2017

Dependency Collapsing in Instruction-Level Parallel Architectures

Victor J. Brunell

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DEPENDENCY COLLAPSING IN INSTRUCTION-LEVEL PARALLEL ARCHITECTURES

By

VICTOR BRUNELL

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

2017

Victor Brunell defended this thesis on July 21, 2017.
The members of the supervisory committee were:

David Whalley
Professor Directing Thesis

Gary Tyson
Committee Member

Xin Yuan
Committee Member

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

ACKNOWLEDGMENTS

I would like to thank Dr. David Whalley for all his support and for always having an open door. I couldn't have done this without your advice and encouragement. Also, I'd like to thank Ryan Baird and Michael Stokes from the Compiler Group for all their assistance. I really appreciate having you both there for all the debugging, compiler conversations, and good times. Finally, I'd like to thank Soner Onder and Gorkem Asilioglu from Michigan Technological University for all your assistance in completing this work.

I would also like to thank the National Science Foundation for supporting this research. This research was supported in part by NSF grants DRL-1640039, DGE-1565215, CCF-1533846, IIA-1358147, DUE-1259462, and DUE-1241525.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Processors that employ instruction fusion can improve performance and energy usage beyond traditional processors by collapsing and simultaneously executing dependent instruction chains on the critical path. This paper describes compiler mechanisms that can facilitate and guide instruction fusion in processors built to execute fused instructions. The compiler support discussed in this paper includes compiler annotations to guide fusion, exploring multiple new fusion configurations, and developing scheduling algorithms that effectively select and order fusible instructions. The benefits of providing compiler support for dependent instruction fusion include statically detecting fusible instruction chains without the need for hardware dynamic detection support and improved performance by increasing available parallelism.

CHAPTER 1

INTRODUCTION

Exploiting instruction level parallelism has been critical for high performance general computing over the past few decades and instruction-level parallel processing now benefits a wide variety of applications. However, the achievable speed-up enabled by any kind of parallel processing is limited by the sequential component of the computation being performed according to Amdahl's law. For ILP, the sequential component is the chains of dependent instructions on a program's critical path. ILP can be increased and the critical path reduced by fusing dependent instructions and executing them simultaneously via cascaded ALUs in order to collapse the dependencies. *Available parallelism* in the program is increased by exploiting dependent parallelism, as opposed to the traditional approach of only exploiting independent parallelism, exploiting dependent parallelism is achieved by performing dependency collapsing. In this paper, *dependent parallelism* is defined to be the amount of parallelism that is available among dependent instructions. By focusing on reducing the sequential portion of Amdahl's law at the instruction level, all programs can potentially achieve speed-up, irrespective of the application domain.

Architectures that can schedule and simultaneously execute dependent and independent instructions via the micro-architecture or the compiler alone, or by a combination of the compiler and the micro-architecture, we refer to as *Dependency Collapsing Architectures* (DeCA). A DeCA can be *Dynamically Scheduled* (DS-DeCA) or *Statically Scheduled* (SS-DeCA). The research conducted in this work is focused on establishing compiler techniques and algorithms that enable effective dependency collapsing in DeCAs. One example of a DS-DeCA is LaZy Superscalar developed by Michigan Technological University (MTU), which we will discuss in detail in Chapter 2.

Instructions amenable to dependency collapsing can be simultaneously executed via cascaded ALUs which is a process we refer to as *instruction fusion*. A cascaded ALU refers to an ALU configuration where the output of one ALU is directly used as input to a subsequent ALU. For instance, consider cascaded integer additions. By taking advantage of the fact that the least significant bits of an addition finish earlier than the most significant bits, the second addition can start before the first addition completes and cascaded ALUs can potentially execute multiple instructions in the same clock cycle.

Performing instruction fusion provides many benefits not seen in traditional processors. By aggressively harvesting dependent ILP as opposed to only exploiting independent ILP, available parallelism is increased and ILP for the program on the whole is augmented. Performance is benefited by cycle time reductions due to instruction fusion reducing the critical path and by delaying the execution of dependent fused instructions so they can be executed in the same clock cycle. Scheduling for fusion offers benefits over traditional instruction scheduling that focuses on grouping independent instructions close together and dependent instructions farther apart. Scheduling for dependent parallelism allows us to aggressively schedule dependent instructions based on the critical path through the program, and removes some of the constraints imposed by processor designs that focus only on independent parallelism. Finally, one of the major benefits of instruction fusion in RISC processors is the ability for fusion to produce CISC-like instruction packets without the need for a CISC ISA. In essence, fusion allows for synthesizing CISC instructions from RISC instructions.

Currently, the most widely used form of fusion allows for three inputs and two outputs in order to fuse two dependent instructions. We refer to this form of fusion as *conventional fusion*. We will also explore new fusion configurations and types that allow for four inputs and three outputs, in order to fuse three instructions in the same cycle instead of two. We will discuss these new fusion configurations in detail in Chapter 3.

The focus of this research is to develop the compiler techniques and algorithms that enable effective and guided dependency collapsing in DeCAs and to explore the availability and effectiveness of new fusion configurations that allow for three instructions to be fused and simultaneously executed. Compiler support for instruction fusion is provided by the following compiler mechanisms:

1. Guiding dependent instruction fusion by:
   (a) Identifying the critical path.
   (b) Selecting fusion configurations to assign each set of fusible instructions to reduce the height of the critical path.
   (c) Providing instruction annotations that indicate when dependent instructions should be fused.
2. Instruction scheduling by:
   (a) Scheduling based on critical path information.
   (b) Reducing the height of the critical path via dependency collapsing.
   (c) Scheduling dependent fused instructions closer together to facilitate exploiting fusion.

CHAPTER 2

PRELIMINARY WORK

2.1 LaZy Superscalar

Preliminary instruction fusion performance results were obtained using a DS-DeCA developed by MTU called LaZy Superscalar [2]. LaZy Superscalar is a processor architecture that delays the execution of fetched instructions until their results are demanded by subsequently fetched instructions. It is the first demand-driven dynamically scheduled architecture that delays the execution of fetched instructions until they are demanded, and provides the capability to identify which producer instructions should be conventionally fused with consumer instructions that demand their results. This allows for collapsing pairs of instructions that represent both near and distant dependencies using a three input, two output single-cycle ALU configuration.

LaZy Superscalar focuses on an architecture design that attempts to combine as many dependent instructions as possible via conventional fusion to improve available parallelism, as opposed to aggressively executing only independent instructions and scheduling dependent instructions as far apart as possible. LaZy Superscalar also provides the benefit of dead code elimination via its use of demand driven execution. Since an instruction that is never demanded is not executed (i.e., the producer lacks a subsequent consumer), the dead instruction will in effect be dynamically eliminated. However, due to compiler optimizations the amount of dynamically detected dead code is generally quite low. Results published in [2] indicate that less than 1% of dynamic instructions in the Spec2006 integer benchmarks were dead instructions.

To summarize, in LaZy Superscalar, a producer instruction is handled in one of the following three ways:

1. It is eventually fused to a consumer instruction.
2. It is executed without being fused due to a demand from a non-fusible instruction.
3. It is identified as dead code and is simply killed.

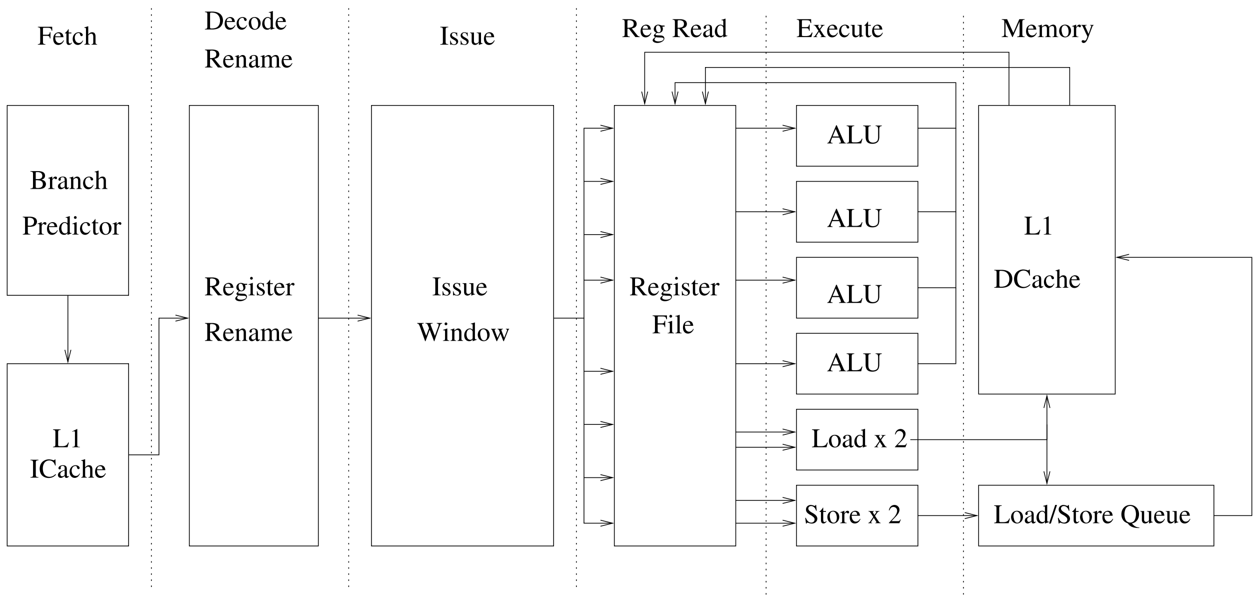


Figure 2.1: Conventional Superscalar Pipeline

2.1.1 The LaZy Superscalar Pipeline

The LaZy Superscalar Pipeline structure is identical to conventional superscalar processors at the front and back end of the pipeline. A conventional superscalar pipeline is shown in Figure 2.1. The LaZy pipeline differs from the conventional pipeline in a few key ways. First, the LaZy pipeline includes a LaZy dependence matrix to provide instruction scheduling by combining both data and demand signalling and allowing the matrix to drive the instruction retire process. *Demand signals* refer to signals which are received by producer instructions from their consumers, as opposed to conventional dataflow style wake-up/select mechanisms, where signals are received by consumers from their producers. Second, the renaming mechanism in the pipeline has been updated to rename all instructions. Third, due to unified dependency checking, the pipeline does not incorporate load queues. It does contain a store queue to hold speculative values from store instructions until they are ready to retire. Finally, there are separate commit and retire phases. The commit phase commits instructions in program order regardless of their completion status, but instructions are retired out of order when they are completed or squashed. Figure 2.2 shows the LaZy Superscalar pipeline organization.

