Sin Yu Ho

Instruction Fusion Limit Study for a RISC-V Architecture

Trinity College

University of Cambridge

Computer Science Tripos - Part II

May 14, 2025

Declaration

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 2451B, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Date May 14, 2025

Acknowledgements

I would like to express gratitude to Dr Jonathan Woodruff, my supervisor, for the amazing help and guidance given consistently throughout the course of the project. I would also like to thank my Directors of Studies, Professor Frank Stajano, Professor Neel Krishnaswami and Dr Sean Holden, for their advice, support and understanding. I would also like to thank everyone who gave invaluable feedback on my dissertation.

Proforma

Project Title: Instruction Fusion Limit Study for a RISC-V Architecture

Examination: Computer Science Tripos - Part II, 2025

Word Count: **11999**[1]

Code Line Count: **3749**[2]

Project Originator: Dr Jonathan Woodruff

Project Supervisor: Dr Jonathan Woodruff, Professor Robert Mullins (UTO)

Original Aims of the Project

Instruction fusion is a promising technique that can significantly improve a processor's performance by fusing multiple instructions into one within the instruction pipeline. RISC-V is in a position to utilise fusion due to its wide range of applications and its simple instruction set. However, there has been no systematic investigation into the effectiveness of a general fusion framework for RISC-V. As such, the goals of this project were to build a library for computer architects and researchers to help incorporate fusion into their design, and to use this library to perform a limit study to characterise the potential benefits of instruction fusion for the RISC-V architecture as a whole.

Work Completed

This project was a complete success, meeting all success criteria. I created a fusion library called HistoSim that could run fusion experiments given a series of instruction trace histograms, designed with computer architects and researchers in mind. I then replicated previous work to validate the library. Finally, I performed a limit study to characterise the effectiveness of instruction fusion on the SPECInt2006 benchmark, gaining insights that are helpful to the wider research community as a whole. These insights were presented at the RISC-V Summit Europe 2025 [1].

Special Difficulties

Personal events and significant mental health difficulties meant that I lost six weeks of work time towards the end of Lent term and at the beginning of Easter break.

[1]computed using `texcount`.
[2]computed using `cloc`.

Contents

Chapter 1

Introduction

This dissertation makes two main contributions. On the engineering side, this dissertation creates a library, HistoSim, that computer architects and researchers can use to experiment with instruction fusion. On the research side, this dissertation proposes a more general approach of considering instruction fusion and performs a limit study to support this approach, using the library created and characterised by the SPECInt2006 benchmark suite.

1.1 Motivation

In this section, we present the motivation behind using instruction fusion, why it might be particularly useful for RISC-V, why a fusion library is useful to the community, and why performing a limit study to evaluate its potential is useful to the field.

1.1.1 What is the Problem?

Instruction Set Architectures (ISAs) are split into two broad categories, **Complex Instruction Set Computer (CISC)** and **Reduced Instruction Set Computer (RISC)**.

There are clear advantages and disadvantages to both. A CISC architecture has better code density compared to a RISC architecture, but the ISA itself consists of many more instructions that every microarchitecture has to implement.

On the other hand, a RISC architecture imposes a limit on the amount of work expressible per instruction. For a high-performance processor that might want to do more work per cycle, using a RISC instruction set might force the processor to take more cycles than strictly necessary to execute the program. Ideally, we would like to have the best of both worlds of having a compact ISA while being able to express more work per instruction when needed.

1.1.2 Why Instruction Fusion?

Instruction fusion, otherwise known as macro-op fusion, is the conceptual counterpart of micro-op splitting. Instead of breaking apart complex instructions into multiple simpler instructions, we fuse multiple simple instructions into a single more complex instruction. A more detailed explanation can be found in Section 2.1.1.

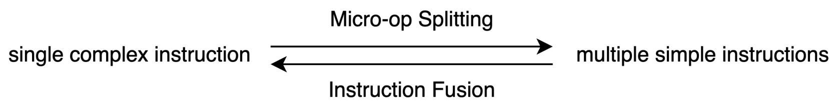


Figure 1.1: The relationship between micro-op splitting and instruction fusion

Micro-op splitting allows a CISC processor to exploit the benefits of a RISC architecture by breaking down the instructions into simpler operations so that there are fewer operations that the processor actually needs to implement. Conversely, instruction fusion allows a RISC processor to exploit the benefits of a CISC architecture by fusing the workload of multiple instructions into one.

Instruction fusion is therefore a good way to reduce the **effective instruction count** while maintaining the simplicity and flexibility of the ISA. Effective instruction count is the number of instructions actually executed by the processor back-end, after taking micro-op splitting and instruction fusion into account.

Microarchitects can choose to fuse certain combinations of instructions into one within their specific microarchitecture, giving them the performance benefits of treating the work as a single instruction, while not bloating the ISA with unnecessary instructions for the other microarchitectures.

