transformed to use instances 2 and 3 instead of instances 0 and 1 previously.

5.2 Nested Multi-Level PGating Transformation

When Gating can be applied to branching regions, if the inner THEN or ELSE BLOCK has a `pgate` instruction then Gate Chaining must be performed. The reason is because a `store` instruction in an original branching region that conditionally executed must always be guarded by a correct predicate expression. An example is shown in Fig. 5.6.

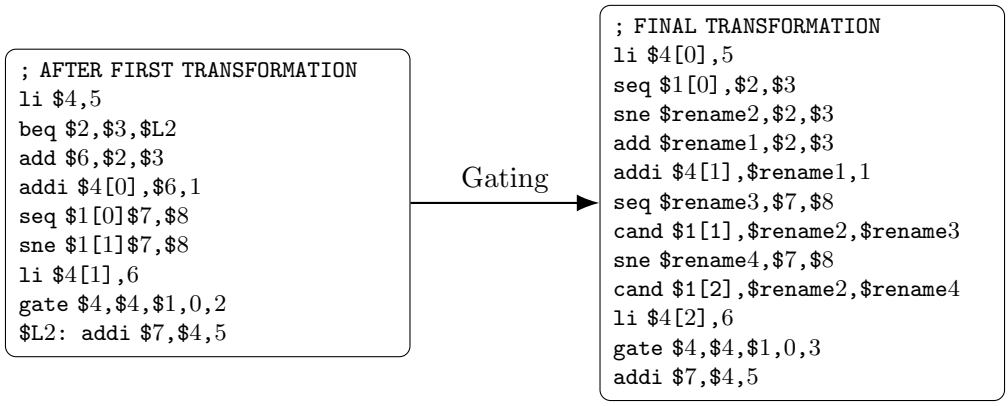

Figure 5.4: Nested Gating Final Transformation with Gate Chaining

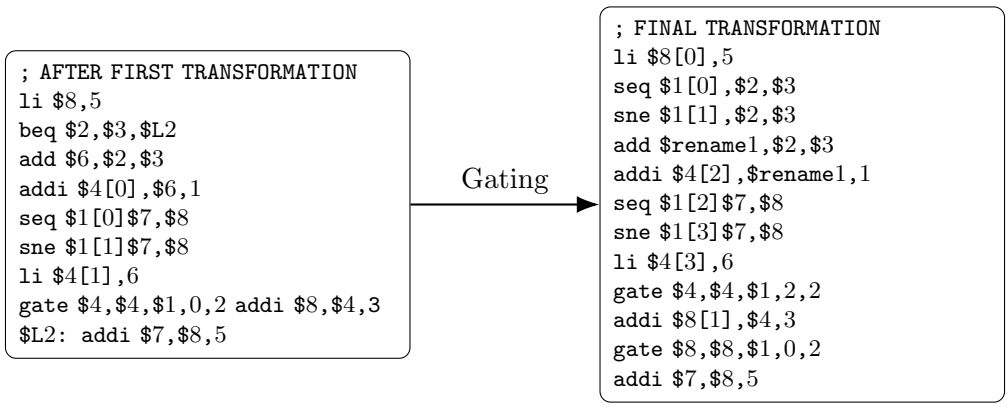

Figure 5.5: Nested Gating Final Transformation without Gate Chaining

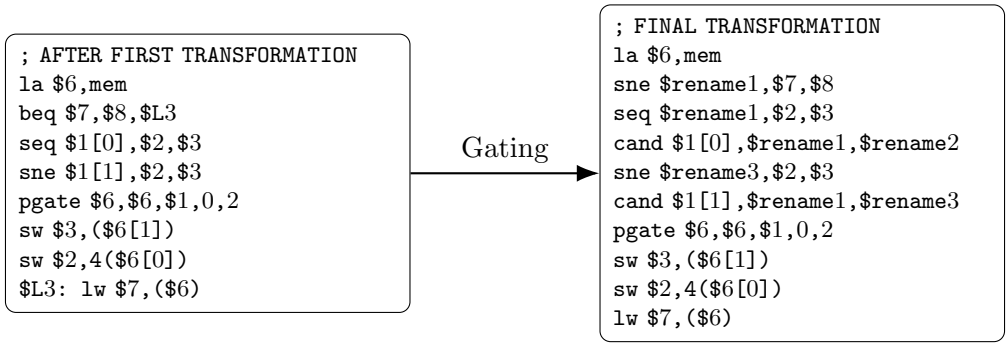

Figure 5.6: Nested PGating Final Transformation with Gate Chaining

CHAPTER 6

ITERATIVE GATING TRANSFORMATION IMPLEMENTATION

This section describes how we iteratively eliminate branches by using Gating. The Gating Transformation is implemented in the asopt system [7][6] using the C programming language.

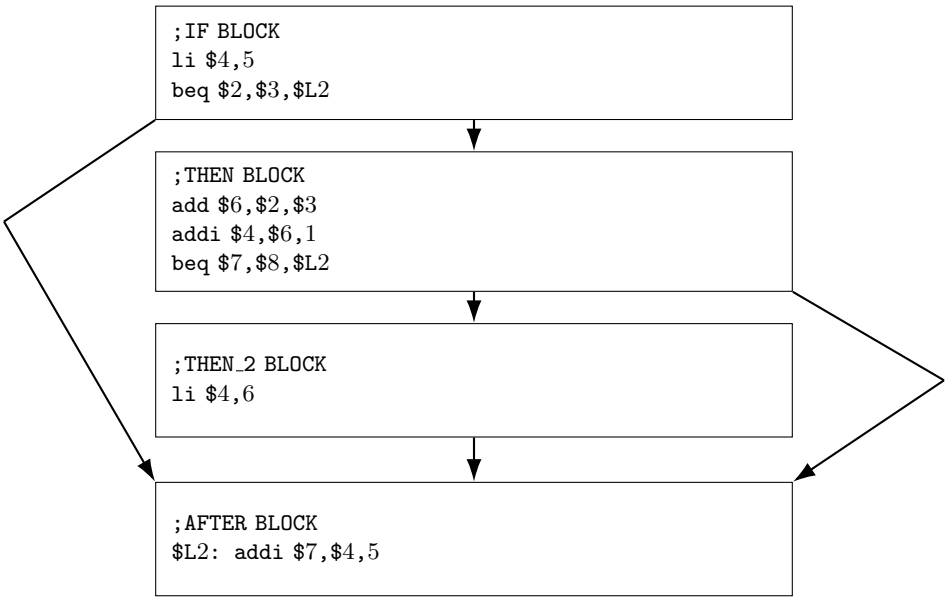

Figure 6.1: Nested Branching Graph with Invalid Structure

6.1 Detect Convertible Branches

A branching region is currently considered applicable for gating only if it has the IF-THEN-AFTER structure or IF-THEN-ELSE-AFTER structure. Additionally, the THEN BLOCK and the ELSE BLOCK cannot have peripheral `predecessor` or `successor` connections to any other block except the IF BLOCK and the AFTER BLOCK. The AFTER BLOCK can only have two predecessors. All the blocks in the branching region must belong to the same loop. Our current Gating algorithm is not supporting branching regions that cross loop boundaries. Some other