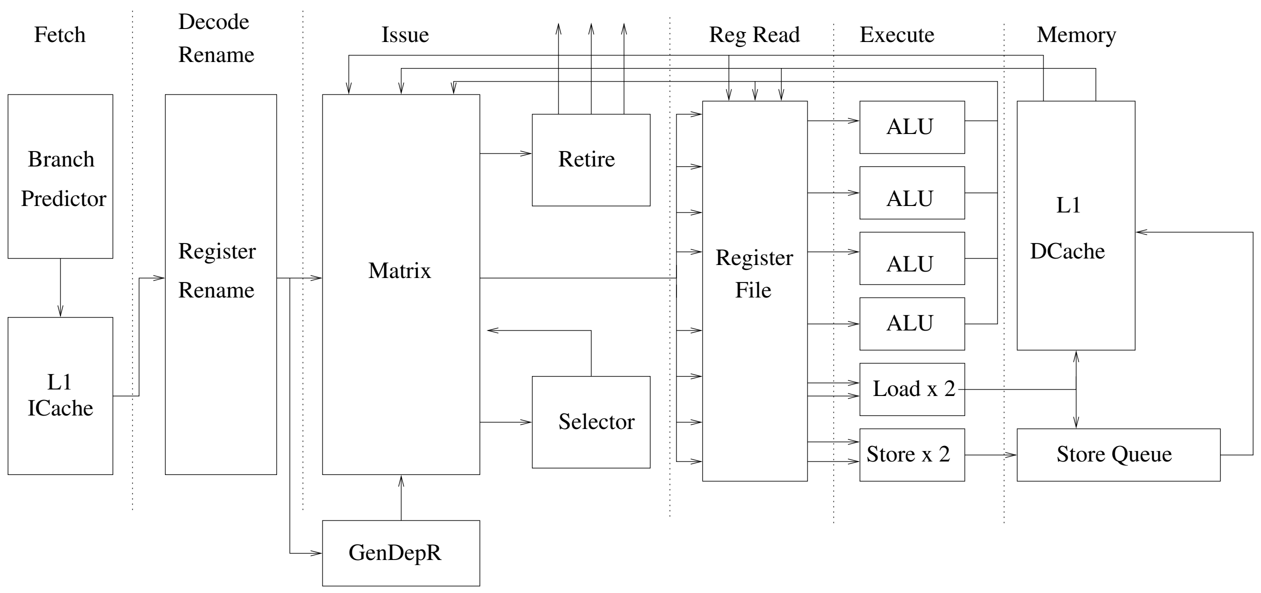


Figure 2.2: LaZy Superscalar Pipeline

2.1.2 LaZy Superscalar - Preliminary Results

LaZy Superscalar and the baseline superscalar processors were simulated by automatically synthesizing them from descriptions written in the ADL processor description language [4]. Figure 2.3 shows the results of executing the Spec2006 integer benchmarks on both architectures and comparing their performance. Also, fusion coverage statistics are given, which show the ratio of instructions that were fused in each benchmark.

As the results of Figure 2.3 indicate, there are cases when fusion is widely available, but commensurate increases in performance are not achieved, as is the case in the *mcf* benchmark. Two possible reasons for this occurring in LaZy Superscalar are:

1. Instructions may be unnecessarily delayed if a fusible instruction never demands them, but a non-fusible instruction eventually does.

2. The choice to fuse instructions that might not be on the critical path, and hence the fusion will not produce a performance benefit.

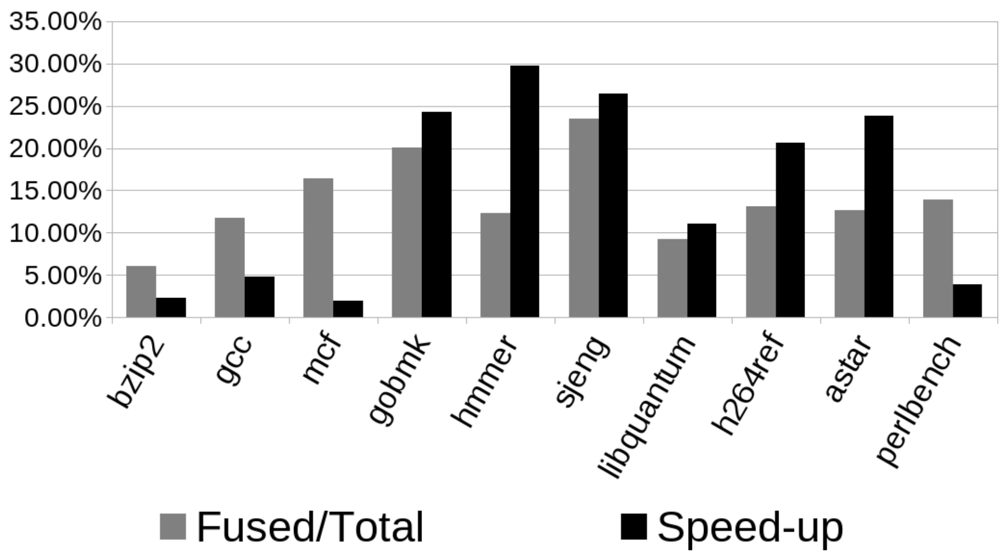


Figure 2.3: Fusion Ratio and Performance Increase in LaZy

2.1.3 LaZy Superscalar - Fusion Example

LaZy Superscalar uses demand-driven execution to schedule and fuse instructions via demand signals sent from consumer instructions to their producers. Figure 2.4 shows how a code sequence would be scheduled and executed in LaZy Superscalar assuming a 3-wide fetch unit. Since LaZy conventionally fuses consumer instructions to their producers, we see two conventional fusions taking place given the data-demand flow graph shown in the figure. The critical path is identified to be i2 → i4 → i7 → i8. The instruction pairs i2 → i4 and i7 → i8 are conventionally fused, leading to a collapse of the height of the critical path from four instructions to three. Each cycle of the code sequence proceeds as follows. In cycle 0, i1, i2, and i3 are fetched and buffered since no consumer instruction has demanded them yet. In cycle 1, i4, i5, and i6 are fetched, a demand signal is sent from i4 to i2 and the pair are fused and start executing, while i5 and i6 sleep because they lack a demand signal. In cycle 2, i7, i8, and i9 are fetched and demand signals are sent from i7 and i8 to i4, i5, and i7, which causes the i7 → i8 pair to fuse and start executing, leading to the final schedule shown at the bottom of the diagram with a critical path height of three instructions.

	Instructions
i_1	addu \$2, \$4, \$6
i_2	addu \$5, \$12, \$21
i_3	addu \$8, \$13, \$22
i_4	addu \$14, \$5, \$23
i_5	addu \$9, \$5, \$24
i_6	addu \$10, \$8, \$5
i_7	addu \$3, \$14, \$26
i_8	addu \$7, \$3, \$9
i_9	addu \$11, \$4, \$6

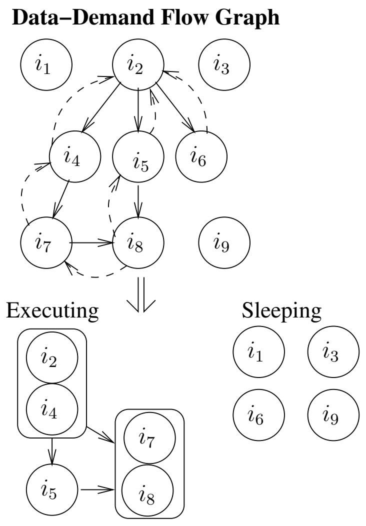


Figure 2.4: LaZy Superscalar Fusion Example

2.2 Extending LaZy Superscalar

The preliminary results produced by work on LaZy Superscalar leave multiple questions open that we seek to address in this work. First, how can we identify which producer instructions to delay and which not to? Second, which dependent instructions should be collapsed and which fusion configurations provide the most benefit for collapsing instructions? Third, how should dependent fused instructions be scheduled to increase performance? Lastly, what compiler support can be given to identify and schedule collapsible dependent instructions?

In order to explore these questions and address the performance issues discussed in the previous section, we extend LaZy Superscalar by providing compiler support for fusion annotation and fused instruction scheduling. First, we provide instruction annotations that can be used to indicate when an instruction should be delayed and when it should not be delayed. Second, we identify the critical path and mark instructions for fusion based on the reduction of the critical path, which guides fusion away from unnecessary delays and the execution of fused instructions that does not benefit performance. Third, we aggressively schedule dependent instructions and increase the proximity of fused instructions so groups of fused instructions can execute earlier.

In addition, we also extend the available types of fusion from only conventional fusion, to include new fusion types that correspond to four input three output ALU configurations. These new fusion types are used to provide greater coverage for fusion opportunities and to allow the collapse of the critical path to occur more rapidly. We will discuss these new configurations in detail in the next chapter.

By exploiting alternative forms of cascading and implementing and evaluating a collaborative approach between the compiler and the microarchitecture, these extensions to LaZy Superscalar target better exploitation of critical path fusing and improved scheduling and optimization of the instruction stream. In the next chapter, we will discuss instruction fusion in detail and describe the compiler support implemented to support DS-DeCAs like LaZy Superscalar.

CHAPTER 3

INSTRUCTION FUSION

The goal of instruction fusion is to reduce the number of cycles required to execute the critical path through the program to improve performance. In this chapter we will discuss the extensions made to traditional conventional instruction fusion and the compiler mechanisms implemented to support instruction fusion in DeCAs. For the purposes of this work, we will consider only single-cycle integer ALU operations as candidates for fusion. We define a *fusible instruction* to be a non-branch ALU instruction with no more than two operands and a single destination. Given this definition, we disqualify branches and multi-destination MIPS instructions, such as multiplies and divides, as candidates for fusion.

3.1 Conventional Fusion

Conventional fusion refers to the fusing of a pair of dependent instructions, i.e., a producer instruction with its consumer instruction. In order to achieve this, we use a conventional cascading of two ALUs. In this configuration, the output of the first ALU is used as an input to the second ALU in the cascade. There are three inputs and two outputs, but note that the output of the first ALU may or may not be used by a subsequent instruction. This ALU configuration is used to implement conventional instruction fusion in DeCAs and is the most widely studied configuration, which includes the LaZy Superscalar [ref] results referenced in Chapter 2. Figure 3.1(a) shows the conventional cascading configuration.

3.1.1 Extending Conventional Fusion

In addition to conventional fusion, we also explore the combination of more than two operations by cascading three ALUs using three new cascaded ALU configurations: wide consumer cascading, wide producer cascading, and deep cascading. Each of these configurations is a four input and three output configuration. These new cascaded ALU designs facilitate exploiting new kinds of fusion opportunities and each can be employed by DS-DeCAs and SS-DeCAs.

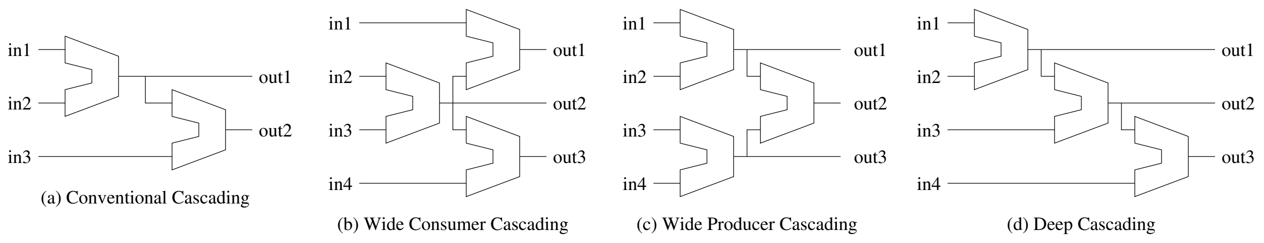


Figure 3.1: Conventional and New Cascaded ALU Configurations

3.2 New Fusion Types

3.2.1 Wide Consumer Cascading

Wide consumer cascading refers to the case where there are two consumer instructions for a single producer instruction. The results of the producer instruction are used as input to the two consumer instructions, necessitating the cascaded ALU configuration depicted in Figure 3.1(b). This form of fusion is applicable when the data dependence graph has a *fork* in it and allows for collapsing two branches of the critical path in a balanced manner, unlike conventional fusion which can only reduce a single branch.

3.2.2 Wide Producer Cascading

Wide producer fusion occurs when a consumer instruction requires the results of two producer instructions. In this case, the results from two ALUs are used as the inputs to a third ALU. This configuration is depicted in Figure 3.1(c). This form of fusion is applicable when there is a *join* in the data dependence graph. Like wide consumer fusion, wide producer fusion also provides for reducing the height of the critical path in a balanced manner.