1.1.3 Why Instruction Fusion for RISC-V?

The RISC-V ISA subscribes heavily to the RISC philosophy of being consisted of relatively simple instructions, opting to keep the ISA simple instead of introducing complex instructions that are important for one microarchitecture but might not provide significant performance benefit for other microarchitectures. This allows microarchitects to design for a RISC-V architecture more easily, as they have fewer instructions to implement.

However, this flexibility is gained at the expense of performance. Microarchitects that want to increase performance by doing more work in each cycle might benefit from more complex instructions, as they can do more within a single pipeline cycle instead of having to split the work across multiple instructions taking up multiple pipeline cycles.

This problem is especially significant for RISC-V as it targets a wide range of applications, from simple microprocessors to high-end server applications. It needs to strike the balance of being simple enough for the low-end applications to implement against, while not impeding on the performance of high-end applications when expressing the amount of work done per instruction. Therefore, we have reason to think that RISC-V is in a unique position to exploit instruction fusion, as it allows the RISC-V ISA to be kept simple while providing a way for microarchitects to merge complex functionality into a single instruction if they want to.

1.1.4 What has Already Been Done?

Research in the area was inspired by a 2016 paper by Celio et al. [2]. They demonstrated a 5.4% effective instruction count decrease by finding common fusion pairs in the SPECInt2006 benchmark. They also compared this against the ARM and x86 ISA, and noted that fusion applied to compressed RISC-V had a lower effective instruction count than the micro-op count of x86, and it also performed better than ARMv7 but worse than ARMv8.

Other research in instruction fusion include compiler optimisations for fusion [3], fusing non-contiguous memory instructions [4], fusion for floating point benchmarks [5] and fusion in multi-core processors [6].

A common theme across all existing academic literature is that they focus on pairwise instruction fusion. On the other hand, more general instruction fusion is known to be used in industrial microprocessors across various ISAs [7, 8] , including RISC-V [9, 10]. However, none of this work is published or discussed in detail openly.

There are certainly reasons for sticking to only fusing known pairs, e.g. lower complexity, simpler logic and fewer processor resources required. However, the prevalence of industrial microprocessors that implement general instruction fusion suggests that the performance gain is worth the increased complexity. It is also important to characterise the potential gains from fusing more generally so that we know what gains we are forgoing if we opt to fuse only common pairs, further motivating the need for a limit study.

1.1.5 Why a Fusion Library?

A fusion library would be beneficial for computer architects and researchers that want to know how their microarchitecture can make use of fusion. It allows for rapid iteration and discovery of fusion rules that suit their target workload, and would help to push research to head in this direction.

1.1.6 Why Limit Study?

A limit study is an analysis of the best-case performance improvement in a processor that a certain feature, technique or improvement in the performance of a single component can bring. In particular, it takes a step back and examines the feature from a **implementation-agnostic view**, allowing us to gain insight on the impact of that feature without being limited by current microarchitectural designs. Historically, limit studies have provided enlightening insights into the potential of a technique, as well as their fundamental limits [11]. A limit study is useful for investigating instruction fusion for RISC-V specifically as we can find out how much performance can be gained in the best case, independent of what RISC-V microarchitectures look like today. This allows us to evaluate whether instruction fusion is worth pursuing further.

1.2 Contribution of my Project

My project was a complete success, meeting all the criteria outlined in my project proposal. The research results based on this project were **presented at the RISC-V Summit Europe 2025**.

The aims of the dissertation were split into 2 main parts, where (*) denotes an extension:

1. **create a fusion library (HistoSim)** for computer architects and researchers to use when experimenting with how fusion can inform processor design.
    - **establish a framework** for reasoning about instruction fusion in a general sense. The outcome of this was fusion rules and fusion algorithms, which are described in Sections 2.3.3 and 2.3.4 respectively.
    - implement a library that can run instruction fusion experiments on **instruction trace histograms**. The library should be flexible to changes in program behaviour and be scalable.
    - design an **in-order pipeline simulator** to compute the reduction in effective cycle count as a result of fusion (*).
2. **perform a limit study** using HistoSim to gain insight on the benefit of applying more general fusion principles to a RISC-V architecture.
    - decide on a set of **fusion rules** to investigate.
    - **evaluate the effectiveness of instruction fusion** by running experiments using HistoSim on trace histograms of the **SPECInt2006 benchmark suite**, measuring the effective instruction count after fusion.
    - measure the **effective cycle count** after fusion, using the same trace histograms and fusion rules but taking into account pipeline stalls (*).
    - explore **further fusion rules** and evaluate their impact on performance (*).

The evaluation of the project is conducted by **replicating previous work** to validate HistoSim, and presenting the **benchmark results** of applying different fusion rules on the SPECInt2006 benchmark suite.

In summary, the project makes three significant contributions to the field.

1. A **fusion library** that lowers the barrier to fusion research in academia, bridging the information gap between academia and industry.
2. A **limit study** that proves the potential of instruction fusion for RISC-V, and encourages further research in this direction.