restrictions of the current algorithm include no gating for floating point branch instructions and no gating for function calling instructions. Some of these restrictions can possibly be lifted in the future versions of the Gating technique.

6.2 Construct Gate Group

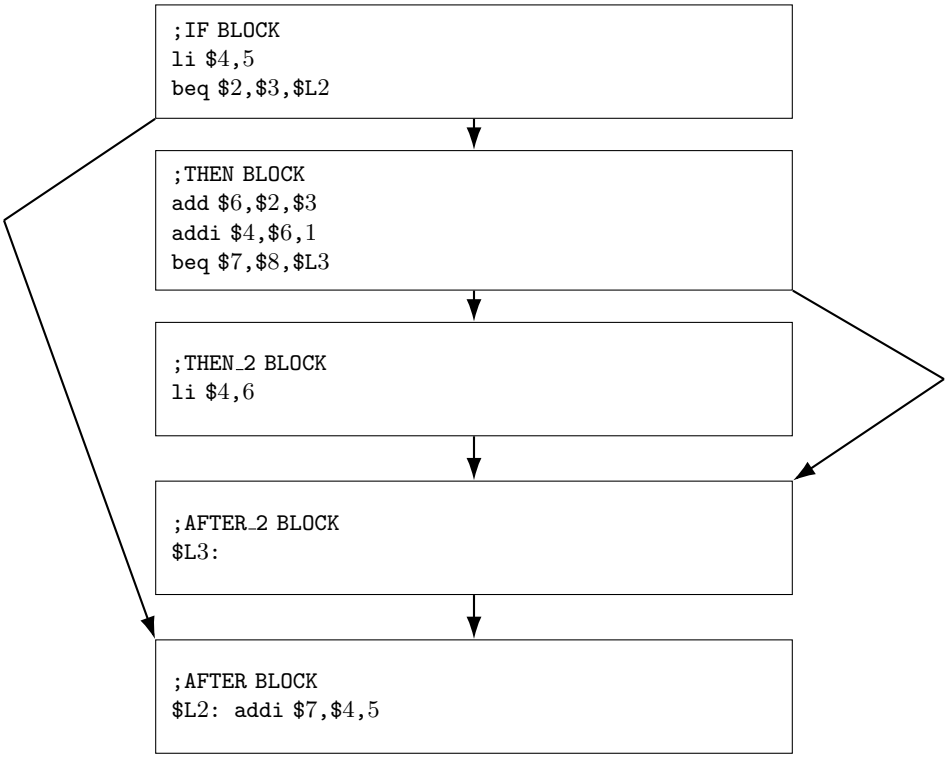

Figure 6.2: Nested Branching Graph After Inserting Inner AFTER_2 BLOCK

A Gate Group can be either a RGate Group (Register Gate Group) or MGate Group (Memory Gate Group). A branching region can have multiple RGate Groups or MGate Groups. An RGate Group represents the different definitions of a register that reaches the AFTER BLOCK and its corresponding predicate expression. An MGate Group represents the different store instructions that uses the same memory register in the branching region and its predicate guards. After the Gating Transformation is performed and a new collapsed block is created, that new block will hold the list of Gate Groups to represent all the Gating information.