3.2.3 Deep Cascading

A third new fusion type that we introduce is *deep* cascading, depicted in Figure 3.1(d). Deep cascading refers to fusing three consecutive dependent instructions so that the result of the first instruction is used as an input to the second instruction and the result of the second instruction is in turn used as an input to the third instruction. Whereas wide consumer and wide producer cascading may reduce the height of the critical path in a balanced manner, deep cascading can simultaneously collapse multiple execution steps and lead to greater dependence height reduction of the critical path. Another interesting consequence of employing deep cascading is potentially lengthening the execution stage of the processor pipeline, which would lower the clock speed for such processors. Scheduling algorithms that include the possibility of deep cascading may yield competitive or better performance than processors with higher clock speeds, which could reduce power consumption and energy usage.

3.3 Applying the New Fusion Types

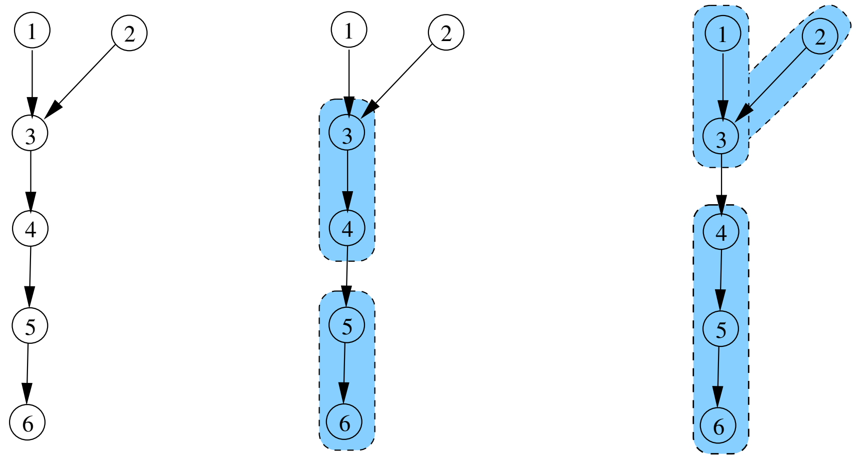
By making use of the new fusion configurations previously discussed, we can potentially increase fusion coverage and reduce the height of the critical path beyond what is achievable by only using conventional fusion. Consider the instructions in Figure 3.2(a). They correspond to the DAG shown in Figure 3.2(b). Using only conventional fusion, instructions 3, 4, 5, and 6 can be covered by fusion and the height of the critical path through the DAG can be collapsed from five to two cycles. In cycle one, instructions 1 and 2 independently execute. In cycle two, instructions 3 and 4 are conventionally fused and execute, and in cycle three instructions 5 and 6 are conventionally fused and execute. This leads to the fusion groups shown in Figure 3.2(c).

Now consider Figure 3.2(d). If we include the new fusion types, the entire DAG can be covered by fusion and the height of the critical path through the DAG can be reduced from five to three cycles. In cycle one, instructions 1, 2, and 3 are fused and execute as a wide consumer fusion group. In cycle two, instruction 4, 5, and 6 are fused and execute as a deep fusion group.

Figure 3.2(e) shows a set of instructions that generate a DAG with a long instruction chain. If a DeCA only supports conventional fusion, the height of the critical path through the DAG can be collapsed to three cycles, by fusing the instruction pairs 1 and 2, 3 and 4, and 5 and 6. If we extend conventional fusion to include deep fusion, long chains can be collapsed more rapidly using fewer fusion groups. Figure 3.2(h) shows deep fusion applied to the same DAG, which reduces the height of the critical path to two cycles by fusing instructions 1, 2, and 3 and executing them in the first cycle, and fusing instructions 4, 5, and 6 and executing them in the second cycle.

```
1. r13=r2+r3;
2. r14=r1-r4;
3. r15=r13+r14;
4. r16=r15+r14;
5. r17=r16-r10;
6. r18=r17-r2;
```

(a) Instructions



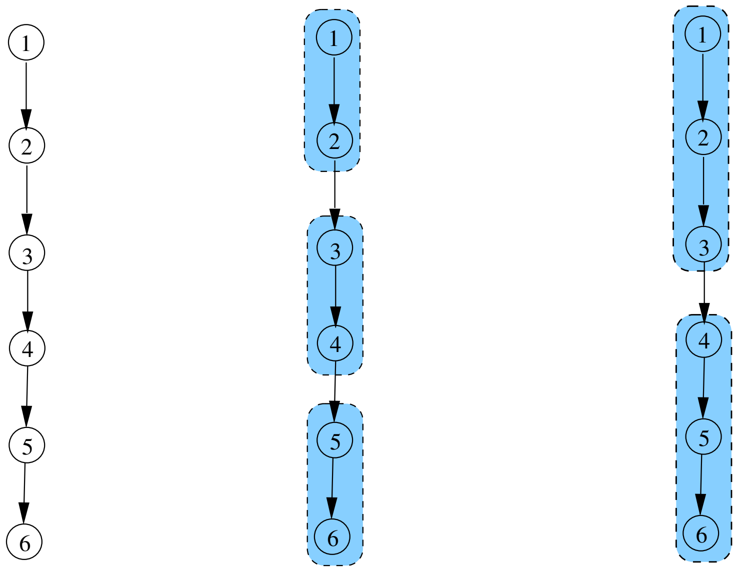
(b) Instruction Dependences DAG

(c) Only Using Conventional Fusion

(d) Using New Fusion Types Wide Consumer and Deep Fusion

```
1. r13=r2+r3;
2. r14=r13-r4;
3. r15=r14+r14;
4. r16=r15+r14;
5. r17=r16-r10;
6. r18=r17-r2;
```

(e) Instructions



(f) Instruction Dependences DAG

(g) Only Using Conventional Fusion

(h) Using Deep Fusion

Figure 3.2: Applying the New Fusion Types

CHAPTER 4

COMPILER SUPPORT FOR FUSION

In general, compiler support can improve the performance benefits from instruction fusion in an out-of-order multiple-issue processor in two ways: annotation and instruction scheduling. Annotating fusible instructions can guide fusion in processors that support fusion by providing hints that make it clear when instructions should be delayed to allow for fusion and when they should not be. Scheduling fusible instructions closer together allows for fusion packets to be executed earlier and ensures that instructions are given priority based on their position on the critical path.

In this work, we use the VPO (Very Portable Optimizer) compiler targeted to the MIPS ISA. VPO is a lightweight research compiler developed by the University of Virgina and Florida State University. For this research, we extend VPO to produce a DAG of instruction dependencies, identify the critical path through the DAG, and annotate fusible instructions present in the DAG. In addition, we add the ability to schedule instructions based on the critical path through the DAG, and allow the scheduler to give priority to fused instructions, which will be discussed in detail in Section 4.2.

4.1 Annotating Instructions for Fusion

In DeCAs, it is necessary to delay producer instructions until their consumer instructions demand their results allowing the producer and consumer instructions to be fused and executed in the same cycle. However, if an instruction is not on the critical path, then delaying the execution of the instruction will not provide a performance benefit and can actually harm performance if instructions that are on the critical path are prevented from executing because an instruction not on the critical path uses a resource the critical path instruction requires to execute. In order to mitigate these deleterious effects, the compiler can:

1. Identify the critical path of instructions through the program.
2. Identify dependent instructions on the critical path.
3. Mark dependent fusible instructions on the critical path to present hints to the processor so it knows when to delay and when not to delay an instruction.

In order to identify the critical path of instructions through the each basic block, we generate a dependence DAG, assign cycle-time values to nodes in the DAG based on the instruction type, and calculate the critical values of each node in the DAG. The DAG is the same dependence DAG that is used to schedule instructions in a basic block. Instructions in the DAG are marked based on the cascaded ALU configurations discussed in Sections 3.2 and 3.3. If instructions on the critical path through the DAG are fusible, they are marked and the fusion group is treated as requiring one cycle, which alters the information represented by the DAG by collapsing the critical path and can generate a new critical path through the basic block. The updated critical path is used to identify and mark new fusible instructions on the critical path, until all possible fusible instructions on the critical path are assigned a fusion configuration. Figure 4.1 shows the algorithm used to mark instructions for fusion.

```
FOR each basic block in the program DO

   DAG = build_dag(basic_block)
   assign_instruction_cycle_time_values(DAG)
   assign_critical_values_to_identify_critical_path(DAG)

   WHILE there are unvisited nodes in the DAG DO

      i1 = get_inst_with_highest_critical_value(DAG)

      IF (i2, i3) = wide_consumer_fusion(i1)
          mark_wc_fusion(i1, i2, i3)
          update_crit_value_to_zero(i1, i2, i3)
          mark_as_visited(i1, i2, i3)
      ELSE IF (i2, i3) = wide_producer_fusion(i1)
          mark_wp_fusion(i1, i2, i3)
          update_crit_value_to_zero(i1, i2, i3)
          mark_as_visited(i1, i2, i3)
      ELSE IF (i2, i3) = deep_fusion(i1)
          mark_deep_fusion(i1, i2, i3)
          update_crit_value_to_zero(i1, i2, i3)
          mark_as_visited(i1, i2, i3)
      ELSE IF i2 = conventional_fusion(i1)
          mark_conventional_fusion(i1, i2)
          update_crit_value_to_zero(i1, i2)
          mark_as_visited(i1, i2)
      ELSE
          update_crit_value_to_zero(i1)
          mark_as_visited(i1)
```

Figure 4.1: Algorithm for Marking Fusion Types within Basic Blocks

When annotating fusible instructions to provide support for dynamically scheduling and executing fusible instructions more effectively we mark the instructions in one of four ways: conventional, wide consumer, wide producer, or deep. In LaZy Superscalar, the execution of single-cycle integer ALU producer instructions was delayed until the result of the instruction was demanded by a fusible consumer instruction, the result was demanded by a non-fusible consumer instruction, or the result was overwritten by a subsequent instruction. The result of the producer instruction is demanded by a fusible consumer and both instructions fused, the result is demanded by a non-fusible and they separately execute, or the result was overwritten and the delayed producer instruction was killed. In either case, the default behavior was to delay a single-cycle ALU instruction until it was demanded or killed.

Like the conventional fusion case, *wide producer fusion* involves delaying producer instructions—in this case two producers instead of one—until a consumer instruction demands their results and all three instructions can be fused by the ALU configuration in Figure 3.1(c). In LaZy Superscalar, by default a producer instruction is immediately fused with the consumer instruction upon the consumer demanding the producer's result. This default behavior must be altered to support *wide consumer fusion*. To enable wide consumer fusion, the compiler marks a producer ALU instruction to delay until two consumer ALU instructions demand the result of the producer and all three instructions are fused and executed on the ALU configuration in Figure 3.1(b). To support *deep fusion* the compiler marks the first and second instructions in the chain to wait for the third instruction to demand the result of the second instruction, which in turn demands the result of the first instruction, causing all three to fuse and be executed by the ALU configuration in Figure 3.1(d).