   In particular, notable results include demonstration of a **50% reduction in effective instruction count** after applying fusion, suggesting a doubling of processor performance in the limit.
3. An example of how **instruction histograms** can be used to drastically speed up the research and development process in computer architecture.

Chapter 2

Preparation

In this chapter, we explore the different design decisions that come into play when creating a fusion library and when performing the limit study.

2.1 Relevant Theory

This section covers the theory that is necessary to understand the project. The novel theory that is developed as part of the project is coloured or in coloured boxes.

2.1.1 Instruction Fusion

Instruction fusion is a microarchitectural technique that is commonly employed in x86 and Arm processors. It works by fusing multiple assembly instructions together before or during the decode stage in a pipeline into a single more complex instruction.

In academic literature, instruction fusion is commonly applied between pairs of adjacent instructions. One example is the indexed load:

```
add    rd,rs1,rs2
ld     rd,0(rd)
```

The first instruction computes the load address, and the second instruction executes the load. By considering this pair as a single instruction in most of the instruction pipeline, we can significantly lower the effective instruction count and therefore increase performance. Power consumption also potentially decreases as we're expending fewer resources to process fewer effective instructions.

Fusion Beyond Common Pairs

Let us consider a series of instructions that form a dynamic instruction trace. A processor takes in instructions from the dynamic instruction trace successively, and it analyses the instruction stream for fusion opportunities.

When the processor is just looking for known pairs of instructions, it simply needs to see if the opcode of the first instruction matches a list of known opcodes and if the opcode of the successive instruction matches as well.

However, when considering instruction fusion beyond common pairs, it's not sufficient to compare the opcodes to a known list. We want a more general framework to describe how a processor can extract fusable blocks, with the only available information being just the current instruction and potentially a few neighbouring instructions.

Therefore, we can model this as the processor applying a **fusion algorithm** on the series of instructions forming the dynamic instruction trace, based on a set of **fusion rules** that specify which instructions were fusable with each other.

To the best of our knowledge, related concepts were not mentioned in existing academic literature, though it is highly likely that industry companies have internally developed similar theory.

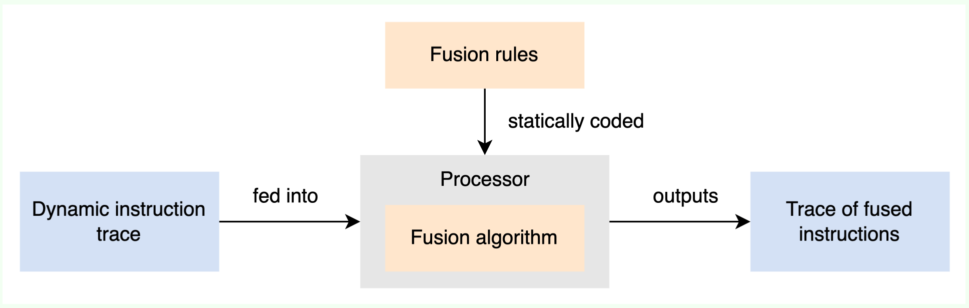


Figure 2.1: The processor runs a fusion algorithm based on a set of fusion rules that the microarchitect has decided to implement. It applies this algorithm on a series of instructions that are being fed into the processor and it outputs a series of fused instructions.

2.1.2 Fusion Metrics

Effective instruction count is the number of instructions actually executed by the processor back-end, after taking micro-op splitting and instruction fusion into account.

Effective cycle count is the number of cycles taken to execute the program, after taking micro-op splitting and instruction fusion into account.

Fusion rate is the percentage reduction in the effective instruction count as a result of fusion.

$$\text{Fusion rate} = \frac{\text{count}_{\text{non-fused}} - \text{count}_{\text{fused}}}{\text{count}_{\text{non-fused}}}$$

2.2 Requirements Analysis

As mentioned in the Introduction and Proposal, the main objectives of the project are:

1. write a **fusion library (HistoSim)** for computer architects and researchers.
2. perform a **limit study** to evaluate the effectiveness of instruction fusion for RISC-V.

A detailed breakdown of these objectives are described in Section 1.2.

Keeping in mind that HistoSim is targeted towards computer architects and researchers, the design goals for HistoSim are as follows:

- **extensibility$^{\mathbf{EX}}$**: HistoSim should be easily extensible to provide further features for computer architecture researchers. A secondary aim is **modularity$^{\mathbf{MO}}$**, as by keeping components separate, HistoSim can be more easily extended to include further functionality.
- **flexibility$^{\mathbf{FL}}$**: HistoSim should be able to cater to the different needs of computer architecture researchers with respect to instruction fusion. It should be able to handle different program traces and different fusion rules.
- **scalability$^{\mathbf{SC}}$**: HistoSim should be able to run multiple experiments at once, on multiple files, in a reasonable amount of time.
- **simplicity$^{\mathbf{SI}}$**: HistoSim should be easy for computer architecture researchers to use and understand.