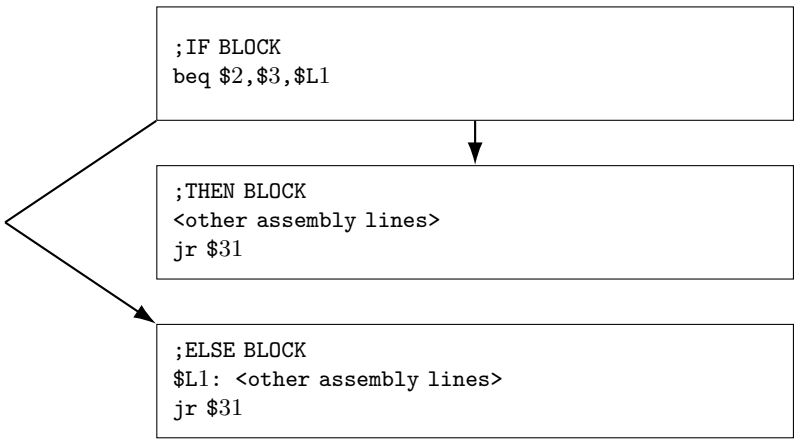

Figure 6.3: IF-THEN-ELSE-AFTER structure without AFTER BLOCK

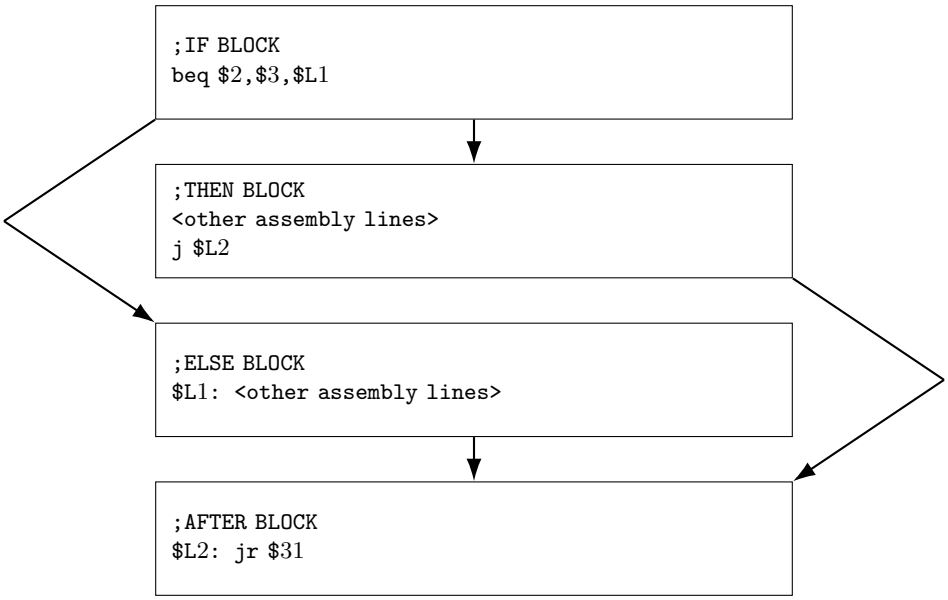

Figure 6.4: IF-THEN-ELSE-AFTER structure after inserting new AFTER BLOCK

6.2.1 Ensure Correct Gating Structure

A branching region can only be gated if the AFTER BLOCK has two predecessors. In Fig. 6.1, the AFTER BLOCK actually has three predecessors, so we have to slightly modify it to make it conform to the rule mentioned above. In order to fix that three predecessors problem, in Fig. 6.2, an empty temporary AFTER_2 BLOCK is inserted between THEN_2 BLOCK and AFTER BLOCK to make the innermost branching region consistent with the IF-THEN-AFTER structure.

There can also be branching regions that exits at the THEN BLOCK and the ELSE block as shown in Fig. 6.3. Hence, the AFTER BLOCK does not exist in those cases. We can easily handle these cases by inserting a new AFTER BLOCK and move the `jr $31` to that block as shown in Fig. 6.4.

6.2.2 Construct Gate Group for Register-Writing Instructions

An RGate Group can be represented by the following pseudocode:

```
struct RGateGroup {
    string gated_reg; \\ gated regular register
    struct PredExpr pred_exprs[]; \\ list of predicate expressions
    struct ReachDef reach_defs[]; \\ list of corresponding reaching definitions
}
```

The Gating algorithm will first search through the list of registers in the `IN` of the AFTER BLOCK to see if there is any register that has more than one `reaching_definitions`. If that register exists, the `reaching_definitions` of that register will be filtered to only include the `reaching_definitions` that appears within the branching region. Each possible path of branching execution will be represented by a predicate expression. In a single-level gating transformation, there can only be two predicate expressions, but in a nested multi-level gating transformation there can be more than two predicate expressions. For example, the gating region in Fig. 4.3 has two RGate Groups:

```
RGate Group 1:
- gated register: $4
- definitions = {(li $4,$5), (addi $4,$6,$1)}
- predicate expressions = {($2 == $3), ($2 != $3)}

RGate Group 2:
- gated register: $5
- definitions = {(li $5,$6), (addi $5,$2,$1)}
- predicate expressions = {($2 == $3), ($2 != $3)}
```

6.2.3 Construct Gate Group for Memory-Writing Instructions

An MGate Group can be represented by the following pseudocode:

```
struct MGateGroup {
    string gated_addr_reg; \\ gated register holds the memmory address
    struct PredExpr pred_exprs[]; \\ list of predicate expressions
    struct AssemblyLine store_instructions[];
        \\ list of corresponding store instructions
}
```

The Gating algorithm will search through the assembly lines in the THEN BLOCK and ELSE BLOCK to find if there are any store instructions that need to be guarded. For example, the gating region in Fig. 4.4 has one MGate Group:

```
MGate Group 1:
- gated register: $6
- memory instructions = {(sw $2,4($6)), (sw $3,($6))}
- predicate expressions = {($2 == $3), ($2 != $3)}
```

Note that a store instruction in the IF BLOCK or the AFTER BLOCK does not need to be guarded because it always gets executed when the branch region is entered. There are cases where the THEN BLOCK has a store instruction but the ELSE BLOCK does not have any store instructions. In those cases, the predicate expressions list of the corresponding MGate Group will only have one predicate expression representing the THEN path. Similarly, there are cases where there is a store instruction in the ELSE BLOCK, but none is found in the THEN BLOCK. The MGate Group can have fewer than two predicate expressions.

6.3 Resolve Gate Indexes

Two Gate Groups will be considered PredSet Equivalent if the sets of predicate expressions of those two Gate Groups are equal to each other. The Gating Algorithm will find Gate Groups that are PredSet Equivalent and put them into a PredSetEquivalent Collection. A PredSetEquivalent Collection can be represented by the following pseudocode:

```
struct PredSetEquivalentCollection {
    struct GateGroup gate_groups[]; \\ list of PredSet Equivalent groups
    struct PredExpr pred_exprs[];
                \\ list of common predicate expression of those groups
    int pred_exprs_count; \\ number of predicate expressions
}
```

Note that `gate_groups` can hold both RGate Groups and MGate Groups because they are both guarded by predicate expressions. As mentioned above, an MGate Group can have fewer than two predicate expressions. Hence, if the set of predicate expressions of an MGate Group is the subset of the set of predicate expressions of another Gate Group, then those two Groups are also considered PredSet Equivalent. After forming all the PredSetEquivalent Collections, the vector index of each Predicate Expression will be calculated by the following pseudocode:

```
struct PredSetEquivalentCollection all_collections[]; \\ list of all collections
idx_count = 0;
for (int i = 0; i < collections_count; i ++) {
    struct PredSetEquivalentCollection current_collection = all_collections[i];
    for (int j = 0; j < current_collection.pred_exprs_count; j ++) {
        current_collection.pred_exprs[j].vector_index = idx_count;
        idx_count ++;
    }
}
```

The Predicate Expressions of the Gate Groups in the same Collection will have the same vector index as the index of Predicate Expression in that Collection. In Fig. 6.5, there are three Gate Groups:

```
Gate Group 1:
- gated register: $4
- definitions = {(addi $4[0],$6,$1), (li $4[1],6)}
- predicate expressions = {($7 == $8), ($7 != $8)}

Gate Group 2:
- gated register: $5
- definitions = {(li $5,5), (addi $5,$4,3)}
- predicate expressions = {($2 != $3), ($2 == $3)}

Gate Group 3:
- gated register: $6
- definitions = {(li $6,6), (addi   $6,$5,8)}
- predicate expressions = {($2 != $3), ($2 == $3)}
```