4.1.1 Annotation Scheme

Initially, the critical path is determined, fusible instructions are identified, and fusion types for each group of fusible instructions are selected. Next, the instructions are annotated to provide hints to the DeCA. The hints provided by the annotations aid the DeCA in determining when instructions should be delayed and when they should not be. The annotation scheme we employ requires that we identify which instructions in a fusion group are *fuseheads*, which instructions in a fusion group are *fusetails*, and whether or not fusion should be delayed, i.e., whether or not the instruction should *wait*, until a second consumer instruction is encountered, as is the case with wide consumer fusion. A *fusehead* refers to an instruction in a fused group of instructions that produces a result that at least one other instruction in the group consumes. A *fusetail* refers to an instruction in a fused group of instructions that consumes a result another instruction in the fusion group produces, but that does not itself produce a result that another instruction in the group consumes. An instruction will be marked to *wait* only in the case that it is the first consumer instruction in a wide consumer fusion group, so that the DeCA will not fuse the first producer/consumer pair until the second consumer is encountered. Given this annotation scheme, we are able to appropriately mark all fused instruction groups according to the fusion type selected to fuse the group. Figure 4.2 shows which nodes in each fusion configuration are marked as fuseheads, fusetails, and which are marked to wait.

To illustrate how this annotation scheme is applied to mark instructions based on fusion type, consider the instructions depicted in Figure 4.3(a). These instructions form the corresponding DAG depicted in Figure 4.3(b). We mark fusehead instructions with an *f*, the first consumer instruction in a wide consumer group with a *w*, and fusetails do not require an annotation. Fusetails do not require annotation because they are responsible for sending the demand signal that completes the fusion group and causes it to execute and they should not be delayed.

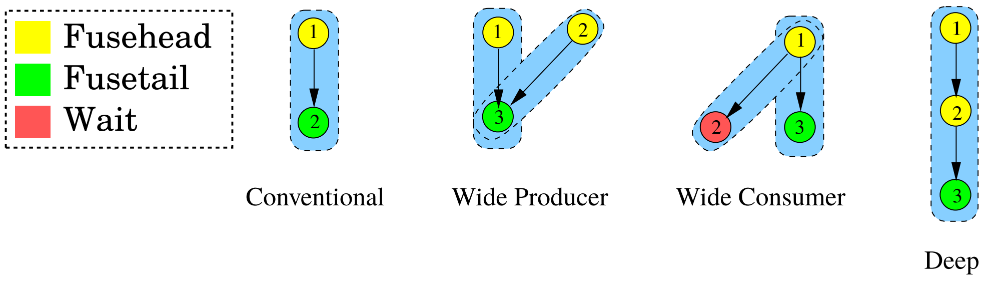
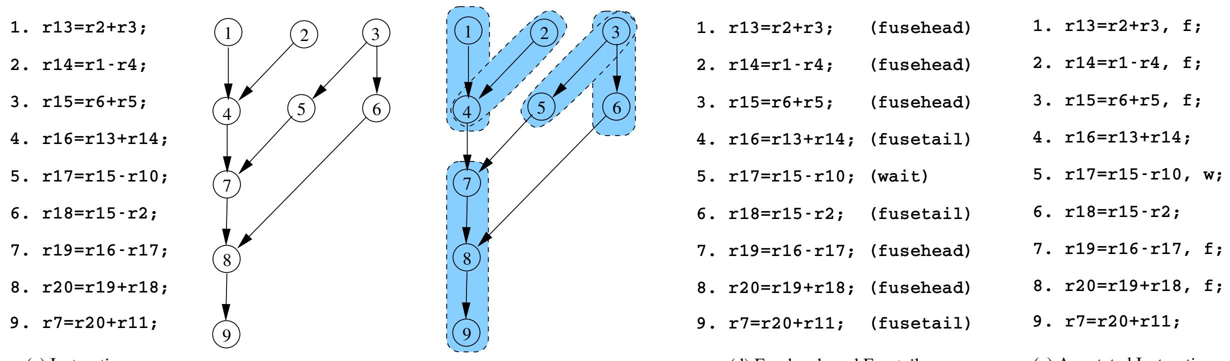


Figure 4.2: Annotating the Various Fusion Types

Now consider Figure 4.3(c). Assume each instruction has a cycle time cost of one. Using the algorithm outlined in Figure 4.1 to mark fusion types, we identify the critical path through the DAG to be five cycles long and the instructions with the highest critical values to be 1, 2, and 3. We select 1, since it occurs first. Instructions 1, 2, and 4 are identified as a wide producer fusion group. Instructions 1 and 2 are marked as fuseheads (*f*), and instruction 4 is a fusetail, and does not require an annotation. Nodes 1, 2, and 4 have their critical values set to zero and are marked as visited. Now, instruction 3 has the highest critical value and represents the first node on the critical path, so we select it next. Instructions 3, 5, and 6 are identified to be a wide consumer fusion group. Instruction 3 is marked as a fusehead, and instruction 5 is marked to wait (*w*) for instruction 6, the last consumer instruction in the group, which is a fusetail.

Again, they are marked as visited and their critical values set to zero. Now, instruction 7 is the first node on the critical path, so it is selected next. Instructions 7, 8, and 9 are identified to be a deep fusion group and both 7 and 8 are marked as fuseheads (*f*), since 8 consumes the result of 7, and 9 consumes the result of 8. The nodes are marked as visited and their critical values are zeroed out. This completes the algorithm for marking this basic block. The different fusion groups identified are shaded in the graph in Figure 4.2(c). Given this set of fusion groups, the fuseheads, fusetails, and which instructions must wait are shown in figure 4.3(d), which gives us the final set of annotated instructions in Figure 4.3(e).



```
1. r13=r2+r3;
2. r14=r1-r4;
3. r15=r6+r5;
4. r16=r13+r14;
5. r17=r15-r10;
6. r18=r15-r2;
7. r19=r16-r17;
8. r20=r19+r18;
9. r7=r20+r11;
```

(a) Instructions

(b) Instruction Dependences DAG

(c) Wide Producer, Wide Consumer and Deep Fusion Groups

```
1. r13=r2+r3;     (fusehead)
2. r14=r1-r4;     (fusehead)
3. r15=r6+r5;     (fusehead)
4. r16=r13+r14;   (fusetail)
5. r17=r15-r10;   (wait)
6. r18=r15-r2;    (fusetail)
7. r19=r16-r17;   (fusehead)
8. r20=r19+r18;   (fusehead)
9. r7=r20+r11;    (fusetail)
```

(d) Fuseheads and Fusetails

```
1. r13=r2+r3, f;
2. r14=r1-r4, f;
3. r15=r6+r5, f;
4. r16=r13+r14;
5. r17=r15-r10, w;
6. r18=r15-r2;
7. r19=r16-r17, f;
8. r20=r19+r18, f;
9. r7=r20+r11;
```

(e) Annotated Instructions

Figure 4.3: Annotating Instructions for Fusion

This annotation scheme has the added benefit of allowing fusion depth to be extended independent of the compiler. By simply marking an instruction as fusible, we can extend the depth of fusion without adding a new annotation and without recompiling the program. This allows modifications to be made at the hardware level to support deeper fusion, by adding a fourth deeply cascaded ALU for instance, without changing the executable itself and without the need to define a new fusion annotation. Another benefit of this annotation scheme presents itself when we consider annotating cross-block fusion for a fusehead instruction that is part of multiple possible fusion configurations available in multiple successor blocks. Instead of sinking instructions and distinctly annotating each available fusion type, we can simply mark the instructions on each path as if they were in the same block. This is illustrated in further detail in Section 4.1.3.

4.1.2 ISA Changes to Support Fusion

In order to support fusion annotations, ISA changes were required to adequately represent fusion annotation information. Three states require annotation to properly annotate fusion: *fusehead*, *wait*, and *no fusion*. We fuse both R-type and I-type single-cycle ALU instructions so two bits must be set aside in R-type and I-type instructions to represent the three states a fusible instruction can represent. This work was done in conjunction with two projects currently under development as a joint effort between FSU and MTU: DAGDA and DDE. The combined ISA for all three projects we refer to as the *D3 ISA*. The DAGDA project also requires two annotation bits, bringing the total number of bits required for both projects to four bits. The D3 ISA is a modification of standard MIPS that allows for four annotation bits in R-type and I-type instructions.

For R-type instructions, we borrow four bits from the *shamt* field, i.e., the bits in positions 10-6 in a 32-bit MIPS R-type instruction. The *shamt* and the *rs* fields are overlapped as they are never both used in the same instruction. For I-type instructions, we borrow four bits from the *immediate* field, i.e., the bits in positions 15-12 in a 32-bit MIPS I-type instruction. The D3 ISA was modified to ensure that the limitations imposed by a 12-bit immediate field still allow for 16-bit offsets and 32-bit addresses to be generated.

4.1.3 Cross-Block Fusion

Fusion across blocks is critical as much of the work done in a program is often performed in loops, and there are many opportunities for fusion across blocks. Cross-block fusion can be achieved using the same annotation scheme previously discussed. However, fusion across blocks requires us to address a few important considerations that arise when determining when cross-block fusion can be performed, whether or not it is advantageous, and how to ensure that fusion will occur given multiple paths. One issue is the distance between fusible instructions. If an instruction is only fusible to instructions that are separated by multiple blocks, the delay required to reach the fusetail that executes a fusion group might be too large and harm performance. To address this, we limit the fusion depth when determining fusible instructions. The *fusion depth* refers to the maximum depth the control flow graph will be searched to find fusible instructions to fuse to a candidate for fusion.

A second issue arises when we consider that a fusehead that is fused across multiple paths may take advantage of multiple fusion types. If each fusion group were annotated to delineate the specific type of fusion, this issue would have to be addressed by sinking or hoisting instructions so that each possible path contained the full set of instructions in the fusion group specified for that path. However, by using the annotation scheme outlined above, we can avoid the need to move instructions across blocks. Consider Figure 4.4, which depicts the case where both deep fusion and wide consumer fusion are available depending on whether the left or right successor block is chosen. In this case, we can easily annotate both cases for their respective fusion types without the need to specify which fusion type will eventually be encountered. In the top block, the instruction can be marked as a fusehead. In the left successor block, the two instructions that complete the deep fusion group can be annotated by marking the first instruction as a fusehead as well, and the third instruction as a fusetail. The instructions in the right successor block can be marked for wide consumer fusion by annotating the first consumer instruction to wait. The second consumer instruction is a fusetail. This allows both fusion groups to be executed without the need to sink or hoist instructions across block boundaries.