Throughout the rest of the preparation and implementation section, I will highlight the design decisions made to meet each of these design goals in the respective colour.

The risks associated with each of the main deliverables are shown in Figure 2.2, and a more detailed dependency analysis of each implementation component is shown in Figure 2.3.

Main Deliverables	**Priority**	**Risk**
input data parsing and formatting	high	low
implementing fusion rules	high	low
implementing fusion algorithm	high	low
output data parsing and presentation	high	medium
instruction counts experiments	high	medium
open source as a C++ library	medium	low
replicate previous work (*)	low	medium
implementing further fusion rules (*)	low	low
pipeline experiments (*)	low	high

Figure 2.2: Main deliverables, where (*) indicates an extension

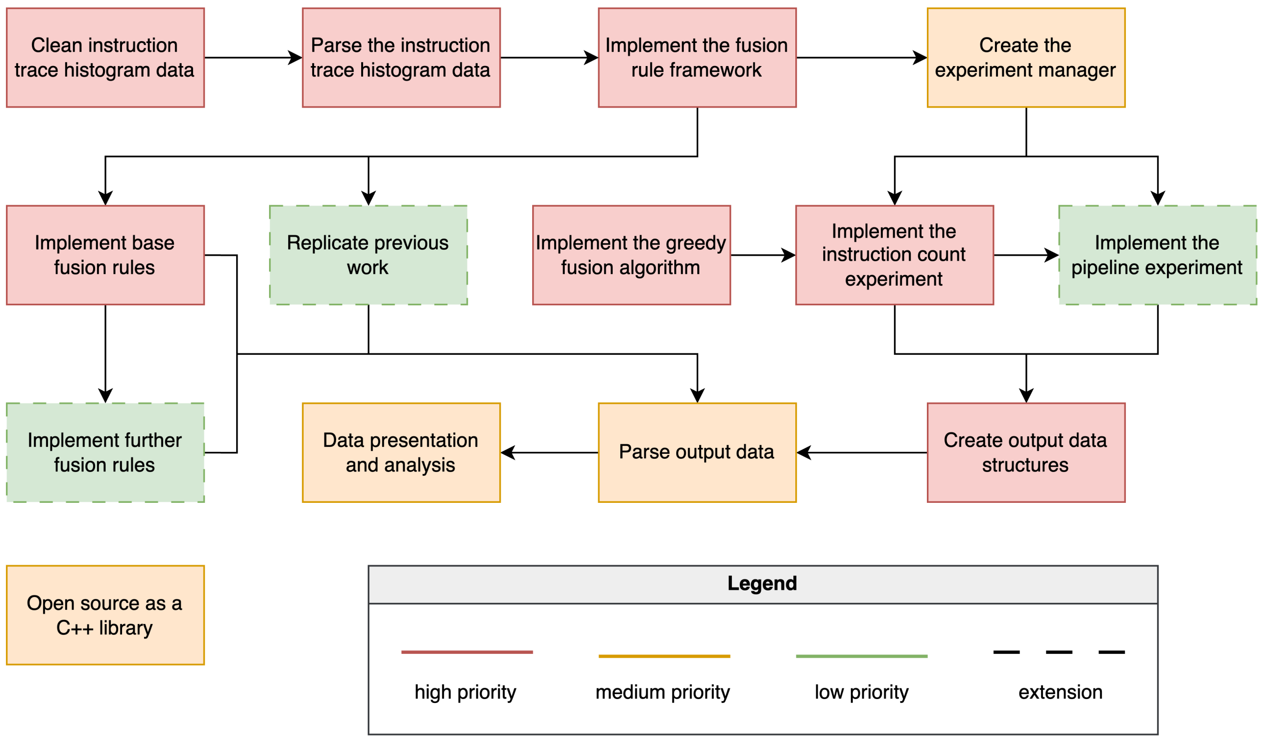

Clean instruction trace histogram data
Parse the instruction trace histogram data
Implement the fusion rule framework
Create the experiment manager
Implement base fusion rules
Replicate previous work
Implement the greedy fusion algorithm
Implement the instruction count experiment
Implement the pipeline experiment
Implement further fusion rules
Data presentation and analysis
Parse output data
Create output data structures
Open source as a C++ library
Legend
high priority
medium priority
low priority
extension

Figure 2.3: Dependency analysis of the implementation components

2.3 Design Decisions

2.3.1 Review of Existing Tools

First, it's important to justify why a separate library needs to be created in the first place. There are a variety of simulation tools that computer architects and researchers use, and could potentially be used to help with this project. However, none of them met my requirements, and I describe why below.