The first Collection contains only Gate Group 1 in the ELSE BLOCK which is guarded by the predicate set {`($7 == $8), ($7 != $8)`}. The second Collection contains Gate Group 2 and Gate Group 3 which have the same set of predicate expressions {`($2 != $3), ($2 == $3)`}. In total, there are 4 predicate expressions that will be indexed as follows: `0: ($7 == $8), 1: ($7 != $8), 2: ($2 != $3), 3: ($2 == $3)`. The gate instruction of each Gate Group will be constructed based on the indices of the predicate expressions. For example, the Gate Group of

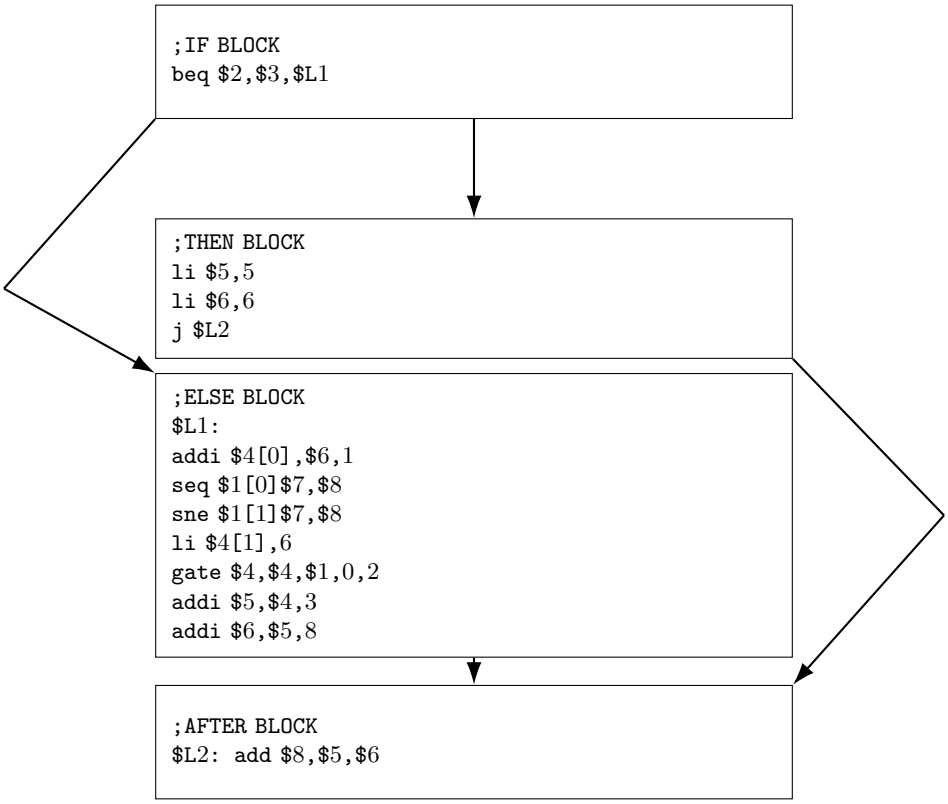

Figure 6.5: Nested Gating Region with Two PredSetEquivalent Collections

register $5 has two predicate expressions with indices 2, 3, respectively, and will have the following gate instruction: `gate $5,$5,$1,2,2`. The definition in each Gate Group will be numbered based on the gating index of the predicate expression that guards it. For example, the `li $6,6` instruction is guarded by the `($2 != $3)` predicate expression which has the gating index 2, so that instruction will be transformed to `li $6[2],6` instruction. Fig. 6.6 shows how the gating indices are used to construct the final gated block.

6.4 Rename Intermediary Registers

In Fig. 4.3, before the gating transformation, register $6 only lives within the fall-through path of the branch. These registers are called Intermediary Registers since the register value will not be used by a `gate` instruction. Because poison bits can only be propagated via vector instance registers, all the Intermediary Registers need to be renamed to be a vector instance register. In Fig. 4.3, after

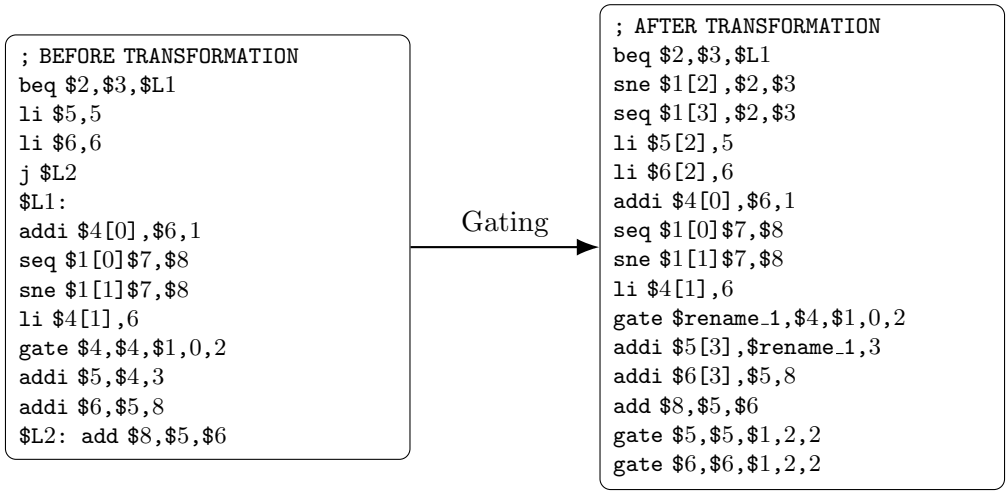

Figure 6.6: Gating Transformation with Two PredSetEquivalent Collections

the gating transformation, placeholders such as `$rename1` signifies that those registers need to be renamed. We chose set `A = {$1[31],$2[31],$3[31],...}` to be the set of instance registers to be used for Intermediary Registers renaming. This is just an arbitrary choice. Set `A` can be modified to be any set of instance registers.