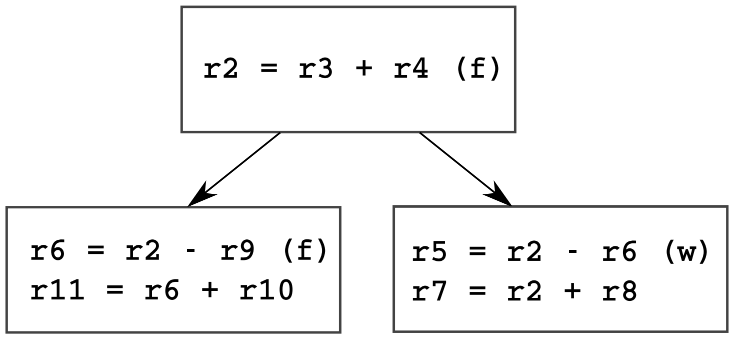
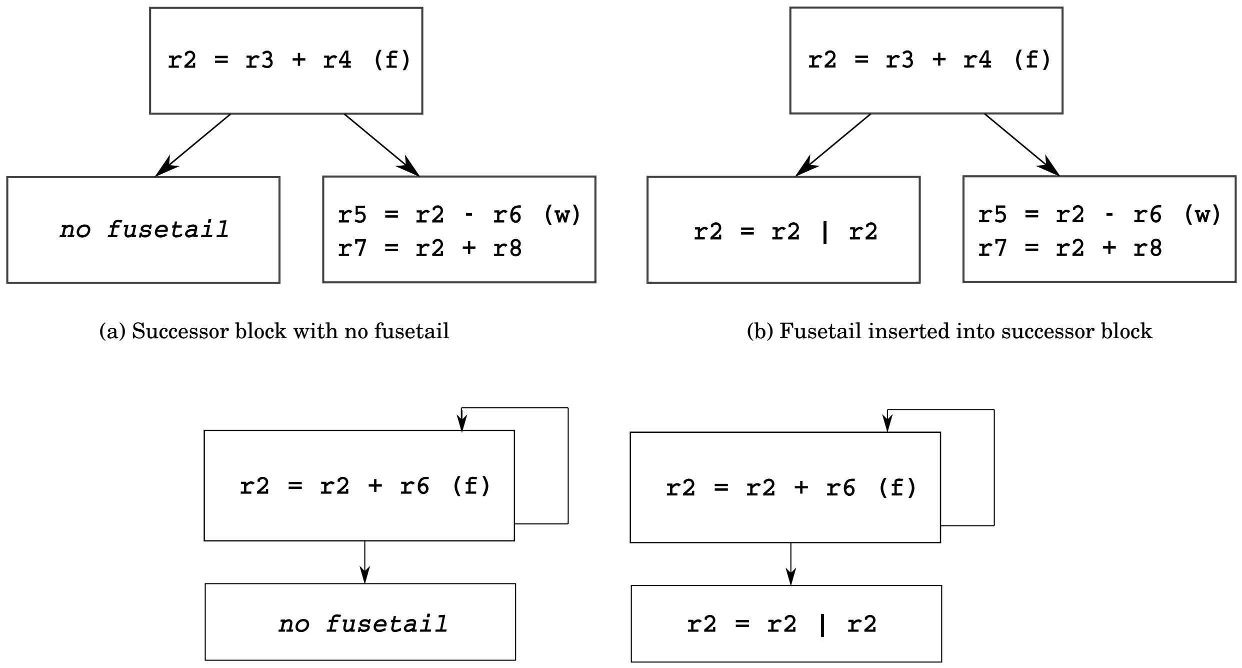


Figure 4.4: Annotating Multiple Fusion Types Across Blocks

A third issue arises when there are cross-block fusion opportunities on some but not all paths exiting a block. This must be addressed to ensure that an instruction marked fusible will execute given that not all paths exiting the block contain fusetails. This is an issue because an instruction that is marked as a fusehead will be delayed until it is demanded by a subsequent instruction. If no consumer instruction exists on one of the paths exiting the block, then the instruction may be unnecessarily delayed waiting for a consumer instruction to issue a demand signal to cause it to execute. In order to address this, when a cross-block fusion opportunity is identified and some paths out of the block do not contain a fusetail that executes the fusehead, a dummy instruction is inserted into successor blocks without fusion opportunities so that the fusehead will be demanded and execute. The dummy instruction that acts as a fusetail is a logical OR operation that ORs the result of the fusehead with itself and is placed at the top of the successor block. Given this, performance should not be greatly affected due to the proximity of the instructions and because both instructions will be fused and executed in the same cycle.

Figure 4.5(a) depicts the case where a cross-block fusion opportunity is present in the right successor block but not the left successor block. In this case, a dummy instruction is inserted into the left successor block to ensure that the fusehead in the predecessor block is executed without a large delay, which is shown in Figure 4.5(b). Figure 4.5(c) depicts the case where cross-block fusion is present across a loop boundary and the loop exit block does not contain a fusetail. Again, a fusetail is inserted into the exit block to ensure the fusehead executes without a long delay, which is shown in Figure 4.5(d).



(a) Successor block with no fusetail

(b) Fusetail inserted into successor block

(c) Loop exit block with no fusetail

(d) Fusetail inserted into exit block

Figure 4.5: Supporting Cross-Block Fusion

4.2 Instruction Scheduling

By scheduling fusible instructions closer together, opportunities for fusion may increase and instructions marked for fusion may execute earlier, reducing the delay time required to wait for fusible consumer instructions. Even if more fusion opportunities are not generated by scheduling fusible instructions closer together, by having multiple fused instructions fetched closer together hardware resources are occupied for fewer cycles because the delay for issuing the initial ALU instructions is shortened.

For a simple example of how scheduling for fusion may be beneficial, consider a simple basic block with only three instructions i1, i2, and i3, where i2 and i3 consume the result of i1. Also, assume that the only fusible instructions are i1 and i3. If the current instruction path through the block is i1 → i2 → i3, but i2 and i3 are independent, a more effective schedule is achievable by scheduling i3 before i2, so that i1 and i3 can be conventionally fused and executed earlier. In this case we alter the schedule from i1 → i2 → i3 to i1 → i3 → i2.

We modify the classic algorithm for instruction scheduling to accommodate giving priority to selecting fusible instructions that are on the critical path and are fused with instructions previously scheduled. The classic algorithm for instruction scheduling uses a DAG to model the dependencies between instructions within a block. A ready set of instructions is created by determining which nodes in the DAG have no dependencies that have not yet been scheduled. The compiler selects an instruction to add to the end of the list of scheduled instructions at each step. To decide which instruction should be selected from the ready set, the compiler determines which instructions can be immediately issued, and which of those instructions is part of the longest chain of dependencies remaining in the DAG, i.e., is on the critical path. In our modified scheduling algorithm, the priorities for selecting instructions from the ready set are updated to be the following in order of priority:

1. If a fusible instruction is in the ready set and if the instruction is on the critical path, schedule it.
    (a) If there are multiple fusible instructions available, give priority to an instruction if another instruction it is fused with has already been scheduled.
2. Else, schedule an instruction that is on the critical path through the DAG.

4.2.1 Classical Scheduling vs Scheduling with Conventional Fusion

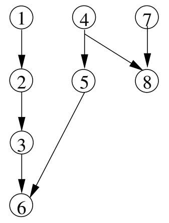
To see how the modified scheduling algorithm combined with instruction fusion can benefit performance, consider the instructions and DAG in Figure 4.6. Assume a multiple issue processor that can simultaneously issue and execute four instructions. If we use the classical approach to instruction scheduling, independent instructions will be aggressively scheduled and grouped closer together. Figure

4.6(c) shows the result of scheduling following the classical algorithm, in which each row can be simultaneously issued. The critical path through the DAG limits the achievable performance benefits and four cycles are required to issue and execute eight instructions.

Now assume the processor allows for conventional instruction fusion and we use the modified scheduling algorithm discussed above. Instead of selecting instruction 1, 4, and 7 in the first cycle, we select instructions according to the priority order previously outlined. First, instruction 1 is selected because it's on the critical path. 1 is fused with 2, so 2 is scheduled next. The critical path is altered and becomes the chain 4 → 5 → 6, so instruction 4 is selected next. 5 is fused with 4, and since 4 was already scheduled, we schedule 5 next, leading to the schedule in cycle 1 shown in 4.5(d). Now, the critical path given the remaining instructions is two instructions long and consists of the chains 3 → 6 and 7 → 8. Since the height of both remaining dependence chains is the same, we select 3 simply because it occurs first, and 6 is selected next because it is fused with 3. Now only 7 and 8 are left and are fused, so both are scheduled, leading to the schedule in cycle two in 4.5(d). This reduces the total number of cycles necessary to execute all eight instructions from four cycles to two cycles, cutting execution time in half.

```
1. r13=r2+r3;
2. r13=r13-r4;
3. r13=r13+r5;
4. r14=r7+r8;
5. r6=r14-r9;
6. r1=r13-r6;
7. r15=r11-r12;
8. r10=r15+r14;
```

(a) Unscheduled Instructions



(b) DAG Representing Instruction Dependences

```
cycle 1:  1. r13=r2+r3;     4. r14=r7+r8;    7. r15=r11-r12;
cycle 2:  2. r13=r13-r4;    5. r6=r14-r9;    8. r10=r15+r14;
cycle 3:  3. r13=r13+r5;
cycle 4:  6. r1=r13-r6;
```

(c) Traditional Multi–Issue Scheduling

```
cycle 1: [1. r13=r2+r3;    2. r13=r13-r4;] [4. r14=r7+r8;    5. r6=r14-r9;  ]
cycle 2: [3. r13=r13+r5;   6. r1=r13-r6; ] [7. r15=r11-r12;  8. r10=r15+r14;]
```

(d) Multi–Issue Scheduling with Instruction Fusing

Figure 4.6: Example of Scheduling for Instruction Fusion

4.2.2 Scheduling with New Fusion Types

Given the instructions in Figure 4.7(a) and the DAG in Figure 4.7(b), there are multiple schedules possible using different combinations of conventional, wide consumer, wide producer, and deep fusion. A traditional superscalar processor with enough resources would be able to complete all eight instructions in four cycles by completing instructions 1, 2, 3, and 5 in the first cycle, 4 and 6 in the second cycle, 7 in the third cycle, and 8 in the fourth cycle. If we apply conventional fusion to the same instructions using the method employed by LaZy Superscalar, a schedule like the one represented in 4.7(c) is possible. In cycle one, instructions 1 and the fused pair 3 and 6 execute; in cycle two 2 and 4 fuse and execute; in cycle three, instructions 5 and 7 fuse and execute; and finally in cycle four, instruction 8 executes for a total of four cycles. This example shows that it is possible that there will be no performance benefit achieved even when conventional fusion is applied to a set of instructions if the fused instructions are not on the critical path.

Now, consider the schedule in Figure 4.7(d). Here *wide producer* fusion is applied to collapse the critical path by fusing and executing instructions 1, 2, and 4 in cycle one. Instructions 3 and 6 also fuse and are executed in cycle one, and instruction 5 independently executes in cycle one. Instructions 7 and 8 fuse and are executed in cycle two, which completes the set of eight instructions in two cycles instead of four.

Applying *deep fusion* can also reduce the number of cycles required to execute the set of eight instructions to two cycles. Figure 4.7(e) depicts how this can be achieved. In cycle one, the independent instructions 1, 2, and 5 and the fused pair 3 and 6 can execute. In cycle two, the dependence chain 4 → 7 → 8 executes, leading to a two-cycle reduction in execution time.

Figure 4.7(f) shows a modified DAG with a dependence added between instructions 3 and 5. Given the dependencies depicted in the new DAG, *wide consumer* fusion can be used to fuse instructions 1, 2, and 5 and *wide producer* fusion can be used to fuse instructions 3, 5, and 6 and both fusion groups can execute in cycle one. In cycle two, instructions 7 and 8 fuse and are executed, reducing the total number of cycles required to execute the eight instructions to two cycles.

In Figure 4.7(g) we again modify the DAG to add a new dependence from instruction 2 to 5. Given the new dependency information, we can apply conventional fusion to fuse the instruction pairs 2 and 5, and 3 and 6, and execute them in the first cycle along with instruction 1. In the second cycle, we apply deep fusion to fuse instructions 4, 7, and 8 and execute them, completing all eight instructions in two cycles instead of four.