The gem5 Simulator The gem5 simulator [12, 13] is an extensive and widely used tool for computer architecture research. It doesn't currently support instruction fusion out of the box, so I would have to add fusion functionality to the simulator before I can run experiments on it. As gem5 is a very big project, we predicted that modifying it to support fusion would be more difficult than writing our own trace-based simulator from scratch.

ChampSim ChampSim [14] is a trace-based simulator for microarchitecture study. It's primarily used for branch predictor and memory system research. I've decided not to use ChampSim as it is a tool focused on memory system simulation, and is lacking when it comes to pipeline simulation. In particular, it treats all non-memory instructions as the same instruction, which is definitely not what I want as the nature of an operation is important in deciding whether it can be fused or not.

There are more tools that I have considered, but I decided not to use them for various reasons.

2.3.2 Input Data

This section concerns the choice of input data to use: we want the data to be relatively easy to obtain, machine-agnostic^{FL}, and easy to format and work with^{SI}. There are a few alternatives that meet these requirements:

1. **execution-driven simulation**: take in the source programs and execute them directly, using a simulator to measure the number of effective instructions before and after fusion.
2. **instruction traces**: have a full trace of the execution of a program in RISC-V assembly, and analyse the traces for fusion opportunities.
3. **instruction histograms**: a compact summary of an instruction trace that records each instruction executed and how many times they are executed.

Execution programs are not suitable due to the increased complexity. Experiments would take much longer to run compared to the alternatives, and the increased accuracy of the results are not as important for a limit study. We would need a compiler to compile programs in any language or form into RISC-V in addition to a simulator to execute these RISC-V programs which would take extra time to execute^{SC}.

Instruction histograms are better than instruction traces because of their compactness^{SC}. A full program trace can be billions of instructions long, even for a relatively small benchmark program. For example, the `astar` benchmark executes over 50 billion instructions. In contrast, the instruction histogram consists of 25 thousand instructions.

Using instruction histograms give a drastic speedup^{SC} in the time it takes to execute experiments, allowing for rapid prototyping. As we are mainly concerned with the instruction pipeline and not other system components, the information we lose from compressing instruction traces into instruction trace histograms is worth the speedup we gain as a result.

2.3.3 Instruction Fusion Rules

There are a few criteria that the fusion rules need to satisfy:

1. we want to explore fusion beyond pairs^{FL}: the fusion rules should be able to represent arbitrary sequences of instructions.
2. the rules should be expressive^{FL}: we're implementing a library that would be used by computer architecture researchers now and in the future. We cannot predict what future architectures would require of fusion rules, so we need to design our rules to be able to express as many requirements as possible.

Fixed Configuration

Fusion rules can be implemented as parameters to a fixed rule format, specifying:

1. a set of instructions that are fusable with each other
2. an optional set of instructions that a fusable block can end with
3. a series of hard-coded parameters to specify different fusion options. An example parameter is `maxFusableLength`, a positive integer parameter that, if set to n, the maximum length of a block of fusable blocks is n.

This method is the logical extension to current practices of fusing common pairs. It is relatively easy to implement, and the logic for implementing the algorithm is also simple to implement as there aren't many edge cases to consider.

However, there are notable disadvantages to this approach. The number of rules that can be specified using this format are very restricted, and any rule that didn't fit this framework would warrant a change in the library source code$^{\mathrm{FL}}$, which is not ideal for a library that is meant to be as flexible as possible.

For example, a computer architecture researcher would not be able to test out fusion rules that limit the number of register read and write ports without modifying the library source code. This would make the library too rigid to be feasible.

Regular Expressions and Finite State Machines

Another line of thought is to specify a sequence of instructions as a sequence of symbols, a fusion rule as a regular expression matching a sequence of instructions, and a fusion algorithm as finding the maximum number of symbols that are matched using the regular expression.

The advantages of using regular expressions is that the fusion rule can be arbitrarily specified$^{\mathrm{FL}}$ as long as it can be expressed. The algorithm for finding the longest match is also standard, and would be able to deal with any regular expression.

The disadvantages are two-fold:

1. The regular expression can get very complicated very quickly, especially with a large number of possible instructions. This might slow down the algorithm to impractical rates$^{\mathrm{SC}}$.
2. Regular languages are not sufficient in representing the space of all feasible fusion rules$^{\mathrm{FL}}$. For example, it is unable to do counting. This presents fundamental limitations on the kinds of rules a regular expression can express.

This approach is interesting, but it is unsuitable for this project due to the disadvantages mentioned above.

User Defined Functions

This approach is the most suitable one for this project. It asks the user to provide a set of fusion rules by specifying a function for each of them, with the function taking a block of instructions that it has previously said were fusable, and an additional instruction which it evaluates by deciding if that instruction is fusable with the block.

The function would return one of four results:

1. `NOT_FUSABLE`: the instruction is not a fusable instruction following the fusion rules.
2. `END_OF_FUSABLE`: the instruction is the end of the current fusable block.
3. `START_OF_FUSABLE`: the instruction is the start of a new fusable block.
4. `FUSABLE`: the instruction is fusable with the current block.