6.5 Iterative Application

6.5.1 Local Reaching Definition Recalculation

Normally, the `reaching_definitions` is iteratively calculated throughout the whole function until no more changes are found. The `reaching_definitions` are propagated through the `predecessor` and `successor` links following Depth-First-Search pattern. When a branching region gets collapsed via gating to form a new basic block, the old `reaching_definitions` are no longer valid. We can recalculate the `reaching_definitions` for the whole function, but that means a global recalculation is performed each time a Gating transformation is applied. Hence, that approach significantly increases the compilation time. Instead, we perform a local `reaching_definitions` recalculation. We first calculate set `A` containing all the invalid `reaching_definitions` after gating. We then perform a Depth-First-Search (DFS) via `successor` links starting from the

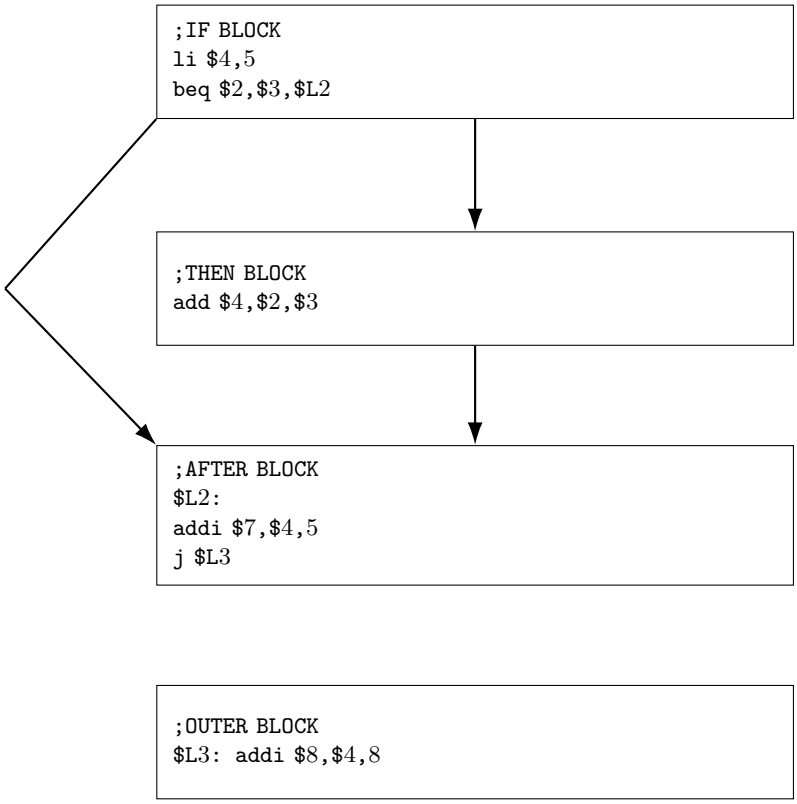

Figure 6.7: Reaching Definitions from Gating Region

newly formed block after gating and then proceeding to find every basic block that uses the definitions inside set `A`. We recalculate the `reaching_definitions` of those blocks, mark those blocks as `changed`, and proceed with the Depth-First-Search until no changes are found. For example, in Fig. 6.7, set `A` will contain all the instructions in the IF BLOCK, THEN BLOCK, and AFTER BLOCK. The `addi $8,$4,8` instruction in the OUTER BLOCK uses register $4 which has two definitions that can reach the instruction: `li $4,5` and `add $4,$2,$3`. Those two definitions are in the set `A`, so after gating is successfully performed the reaching definitions in OUTER BLOCK will need to be recalculated and marked as `changed`. The DFS determines which block needs to recalculate reaching definitions.

6.5.2 Gate Chaining

In Fig. 6.8, when constructing a new Gate Group for register $4, we can encounter a definition that is set by a gate instruction. That gate instruction is linked to another Gate Group in the

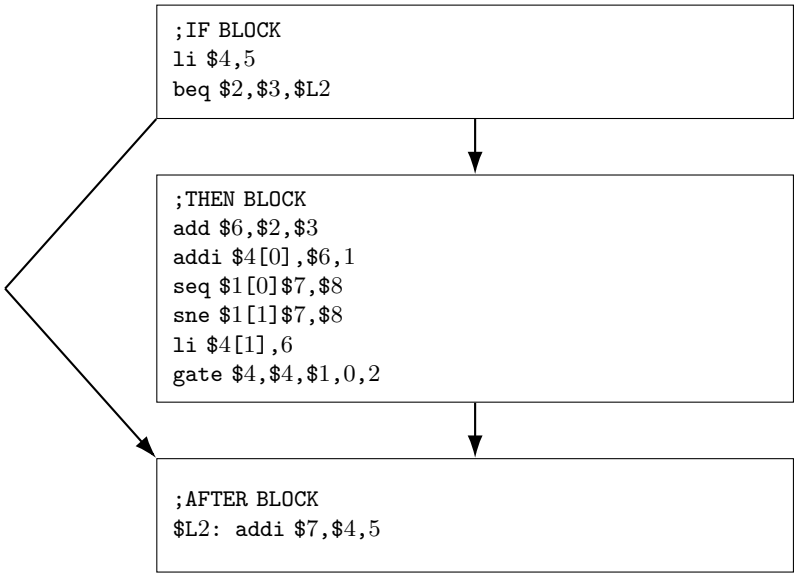

Figure 6.8: Nested Branching Region with Gate Chaining Opportunity

THEN BLOCK:

```
Gate Group 1:
- gated register: $4
- definitions = {(addi $4[0],$6,$1), (li $4[1],6)}
- predicate expressions = {($7 == $8), ($7 != $8)}
```

All the definitions of register $4 in that old Gate Group will be transferred to the new Gate Group that we are constructing. The predicate expression of that old Gate Group will be modified to be the logical `and` of the outer predicate and the inner predicate. Those modified predicate expressions will also be transferred to our new Gate Group of register $4. The final Gate Group of register $4 will have three reaching definitions:

```
Gate Group 1:
- gated register: $4
- definitions = {(addi $4[0],$6,$1), (li $4[1],6), (li $4,5)}
- predicate expressions = {($2 != $3) && ($7 == $8), ($2 != $3) && ($7 != $8),
                           ($2 == $3)}
```

The logical `and` expression will be calculated by the `cand` instruction in the VIS ISA. Fig. 6.9 shows how the gated block is constructed when gate chaining is applied.