These examples illustrate that there are some cases for which applying fusion produces no performance benefit because the fused instructions are not on the critical path, as in Figure 4.7(c). Also, we can see that commensurate performance benefits can be obtained using different fusion combinations, as is the case in Figures 4.7(d) and 4.7(e). The advantage of employing the new fusion types discussed in Section 4.6 is also made clear. Without the addition of wide consumer, wide producer, and deep fusion, we would not have been able to collapse the height of the critical path through the DAGs in Figures 4.7(f) and 4.7(g) from four to two due to the dependences represented in the DAGs.

```
1. r13=r2+r3;
2. r14=r1-r4;
3. r15=r6+r5;
4. r16=r14+r8;
5. r17=r7-r10;
6. r18=r15-r2;
7. r19=r16-r17;
8. r20=r19+r18;
```

(a) Instructions

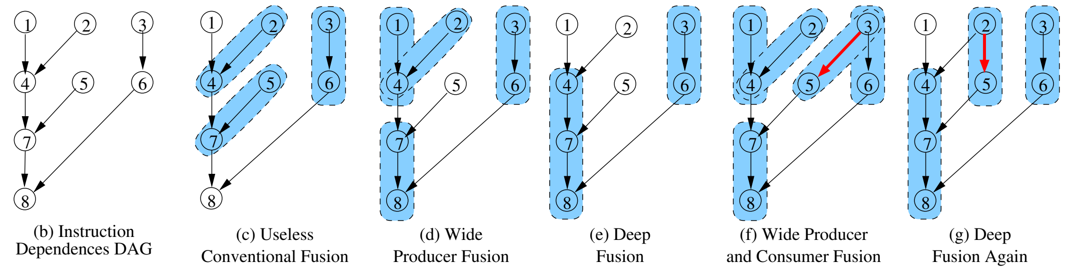

(b) Instruction Dependences DAG
(c) Useless Conventional Fusion
(d) Wide Producer Fusion
(e) Deep Fusion
(f) Wide Producer and Consumer Fusion
(g) Deep Fusion Again

Figure 4.7: Non-beneficial and Beneficial Fusion Scheduling

CHAPTER 5

EVALUATION

5.1 Experimental Setup

To gather results, we run the benchmarks from the Spec2006 Integer suite and the MiBench suite shown in Table 5.1. To compile the benchmarks, we use the VPO (Very Portable Optimizer) compiler targeted to the MIPS ISA and modified to produce fusion annotations and perform instruction scheduling for fusion. In the future, we intend to take advantage of ADL, a language to facilitate building architectural simulators developed by MTU, to develop a simulator that implements fusion in an out-of-order superscalar processor. Currently, we use an in-order MIPS simulator written in ADL that executes the D3 ISA to obtain results. In addition to the benchmarks, the standard library was also compiled, annotated, and scheduled for fusion using VPO.

Table 5.1: Benchmarks Tested

Suite	*Category*	*Benchmark*	*Suite*	*Category*	*Benchmark*
Mibench	automotive	bitcount	**Spec2006**	integer	400.perlbench
		qsort			401.bzip2
		susan			429.mcf
	consumer	jpeg			445.gobmk
		tiff			456.hmmer
	network	dijkstra			458.sjeng
		patricia			403.gcc
	office	ispell			464.h264ref
		stringsearch			
	security	blowfish			
		pgp			
		rijndael			
		sha			
	telecomm	adpcm			
		CRC32			
		FFT			
		gsm			

5.2 Results

We generate two sets of data that reveal information about the availability and types of instruction fusion that occur in the benchmarks previously introduced. First, we provide how many instructions are covered by each type of fusion both statically and dynamically in each benchmark. For dynamic coverage, we annotate library code used by both benchmark suites and this is included in the coverage results. For static coverage, we show coverage for code specific to each benchmark, but not annotated library code. Second, we provide the percentage of dynamic and static instruction combinations identified in instructions marked for conventional and deep fusion in both Spec and MiBench.

5.2.1 Fusion Coverage

Table 5.2 shows the percentage of dynamic instructions covered by each type of fusion in the Spec2006 Integer benchmarks. For conventional fusion, the most coverage occurs in *456.hmmer*, in which 39.58% of dynamically executed ALU instructions are covered, and the least conventional fusion coverage occurs in *429.mcf*, with only 12.18% of instructions covered by conventional fusion. In *456.hmmer*, as in many of the benchmarks tested, address generation involving *lui* and *ori* instructions account for a large portion of conventional fusion. For deep fusion, the most coverage occurs in *401.bzip2*, with 36.97% of instructions covered by deep fusion, and the least deep fusion coverage occurs in *456.hmmer*, with only 6.71% coverage. In deep fusion, cross-block fusion involving increments and shifts occurs frequently, as can be seen from the instruction combination data discussed in the next section. Both wide producer and wide consumer fusion occur much less frequently than either conventional or deep fusion in all of the Spec2006 benchmarks tested, but wide consumer fusion does approach deep fusion coverage in *456.hmmer*.

On average, 21.78% of dynamically executed instructions in the Spec2006 benchmarks tested were covered by conventional fusion, and 24.38% were covered by deep fusion. Wide consumer and wide producer fusion combined accounted for less than 3% of coverage on average. All fusion types combined covered almost half (48.88%) of dynamically executed instructions on average. Interestingly, we find that deep fusion opportunities occur more frequently than conventional fusion opportunities, and this can be explained by the decision to mark as deep fusion cross-block fusion opportunities in which an instruction demands a value it itself produces, rather than marking these cases as conventional fusion.

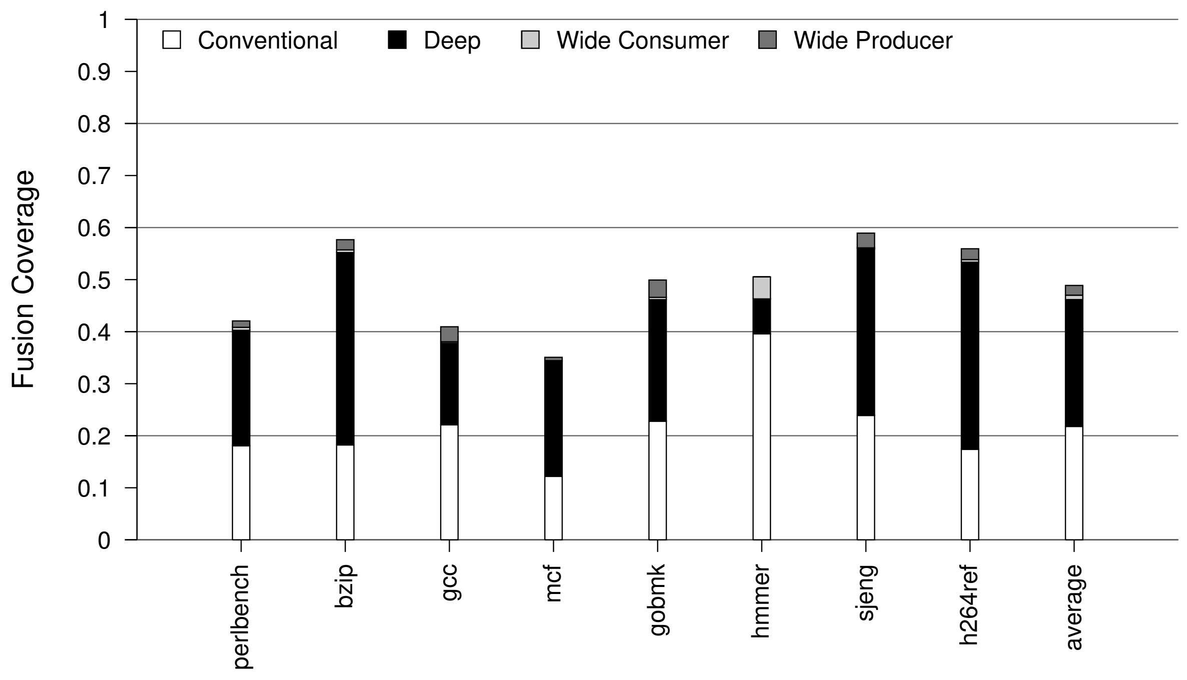


Figure 5.1: Dynamic Fusion Coverage (Spec2006)

Table 5.3 shows fusion coverage obtained statically for code specific to each Spec benchmark and does not include code in libraries used by each benchmark. Total fusion coverage results are similar to the results for dynamic coverage on average–40.75% of instructions are covered statically as opposed to 48.88% covered dynamically. As expected, fusion coverage is generally higher dynamically than statically, but in some cases coverage is higher statically, as in deep fusion in *456.hmmer*. This can be accounted for by dynamic execution of library code and annotated code within loops.

Table 5.4 shows the percentage of dynamic instructions covered by each type of fusion in the MiBench benchmarks. The *susan* benchmark from the automotive category provides both the least conventional fusion coverage and the most deep fusion coverage, at 8.10% and 59.28% respectively. For deep fusion, this is a difference of 22.31%, from the largest coverage of 36.96% in *464.h264ref* in Spec. The most conventional fusion coverage occurs in *adpcm* at 45.22%, which is 5.64% higher than the 39.58% identified in *456.hmmer* in Spec. The least deep fusion coverage occurs in *qsort* at only 1.93% of dynamically executed instructions. This is a difference of 4.78% when compared to the low of 6.71%, again identified in

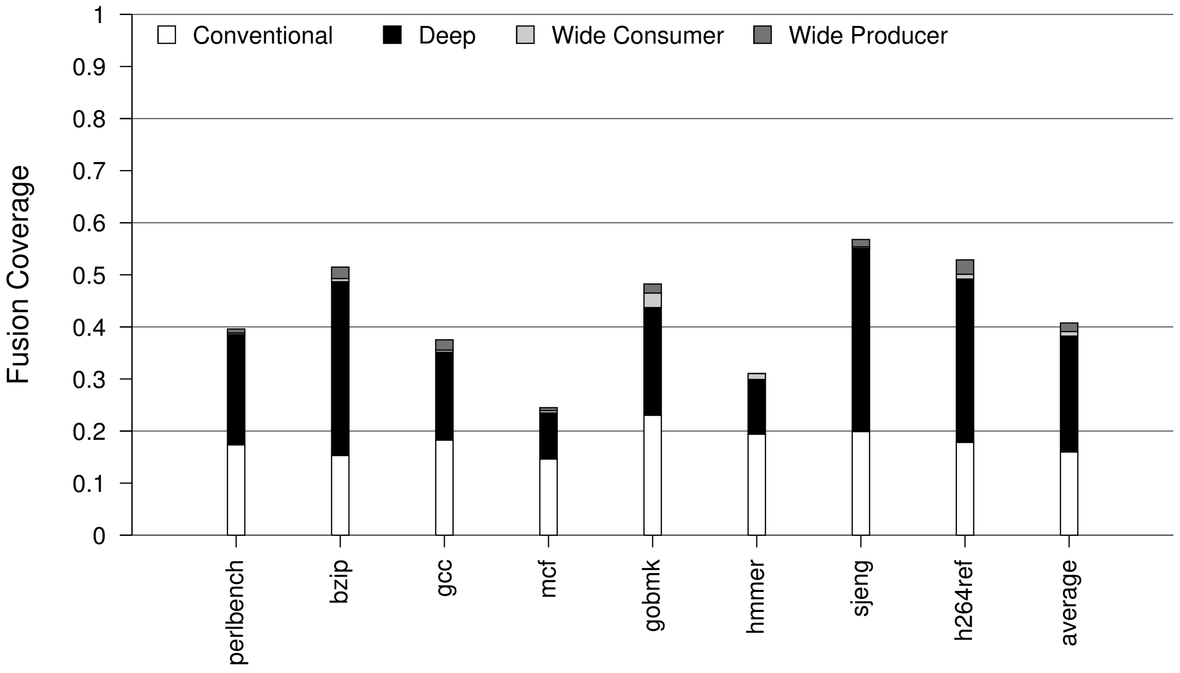


Figure 5.2: Static Fusion Coverage (Spec2006)

456.hmmer. Again, wide producer and wide consumer fusion occur much less frequently, but wide producer fusion occurs more frequently than conventional fusion and shows similar coverage to deep fusion in two of the seventeen benchmarks tested, *dijkstra* and *sha*. In both *dijkstra* and *sha*, we find that wide producer fusion occurs in tight loops, which explains the higher frequency in these benchmarks.