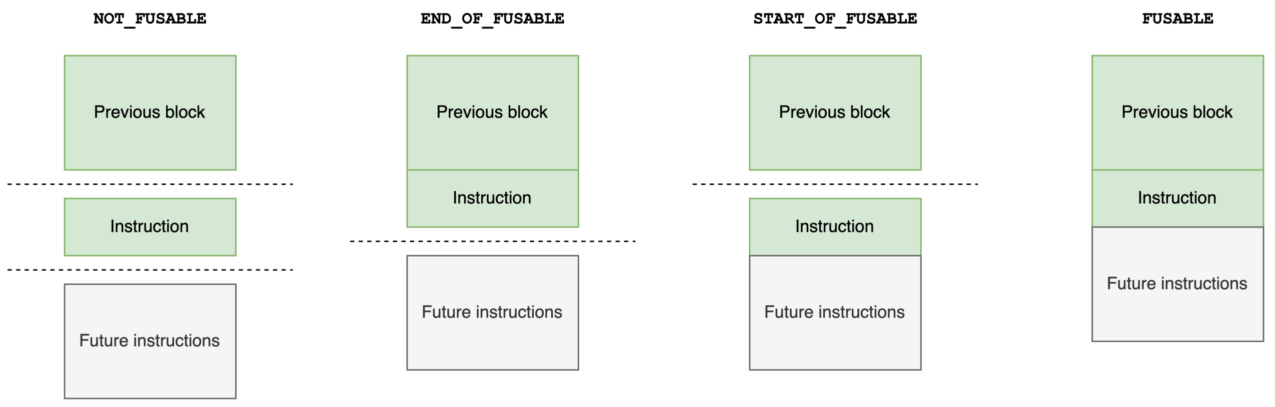


Figure 2.4: A representation of what each fusion result means, where the dashed line signifies a boundary between fusable blocks.

Notably, only `FUSABLE` and `START_OF_FUSABLE` are strictly necessary for the fusion rule to be definable, but `NOT_FUSABLE` and `END_OF_FUSABLE` are convenient and simplify the logic for some of the user-defined functions.

The advantages are that arbitrary fusion rules can be specified$^{\mathrm{FL}}$, as long as we stick to fusing contiguous instructions only. It's also very easy to understand and use$^{\mathrm{SI}}$, as the logic for implementing a fusion rule is simple to implement.

The disadvantages are that we need to implement a custom fusion algorithm to make use of the user-defined functions. The way a fusion rule is specified and the corresponding fusion algorithm are intertwined with each other$^{\mathrm{EX}}$, and a change in the way a fusion rule is specified might require a change in the fusion algorithm as well.

2.3.4 Instruction Fusion Algorithms

One important aspect to consider when implementing a fusion library is to consider how instructions would be picked to form a fusable block.

There are three approaches to solving this problem:

1. **Fusing adjacent pairs**: this is the approach taken by the previous work mentioned in Section 1.1.4. However, as this is a limit study where we want to consider longer sequences of instructions$^{\mathrm{FL}}$, this is not a viable approach to take for this project.

2. **Sliding window**: we can extend the first method to a sliding window of a set size, and consider all possible fusable combinations of instructions within this sliding window. I will elaborate on this approach in Section 2.3.4, but I didn't choose to implement this as performance scaled poorly with instruction window size$^{\mathrm{SC}}$ and I wanted to be able to capture arbitrary fusion lengths before I had enough data to characterise the properties of fusable blocks$^{\mathrm{FL}}$. Other computer architects and researchers might have the same need of characterising the program before adding hardware limits.

3. **Greedy algorithm**: we consider the problem from first principles, model it as a dynamic programming problem, and consider a greedy solution to that dynamic programming problem. This is the approach we use in the project as it is intuitive to understand$^{\mathrm{SI}}$ and gives us the theoretical limit of instruction fusion$^{\mathrm{FL}}$, but it has the limitation in that it is unable to fuse non-contiguous instructions$^{\mathrm{FL}}$.

Sliding Window

The sliding window is a practical way of looking at the problem, and is likely the way a hardware processor would implement instruction fusion. At each point, we have a fixed window of instructions that can potentially be fused together, and we fuse the **largest block that includes the current instruction** (the oldest instruction in the block).

This guarantees progress as we are always fusing the oldest instruction in the block, while attempting to fuse the largest number of instructions possible within that fixed window size. This approach allows us to fuse non-contiguous instructions$^{\mathrm{FL}}$, but the main limitation is that additional fusion opportunities might be missed due to the fixed window size.

For a sliding window of size n, the time complexity of the algorithm is $O(n^2)^{\mathrm{SC}}$, as in the worst case every instruction is not fusable with one another meaning that in every iteration, the algorithm looks n instructions ahead but only moves the window forward by one instruction. Therefore this approach works much better for smaller values of n, which we can only use when we're certain that most of the benefits of instruction fusion are exploited even with short fusable block lengths.