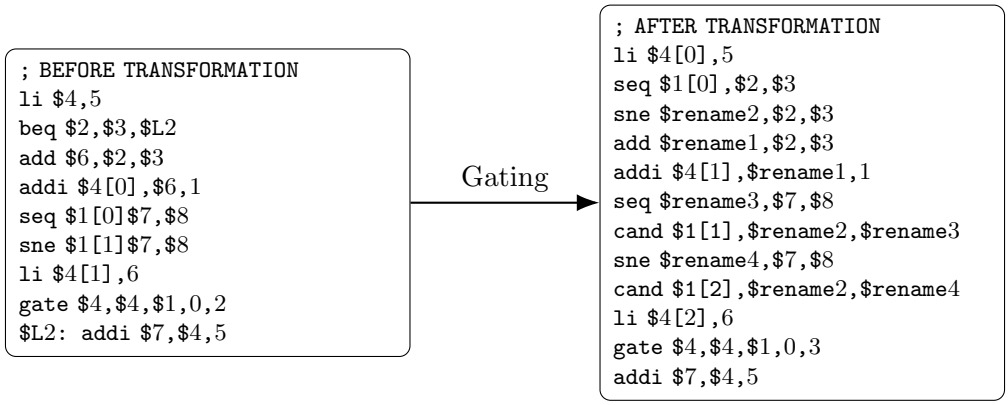

Figure 6.9: Gating Transformation with Gate Chaining

6.6 Block Scheduling

Jacob (Judas) Smith at Florida State University has implemented an algorithm in asopt to schedule instructions within a basic block [5]. To reduce stalls due to true dependencies (read-after-write), the algorithm builds a weighted Direct Acyclic Graph (DAG) of dependency between instructions, and the weight of each connection is the latency of the instruction. From this graph, we will be able to sort chains of dependencies by its weighted length. The instructions at the start of the longer chains of dependency will be prioritized to be put at the beginning of the basic block. Instructions on different chains of dependencies will be interleaved to overlap execution and hide the latency of each other.

CHAPTER 7

RESULTS

Benchmark	Description
099.go	AI program playing the game of Go.
124.m88ksim	Simulator for the Motorola 88100 processor.
126.gcc	GNU C compiler compiling preprocessed C source files.
129.compress	Data compression using Lempel–Ziv algorithm.
130.li	Lisp interpreter running symbolic computation tasks.
132.ijpeg	Image compression and decompression using JPEG algorithm.
134.perl	Perl interpreter executing shell scripts.
147.vortex	Build and manipulate databases

Table 7.1: SPEC95 Integer Benchmarks

We tested the Gating Transformation on the SPEC95 integer benchmark suite (Table. 7.1) using the ref input. We conducted experiments on two CPU configurations, where the first is less aggressive and the second is more aggressive. We use the 64-bit MIPS ISA (Section. 3.1) as the baseline to compare the performance of the gating code.

7.1 Static Statistics

The average percentage of static branches that got eliminated by Gating is 11% (Fig. 7.1). The two main factors that make the remaining branches inconvertible are wrong structure (38%) and the use of function calling instructions in the branching region (34%). Regarding the branches not removed because of wrong structure, we are working on a new version of Gating to eliminate branches that are not IF-THEN-AFTER and IF-THEN-ELSE-AFTER structures. The branches that have function calling instruction in the branching region are difficult to eliminate because VIS vector registers are scratch registers and the function calling instructions would overwrite scratch registers. On average, the proportion of loop branches is 14.6%, and we cannot eliminate these branches unless we perform loop unrolling.

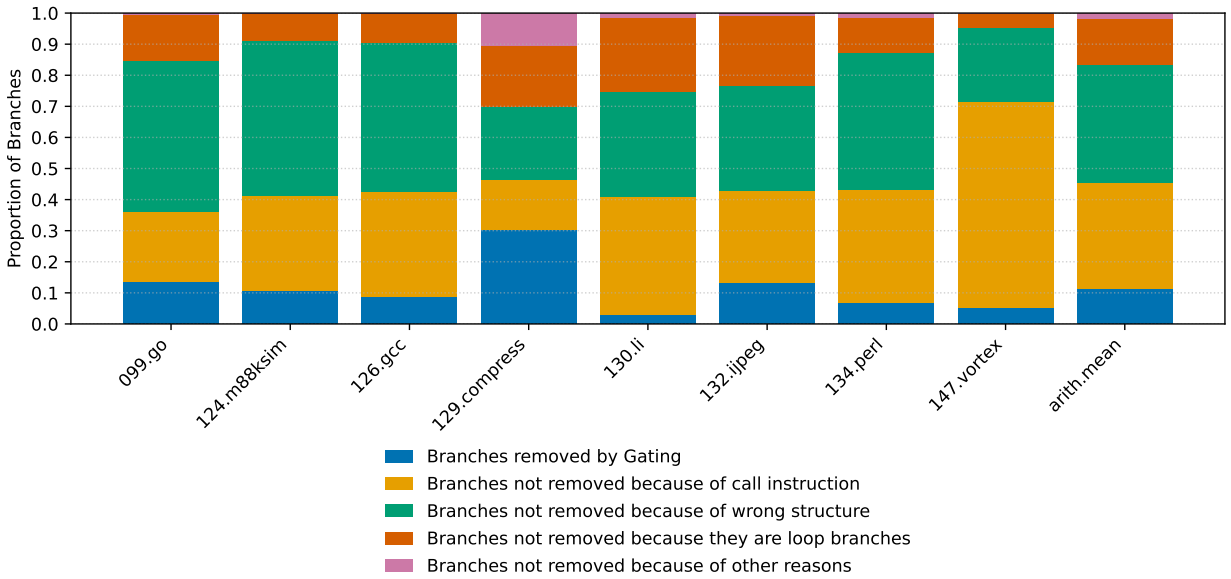

Figure 7.1: SPEC95 Branches Conversion Categories

7.2 Run-time Statistics

On average the percentage reduction in cycles count is 6.0% on CPU Config 1 and 6.3% on CPU Config 2 (Fig. 7.2). The best performing benchmark is 129.compress with 20.2% decrease in cycle count on CPU Config 1 and 21.2% on CPU Config 2. The CPU specifications are explained in Table 7.2. Compared to 64-bit MIPS performance, the number of branch mispredictions also decreases on average around 48.5% on both CPU Configs (Fig. 7.3). Only 130.li benchmark has more branch mispredictions on the gated code which contributes to the increase in cycles count on this benchmark. There can be many factors that cause the increase in branch mispredictions in 130.li, but we think the leading factor is because this benchmark has many branching regions that do not follow IF-THEN-AFTER or IF-THEN-ELSE-AFTER structures. 134.perl benchmark has significantly fewer branch mispredictions compared to 64-bit MIPS, but the cycles count of this benchmark does not improve much. We think that the decrease in branch mispredictions in 134.perl is offset by the sharp increase in speculative load rollbacks (Fig. 7.4), so overall there is no performance gain. The increase in cycles count on 126.gcc can be partly explained by the significant rise in speculative load rollbacks. A load instruction is speculatively executed based on a prediction