On average, coverage is similar to the results obtained for the Spec2006 Integer benchmarks. For conventional fusion, 25.15% of dynamically executed instruction were annotated in MiBench, which is slightly higher than the 21.78% covered in Spec. For deep fusion, 25.20% of dynamically executed instructions were covered, which again is slightly higher than the 24.38% covered in Spec. Despite high rates of wide producer fusion in *dijkstra* and *sha*, on average wide producer fusion coverage is still low, at only 2.45%. In total, slightly more than half of all ALU instructions dynamically executed belong to one of the fusion types annotated.

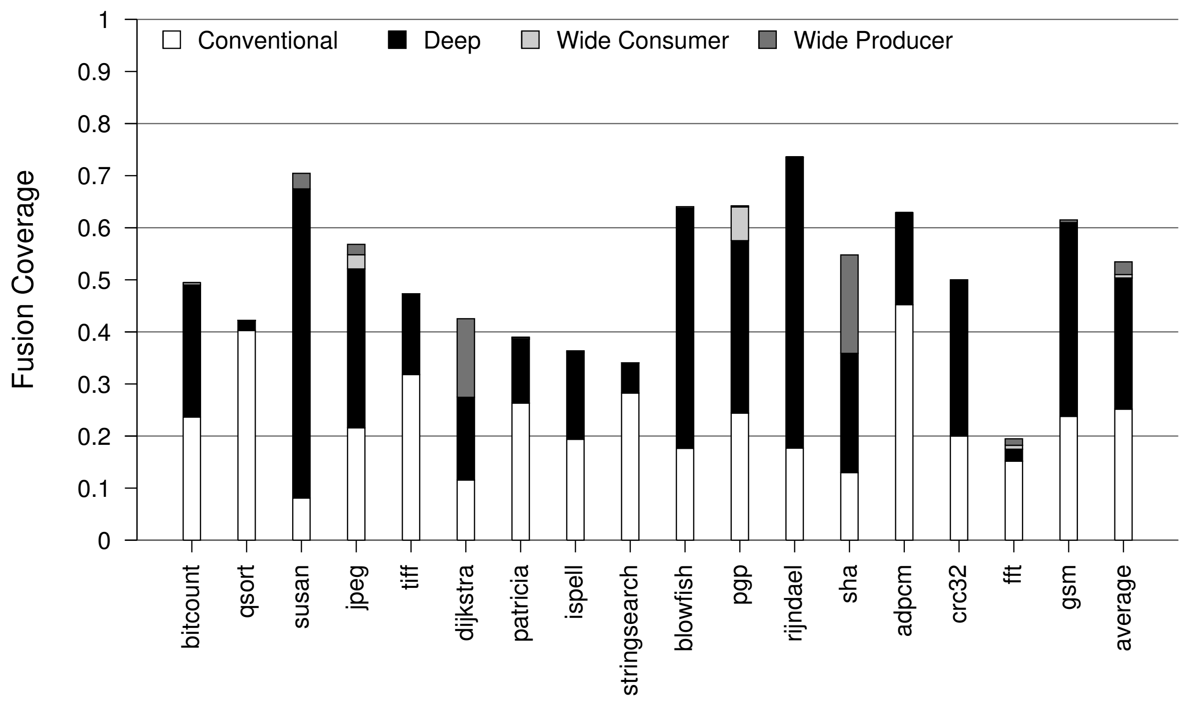


Figure 5.3: Dynamic Fusion Coverage (MiBench)

5.2.2 Fused Instruction Combinations

Table 5.2 shows dynamic conventional fusion instruction combinations identified in the MiBench benchmarks listed in Table 5.1. The instruction matrix shows the frequency of producer/consumer pairs present in conventionally fused instructions as a percentage of total conventional fusion combinations found. Each cell represents a producer/consumer pair. The producer instructions correspond to the rows and the consumer instructions correspond to the columns in the table. For instance, we can see that the combination *add* → *add* occurs for 21.69% of all conventional fusion combinations identified. The combinations *add* → *add*, *add* → *sub*, *add* → *or*, *add* → *slt*, *sll* → *srl*, and *lui* → *or* account for almost 70% of conventional fusion instruction combinations identified.

Table 5.3 and 5.4 show dynamic deep fusion instruction combinations identified in the MiBench benchmarks. We use two tables to represent deep fusion combinations by listing the frequency of the first producer/consumer pair in the deep fusion group in the first matrix, and the frequency of the second producer/consumer pair in the second matrix. For example, we can see that the instruction combination

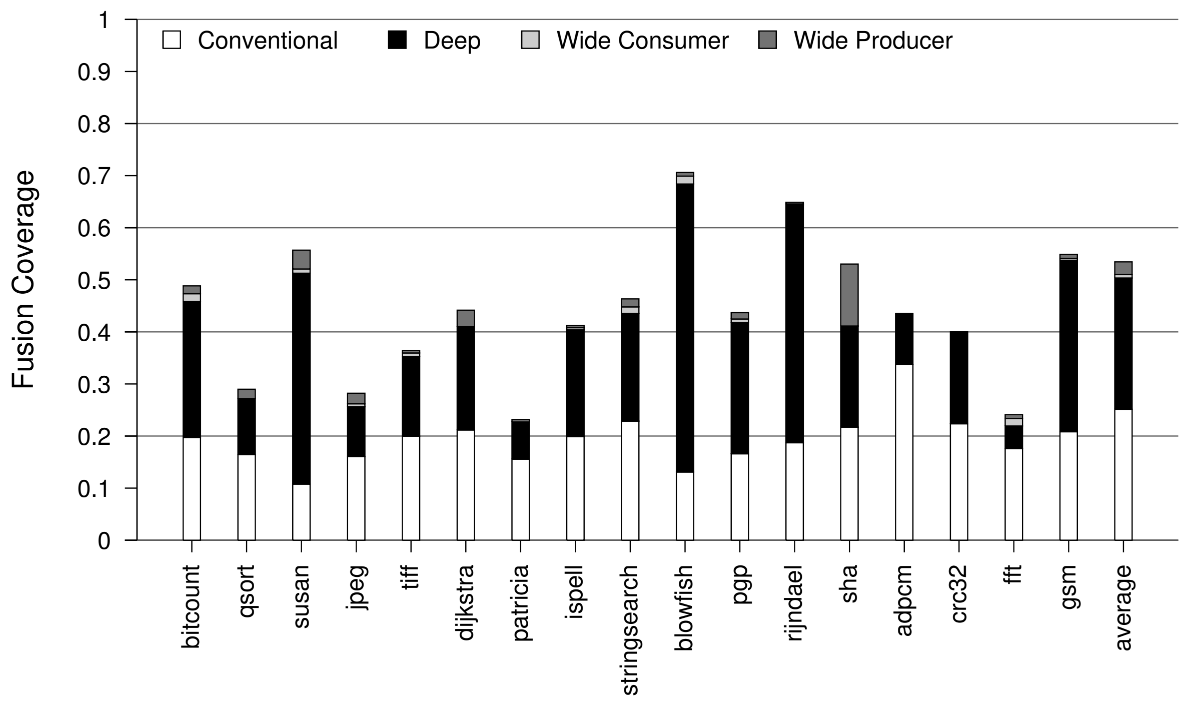


Figure 5.4: Static Fusion Coverage (MiBench)

Table 5.2: Dynamic Conventional Fusion Combinations (MiBench)

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	21.69	0.23	0	0	0	4.75	0.01	13.19	1.09	0.02	0.05	5.22	0
AND	3.9	0	0	0	0	0.07	0	0	0	0	0	2.28	0
LI	1.19	1.33	0	0	0	0.15	0.03	2.07	0.99	0	0	2.18	0
LUI	0.12	0	0	0	0	9.69	0	0	0	0	0	0	0
NOR	0	0.02	0	0	0	0	0	0	0	0	0	0	0.02
OR	0.06	0	0	0	0	1.14	0	0	0	0	0	0	0
SLL	2.87	0.05	0	0	0	0.2	0	1.13	0	0.81	15.36	0.1	0
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0	0	0	0	0	0
SRA	2.47	0.06	0	0	0	0.12	0.01	0.5	0	0	0	1.13	0
SRL	0.57	0	0	0	0	0.02	0.06	0.06	0	0	0	0.03	0.13
SUB	0.44	0	0	0	0	0.02	0.01	1.59	0.07	0	0	0.64	0
XOR	0.02	0	0	0	0	0	0	0	0	0	0	0	0.02

add → *sll* occurs 5.58% of the time for the first producer/consumer pair, and *sll* → *sra* occurs 5.12% of the time for the second producer/consumer pair. Instruction combinations involving combinations of adds, shifts, luis, and ors account for over half of the instruction combinations identified in both the first and second producer/consumer pairs.

Table 5.5 shows dynamic conventional fusion instruction combinations identified in the Spec2006 Integer benchmarks listed in Table 5.1. The combination *add* → *add* occurs for 22.82% of all conventional fusion combinations identified. The combinations *add* → *add*, *li* → *sub*, *add* → *or*, *add* → *slt*, *sll* → *srl*, and *lui* → *or* account for over 70% of conventional fusion instruction combinations identified.

Table 5.6 and 5.7 show dynamic deep fusion instruction combinations identified in the Spec2006 Integer benchmarks. The instruction combinations *add* → *add* and *sll* → *add* account for almost a third of the instruction combinations identified for both the first and second producer/consumer pairs. As in the MiBench benchmarks, instruction combinations involving combinations of adds, shifts, luis, and ors account for over half of the instruction combinations identified in both the first and second producer/consumer pairs.

Table 5.3: Dynamic Deep Fusion Combinations (MiBench). First Pair.

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	13.97	3.51	0	0	0	0.13	5.58	0	0	0.18	1.24	0.12	0.46
AND	2.12	0.76	0	0	0	0.09	1.19	0	0	0.13	0.66	0	0
LI	0.43	0.03	0.03	0	0	0.02	0.52	0	0	0.04	0	1.29	0
LUI	0.01	0	0	0.2	0	13.93	0	0	0	0	0	0	0
NOR	0	0.03	0	0	0	0	0	0	0	0	0	0	0
OR	0	0	0	0	0	0.13	0.17	0	0	0	0	0	0
SLL	15.2	0	0	0	0	0.12	1.44	0	0	2.39	24.82	0.01	0
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0	0	0	0	0	0
SRA	0.01	0.01	0	0	0	0.04	0.14	0	0	0	0.04	0	0
SRL	1.73	0.89	0	0	0	0	2.4	0	0	0.02	0.61	0	0.16
SUB	0.11	0.07	0	0	0	0	0	0	0	0.01	0.18	1.57	0
XOR	0.01	0	0	0	0	0	0.1	0	0	0	0	0	0.95

Table 5.4: Dynamic Deep Fusion Combinations (MiBench). Second Pair.