This approach is not suitable if I want to conduct a limit study as part of my project, as the sliding window isn't able to consider blocks of instructions of potentially infinite lengths. However, this approach can definitely be implemented as part of future work, when the range of fusable block lengths are well characterised by the limit study.

Greedy Algorithm

First, we present a model for instruction fusion. Then, we discuss a dynamic programming solution to this problem, and lastly we talk about the greedy approach to this dynamic programming problem that gives an optimal solution.

Modelling Instruction Fusion We model a sequence of n instructions indexed from 1 to n, and a set of fusion rules $A, B, \ldots, N$.

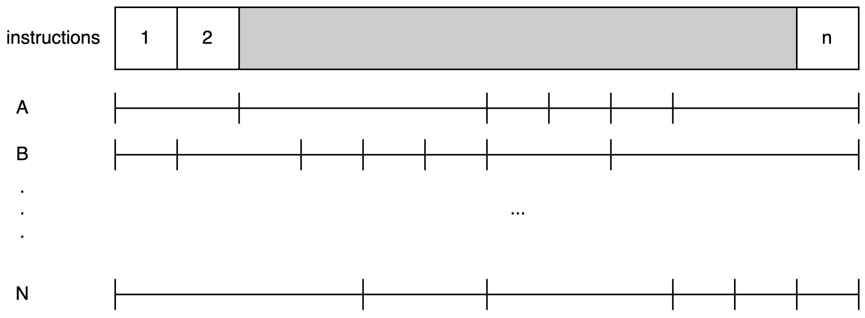


Figure 2.5: An abstract representation of the fusion algorithm.

Each fusion rule operates on the sequence of n instructions, and outputs a series of segments $1, 2, \ldots, k$ which cover the entire sequence. Each segment represents a single fusable block, where the start and end index i, j represent the start and end of the fusable block respectively.

The aim of instruction fusion is to find the minimum number of segments across all the fusion rules that can cover all n instructions.

This model doesn't extend to fusing non-contiguous instructions, as segments assume all instructions within it can be fused into a single instruction. To model non-contiguous fusion, we would need to discard the notion of segments, and have fusion rules output sets of instructions which are fusable with each other. This is more complex than the model above, and is not within the scope for this project.

Dynamic Programming Approach Instruction fusion can be modelled as a dynamic programming problem, with equation:

$$\text{effectiveCount}_j = \min_{i,k}(\text{effectiveCount}_i + \text{count}(i \rightarrow j, k))$$

where i is the start of the segment, j is the end of the segment, and k is the index of the fusion rule. `count` is the number of fused instructions resulting given a segment $i \rightarrow j$ and a fusion rule k. effectiveCount_j is the effective count of all the instructions from index 1 to index j.

The time complexity of the dynamic programming solution would be $O(n^2k)$, where n is the length of the instruction sequence. This is infeasible for all practical purposes, so we need to find a quicker solution.

Greedy Approach We can come up with a greedy algorithm that solves the dynamic programming problem optimally. This is the algorithm that we chose for this project. The greedy criterion is to at each step, **choose the segment that ends last**. The proof of correctness of the greedy algorithm is detailed in Appendix A.

2.4 Software Engineering Tools and Techniques

Development Methodology I have used the **spiral model**, first focusing on implementing core functionalities then targeting extensions one at a time. We targeted the high priority tasks first as shown in Figure 2.3, then worked towards the lower priority tasks. The low risk, high priority tasks were left to be extensions to the project.

Languages and Libraries C++ is the most common language for computer architects and researchers alike when doing simulation work [14]. Therefore writing the fusion library in C++^{SI} made the most sense to lower the barrier to entry as much as possible.

Licensing The project is released as an open-source library^{SI} under the Apache 2.0 license. It allows users to use the software for any purpose, to distribute it, to modify it, and to distribute modified versions of the software under the terms of the license.

Testing Framework Every module implemented had their own suite of unit tests written using the C++ Catch2 testing framework. I create short instruction trace histograms with program properties that I wanted to test, and ran the fusion algorithms on them to test for the expected output.

Version Control The project was version controlled using Git, and each feature was worked on a separate branch then merged into main when it was complete.

Logging The data generated from each test run were saved in their own folder, titled by a timestamp with sub-second granularity. Each test folder included a description of the configuration of the experiments^{SI}, making it easy for the user to keep track of the results. All experiment logs were backed up on Google Drive.

2.5 Starting Point

Concepts Knowledge from Part IB Introduction to Computer Architecture was useful throughout the project. I have read one reference paper on instruction fusion for RISC-V [2] but have not read any other papers. I had no background in research in computer architecture, and I have heard of limit studies before but had no knowledge of how they were actually performed.

Tools and Code I have not started any programming or implementation prior to starting the project. I have had previous experience implementing other projects in C++, including the Catch2 testing framework, the CMake build system and Git version control.