Acronym	Explanation
decode_width	Max number of instructions decoded per cycle.
issue_width	Max number of instructions issued to Execution Units per cycle.
exu_int_sc_width	Parallel single-cycle integer ALU slots (eg., add/or).
exu_int_mc_width	Parallel multi-cycle integer ALU slots (e.g., mul/div).
load_ports	Number of loads that can start per cycle.
store_ports	Number of stores that can start per cycle.
store_queue_ports	Number of stores the Store Queue can accept per cycle.
retire_width	Max number of instructions committed per cycle.
reorder_buf_entries	Max number of in-flight instructions the core can keep in progress.

Table 7.2: CPU Specification Terms

that the preceding store instruction does not have the same address as that load instruction. When the prediction is wrong the processor needs to perform a rollback. Most of the benchmarks have more speculative rollbacks because the number of load instructions greatly increase after gating. We are trying to create a memory dependency predictor that is more suitable to gated code. Some other micro-architecture features such as a bigger memory set in the memory predictor are also being considered and tested.

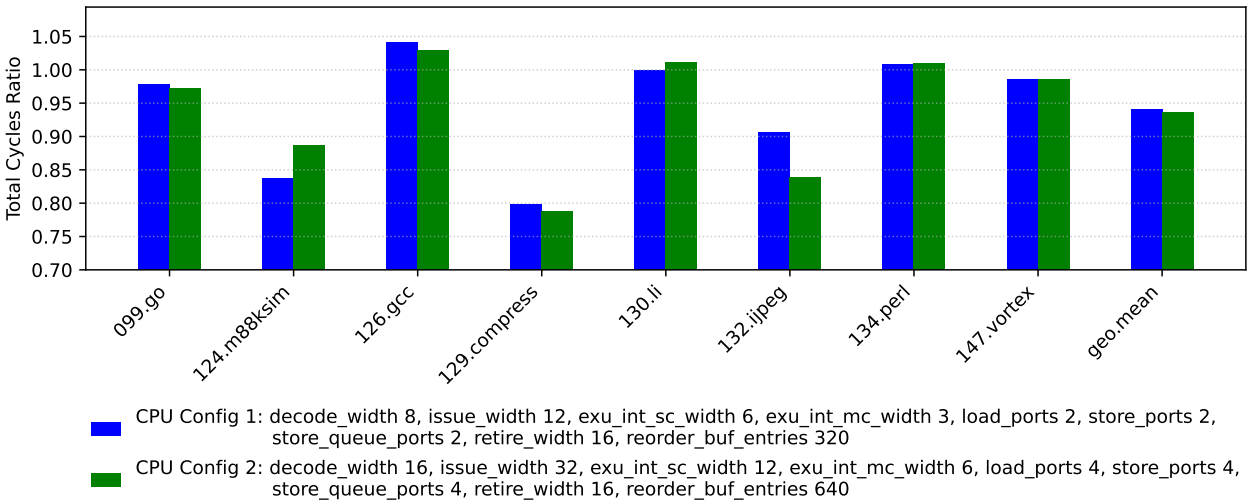

Figure 7.2: SPEC95 Total Cycles compared to 64-bit MIPS

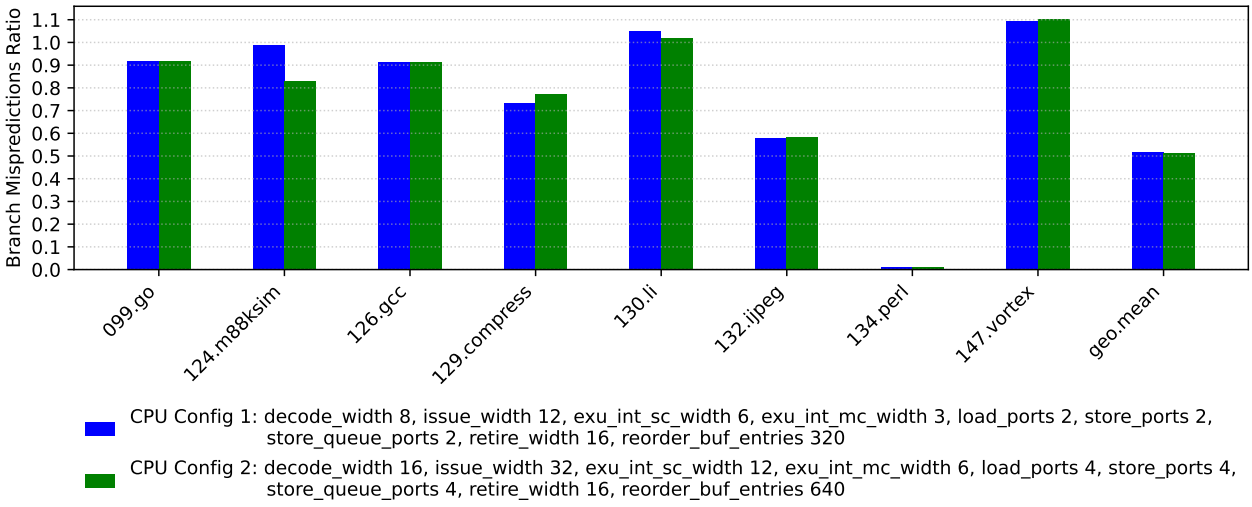

Figure 7.3: SPEC95 Total Branch Mispredictions compared to 64-bit MIPS

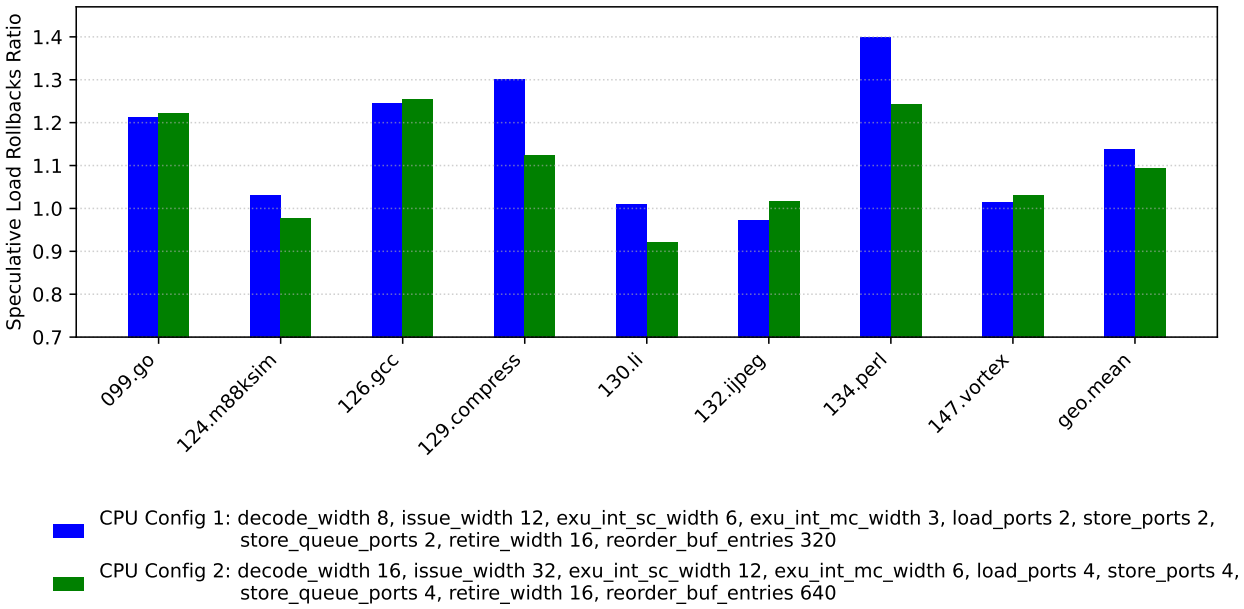

Figure 7.4: SPEC95 Speculative Load Rollbacks compared to 64-bit MIPS

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

We developed a selected code region gating conversion algorithm to generate code that can be effectively executed on a VIS architecture. The simulation results on the ADL simulator have proved that this transformation successfully improves cycles count in the SPEC95 integer benchmarks. The Gating method provides more freedom in regards to scheduling when compared to the predication method. The instructions in the branching region can be scheduled before the predicate setting instructions. Compared to the predication method, the Gating optimization also provides a better way to handle imbalanced branches. When the original control flow takes the shorter path, the `gate` instruction does not get delayed because of the instructions on the longer non-taken path. In the future, we will implement loop unrolling to eliminate some of the loop branches. We are also building a new algorithm that will be able to handle more types of branching structures. We also plan to test with SPEC06 and SPEC17 integer benchmarks. In addition, we will apply the Gating transformation to the library code.

BIBLIOGRAPHY

[1] F. E. Allen and J. Cocke. "A program data flow analysis procedure." In: *Commun. ACM* 19.3 (Mar. 1976), p. 137. ISSN: 0001-0782. DOI: 10.1145/360018.360025. URL: https://doi.org/10.1145/360018.360025.

[2] Shuhan Ding, John Earnest, and Soner Önder. "Single Assignment Compiler, Single Assignment Architecture: Future Gated Single Assignment Form*; Static Single Assignment with Congruence Classes." In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14. Orlando, FL, USA: Association for Computing Machinery, 2018, pp. 196–207. ISBN: 9781450326704. DOI: 10.1145/2544137.2544158. URL: https://doi.org/10.1145/2544137.2544158.