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	23.42	0.09	0	0	0	0.03	0.74	0.24	0.03	2.01	2.86	0.07	0
AND	0.68	0.55	0	0	0	0	0.4	0.1	0	0	0	3.55	0
LI	0	0	0.02	0	0	0	0	0	0	0	0	0	0
LUI	0.07	0	0	0	0	0	0	0.04	0	0	0	0.09	0
NOR	0	0	0	0	0	0	0	0	0	0	0	0	0
OR	10.48	2.9	0	0	0	0.04	0.06	0.02	0.02	0	0.31	0.53	0.02
SLL	1.52	0.41	0	0	0	0.54	0.18	0.06	0.18	5.12	3.01	0.51	0
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0	0	0	0	0	0
SRA	0.55	1.58	0	0	0.1	0	0.52	0	0	0	0	0.01	0
SRL	3.74	2.57	0	0	0.07	1.79	8.33	0.85	0.12	3.38	0.19	10.44	0.27
SUB	1.54	0	0	0	0	0	0.08	1.35	0	0	0	0.02	0
XOR	0.49	0.16	0	0	0	0	0	0	0	0	0.25	0	0.7

Table 5.5: Dynamic Conventional Fusion Combinations (Spec2006)

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	22.82	0.74	0	0	0	0.74	0.43	14.8	2.13	0.63	0.25	3.21	0
AND	0.26	0.04	0	0	0	0.02	0.02	0.14	0.07	0	0	1.43	0
LI	2.76	0.04	0	0	0	0.19	0.06	4.42	0.43	0	0.02	7.79	0
LUI	0.09	0	0	0	0	11.23	0	0.01	0	0	0	0.08	0
NOR	0	0.04	0	0	0	0	0	0	0	0	0.13	0.05	0
OR	0	0	0	0	0	0	0	0	0	0	0	0.01	0
SLL	5.78	0.01	0	0	0	2.07	0.02	0.22	0	0.27	9.01	0.26	1.27
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0.01	0	0	0	0	0	0	0	0	0	0	0.02	0
SRA	0.03	0.02	0	0	0	0	0.16	0.04	0	0	0	0.05	0
SRL	0.05	0.08	0	0	0	0.02	0.05	0.08	0.08	0	0.02	3.47	0
SUB	0.41	0	0	0	0	0.02	0.05	1.08	0.02	0	0	0.09	0
XOR	0.05	0	0	0	0	0	0.1	0	0	0	0	0.01	0

Table 5.6: Dynamic Deep Fusion Combinations (Spec2006). First Pair.

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	13.48	5.99	0	0	0	0.34	3.86	0	0	0.01	1.65	0.09	0.09
AND	0.44	0.33	0	0	0	0	7.85	0	0	0	0.03	0	0
LI	1.41	0.09	0.19	0	0	0.11	0.21	0	0	0	0.01	0.16	0
LUI	0.18	0	0	0.39	0	20.12	0	0	0	0	0	0	0
NOR	0	0.01	0	0	0	0	0	0	0	0	0	0	0.03
OR	0	0	0	0	0	0.11	0.03	0	0	0	0.05	0.01	0
SLL	21.6	0.01	0	0	0	0.33	2.35	0	0	0.62	9.21	0.06	0
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0	0	0	0	0	0
SRA	0.07	0	0	0	0	0.08	0.22	0	0	0.01	0	0.01	0
SRL	0.31	0.14	0	0	0	0	2.97	0	0	0.26	0.05	0	0.65
SUB	2.8	0.22	0	0	0	0	0.58	0	0	0	0.1	0.05	0
XOR	0	0.01	0	0	0	0	0.02	0	0	0	0	0	0

Table 5.7: Dynamic Deep Fusion Combinations (Spec2006). Second Pair.

Type	ADD	AND	LII	LUI	NOR	OR	SLL	SLT	SLTU	SRA	SRL	SUB	XOR
ADD	33.49	0	0	0	0	0.13	0.38	1.29	1.95	0.05	0.41	2.59	0
AND	0.08	0.09	0	0	0	0.09	5.86	0.03	0	0	0	0.65	0
LI	0.11	0	0	0	0	0	0	0.09	0	0	0	0	0
LUI	0.21	0.01	0	0.02	0	0.01	0	0.06	0.01	0	0	0.06	0
NOR	0	0	0	0	0	0	0	0	0	0	0	0	0
OR	17.49	2.32	0	0	0	0.2	0.08	0.22	0.02	0.2	0.01	0.54	0.01
SLL	5.47	0.04	0	0	0	0.05	0.94	0.16	0.02	0.05	10.59	0.78	0
SLT	0	0	0	0	0	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0	0	0	0	0	0
SRA	0	0	0	0	0	0.26	0.12	0	0	0	0	0.51	0
SRL	0.62	1.81	0	0	0.01	0.05	2.64	0.22	0	0	0.01	5.73	0.01
SUB	0.05	0	0	0	0	0	0.08	0.02	0.06	0	0	0.17	0
XOR	0	0.07	0	0	0	0	0.7	0	0	0	0	0	0

CHAPTER 6

RELATED WORK

Malik *et al.* [3] propose a method to collapse two data dependent instructions via interlock collapsing ALUs. In their work, they show that it is possible to execute two data dependent instructions in the same clock cycle by making use of an implementation that uses multiple multi-operand execution units and their results reveal that dependency collapse through interlock ALUs can increase parallelism by 10% on average in a subset of the Spec benchmark suite. Our work extends these findings and introduces compiler annotations and instruction scheduling for dependent instruction collapse, and three new ALU configurations that allow for the collapse of new dependent instruction configurations.

Sharifan *et al.* [7] propose *Chainsaw*, a Von-Neumann based accelerator that amortizes fundamental instruction overhead, like fetch-decode, by adopting the chains instruction abstraction. A *chain* refers to a compiler fused sequence of instructions in a long instruction chain. A chain conveys producer-consumer locality between dependent instructions and the Chainsaw architecture uses this information to schedule a chain on a single execution unit that uses pipeline registers to forward values between dependent operations. Chainsaw uses two strategies to exploit dependent instruction chains. Either it identifies the longest instruction chain possible and assigns it to a lane to be processed by a functional unit, or it identifies the interdependences between long chains of dependent instructions and breaks the chains apart at these points to form smaller chains to increase chain count and reduce stall time since interdependent chains are removed to allow chains to start execution earlier.

Chainsaw does not make use of wide consumer or wide producer fusion, but instead opts to extend the depth of conventional fusion to an arbitrary depth. Also, the goal of Chainsaw is not to use fusion to reduce the height of the instruction critical path, but is instead employed to reduce energy consumption by determining the ins and outs of a dependence chain and avoiding unnecessary reads and writes to a register file by forwarding producer values within the chain to consumer instructions that require them. Furthermore, new ALU configurations are not implemented to support Chainsaw. Instead, whole instruction chains are statically analyzed, information about producer and consumer instructions is stored in an instruction buffer, and a chain is assigned to an instruction lane and processed by a single ALU.

Abboud *et al.* [1] propose a method for reducing the height of algebraic circuits by extracting three-argument instructions (TAI) from sequential arithmetic code to improve VLIW scheduling that supports TAIs. Three TAIs are introduced in the paper: MUL3, ADD3, and MAD, representing two multiplication operations, three addition operations, and an addition and a multiplication operation respectively. The height reduction technique uses a modified version of the MRK theoretical algorithm used to collapse algebraic circuits. The algorithm is modified to reduce the number of instructions emitted by generating TAI VLIW instructions. The results of their paper show that the use of TAIs and the modified MRK algorithm can emit TAIs and improve VLIW scheduling using TAIs in basic blocks with a sufficient number of arithmetic instructions.

This technique only addresses height reduction in algebraic circuits composed of multiplication and addition operations. Also, in order to reduce the height of the algebraic circuit, multiple groups of three interdependent multiplication or addition operations must be identified to package into VLIW instructions composed of multiple TAIs for scheduling, unlike our approach which allows for fusion of single-cycle ALU operations in configurations that do not require exactly three operands. In addition, algebraic circuits must be analyzed to determine how the operations represented in the circuit can be mathematically simplified and combined into a single TAI. Instead of reducing combinations of interdependent multiplication and addition operations by algebraic simplification to package into VLIW instructions composed of TAIs, our method is much more general and instead analyzes the critical path to identify fusible single-cycle ALU operations and reduces the height of the critical path by fusing and simultaneously executing fusible instructions in order to improve performance.

Sassone *et al.* [5] propose a method for increasing embedded power efficiency by avoiding unnecessary writes to a register file. They collapse dependent instructions by marking static strands of instructions to provide information to dynamically collapse instructions. Unlike our approach, their method focuses on dependent strands of instructions that make use of transient operands. They define *transient operands* to be operands that are used in an instruction chain, but are not subsequently used. By identifying instruction strands that make use of transient operands they avoid writes to the register file and improve energy efficiency.

Sazeides *et al.* [6] discuss a method that combines load speculation and data dependence collapsing. They achieve dependence collapsing by sinking dependent operations into the operations that demand their results and create long instructions composed of multiple operations. However, their approach can only collapse instructions within the instruction window, and requires complex logic to recognize dependencies in the decode stage. Also, this approach requires that long instruction words composed of multiple operations are supported and does not allow for synthesis of CISC-like instructions from RISC instructions, nor does it discuss the potential of modifying the execution stage to enable multiple instructions to be executed in the same clock cycle.

CHAPTER 7

CONCLUSIONS

Instruction-level parallelism can be augmented by identifying dependent parallelism in addition to independent parallelism. Instruction fusion via cascaded ALUs can take advantage of dependent parallelism to potentially increase performance by collapsing the critical path via simultaneously executing dependent instructions on the critical path. Compiler support can identify and annotate dependent instructions on the critical path to provide hints to architectures capable of executing multiple dependent instructions in a single cycle, making complex hardware to dynamically identify dependent instructions unnecessary and increasing available parallelism. In addition, the compiler can aggressively schedule dependent fused instructions to take advantage of cascaded ALUs and improve performance. By extending two-deep fusion to include new fusion configurations, especially three-deep fusion, large portions of programs composed of such dependent instructions can be covered by fusion and collapsed to improve performance.

BIBLIOGRAPHY

[1] Fadi Abboud, Yosi Ben-Asher, Yousef Shajrawi, and Esti Stein. Combining height reduction and scheduling for vliw machines enhanced with three-argument arithmetic operations. *International Journal of Parallel Programming*, 40(5):488–513, 2012.

[2] Gorkem Asilioglu, Zhaoxiang Jin, Murat Koksal, Omkar Javeri, and Soner Onder. Lazy superscalar. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 260–271, New York, NY, USA, 2015. ACM.

[3] Nadeem Malik, Richard J. Eickemeyer, and Stamatis Vassiliadis. Instruction-level parallelism from execution interlock collapsing. *SIGARCH Comput. Archit. News*, 20(4):38–43, September 1992.

[4] S. Onder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, pages 80–89, May 1998.

[5] Peter G. Sassone, D. Scott Wills, and Gabriel H. Loh. Static strands: Safely exposing dependence chains for increasing embedded power efficiency. *ACM Trans. Embed. Comput. Syst.*, 6(4), September 2007.

[6] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation &amp; collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 238–247, Washington, DC, USA, 1996. IEEE Computer Society.

[7] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

BIOGRAPHICAL SKETCH

Victor Brunell received a B.A. in Asian Studies and Business from Florida State University (FSU) in June of 2006. After graduation, he moved to Japan in the summer of 2006 to work for the Ministry of Education in Tokushima Prefecture as a coordinator for the English teaching program and as a high school English teacher at Itano High School. Upon returning to the States, he received a scholarship to attend Levin College of Law at the University of Florida and began his studies there in Fall of 2012. Family issues caused him to return to Tallahassee, where he managed his family's water purification business for two years. His interest in computer science led him to pursue a Master's degree in Computer Science at FSU, which he began in the Fall 2015 Semester. At FSU, he worked as a research assistant on compiler research with Dr. David Whalley and as a member of a group of educators and computer scientists working on CSIMMS, a project to develop standards and curricula for computer science education in Florida's middle schools. He also worked as a teaching assistant for courses in programming, data structures and algorithms, and computer organization and design. His research dealt with optimizing compilers and improving performance by increasing instruction-level parallelism.