[3] Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." In: *IEEE Transactions on Computers* C-30.7 (1981), pp. 478–490. DOI: 10.1109/TC.1981.1675827.

[4] Wen-Mei W. Hwu et al. "The superblock: an effective technique for VLIW and superscalar compilation." In: *J. Supercomput.* 7.1–2 (May 1993), pp. 229–248. ISSN: 0920-8542. DOI: 10.1007/BF01205185. URL: https://doi.org/10.1007/BF01205185.

[5] David Landskov et al. "Local Microcode Compaction Techniques." In: *ACM Comput. Surv.* 12.3 (Sept. 1980), pp. 261–294. ISSN: 0360-0300. DOI: 10.1145/356819.356822. URL: https://doi.org/10.1145/356819.356822.

[6] A. Mortensen. "Isolating Errors for an Assembly Optimizer." Master's thesis. Florida State University, 2022. URL: https://purl.lib.fsu.edu/diginole/2022_Summer_Mortensen_fsu_0071N_17367 (visited on 10/02/2025).

[7] Abigail Mortensen et al. "Facilitating the Bootstrapping of a New ISA." In: *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2023. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 2–12. ISBN: 9798400701740. DOI: 10.1145/3589610.3596282. URL: https://doi.org/10.1145/3589610.3596282.

[8] S. Onder and R. Gupta. "Automatic generation of microarchitecture simulators." In: *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*. 1998, pp. 80–89. DOI: 10.1109/ICCL.1998.674159.

[9] Soner Onder. *An introduction to Flexible Architecture Simulation Tool (FAST) and Architecture Description Language ADL*. Tech. rep. TR 05-01. Version 1.0. Technical Report. Houghton, MI, USA: Department of Computing Science, Michigan Technological University, 2005.

[10] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages." In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. PLDI '90. White Plains, New York, USA: Association for Computing Machinery, 1990, pp. 257–271. ISBN: 0897913647. DOI: 10.1145/93542.93578. URL: https://doi.org/10.1145/93542.93578.

[11] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. *Gated Single Assignment Form: Dataflow Interpretation for Imperative Languages*. Tech. rep. LA-UR-89-3654. Los Alamos, NM: Los Alamos National Laboratory, Oct. 1989, p. 23.

[12] Gary Scott Tyson. "The effects of predicated execution on branch prediction." In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 196–206. ISBN: 0897917073. DOI: 10.1145/192724.192753. URL: https://doi.org/10.1145/192724.192753.

BIOGRAPHICAL SKETCH

Quan Pham completed his Bachelor of Science in Computer Science at Florida State University in 2023. In the senior year of his undergraduate study, he joined Dr. Whalley's research lab in Compilers and Computer Architecture. He worked in this research lab throughout his time pursuing Master's degree in Computer Science. During two and a half years working as a research assistant, he has built testing tools and the Gating optimization technique which is part of the NSF project Vectorized Instruction Space (VIS).