Chapter 3

Implementation

This chapter explores the implementation details, first of HistoSim in Section 3.1, and second of the limit study in Section 3.2. Finally, we give an overview of the project repository structure in Section 3.3.

3.1 Overview of HistoSim

Figure 3.1 shows an overview of the main components of HistoSim. The rest of the section goes into each of these components in detail.

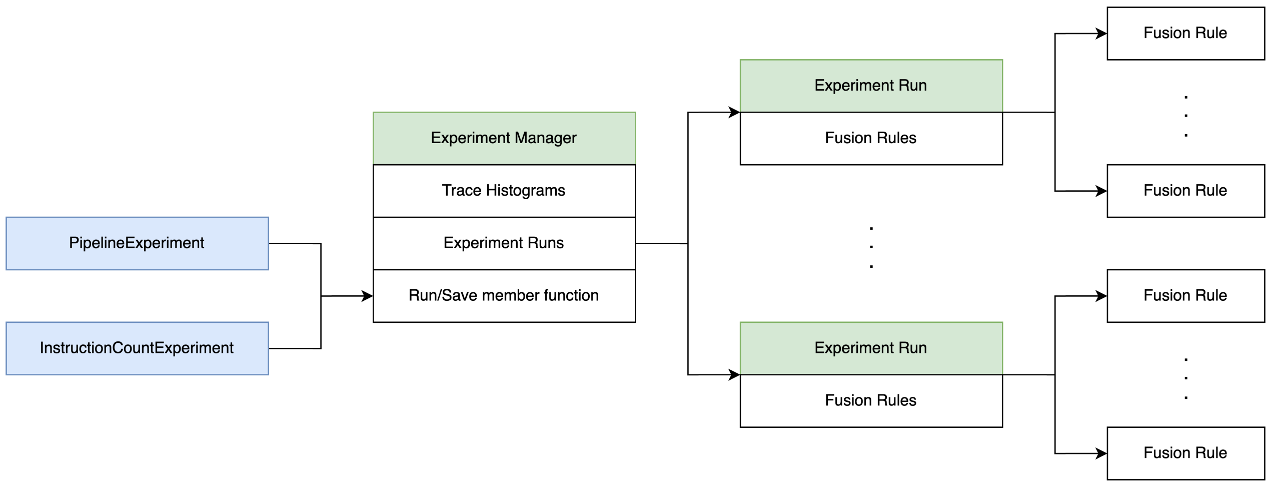


Figure 3.1: Overview of the structure of HistoSim. The blue boxes are the experiment classes that are passed in as a template argument to the run and save member functions of the Experiment Manager.

3.1.1 Experiment Manager

The experiment manager can be thought of as an **control centre** that stores the common data and runs different experiments based on that data.

The experiment manager is able to handle a wide variety of inputs[FL], including those that differ in the types of output data as well as types of experiments. It does this using **templating**, a core feature in C++. An example of using templating can be seen in the `run` function of the experiment manager, which is as follows:

```
template<class Experiment, typename Result, typename... Args>
ExperimentResults<Result> run(Args... args) {
    Experiment experiment(this->shared_from_this());
    auto results = experiment.run(std::forward<Args>(args)...);
    return results;
}
```

The function takes in any experiment class which has a `run` function, and calls that run function using the arguments that were passed in. The `run` function of the experiment itself can take in a variable number of arguments, which increases flexibility while providing a standard interface to the user.

The experiment manager stores all the input data that is necessary when it is constructed, meaning that multiple experiments can share the same copy of the input data, saving memory. This is important as trace histograms can be very large, and we don't want to have to duplicate them across different experiments.

3.1.2 Input Data

Out of the various approaches mentioned in Section 2.3.2, the input data format that I settled on for this project is the **instruction histogram**. The instruction histograms I'm working with are a list of instructions with each instruction consisting of the following information:

1. **address**: the address of the instruction located in memory. The absolute address doesn't matter, but it is important that addresses are ordered relative to each other. This is so that we can make the assumption that the instruction in the next address will get executed next in the absence of a change in control flow.

2. **count**: the number of times the instruction was executed over the course of the program. This is where the name "histogram" comes from, as it significantly compacts the information compared to a typical program trace.

3. **opcode**: denotes the type of RISC-V instruction, examples are `add` and `ld`. This is important as some fusion rules are specified relative to the opcodes (e.g. `add` instructions might be fusable with each other but not `ld` instructions).

4. **operands**: these are the registers that the instruction will operate on. This is important as some fusion rules are specified relative to the operands (e.g. independent instructions are fusable but not dependent instructions).

The list of instructions do not have to be listed in address order.

The user is in charge of formatting their data to meet the requirements above. For my own experiments, I used python and the `pandas` library to combine a dump of the assembly instructions and counts for each instruction executed into the format above.

3.1.3 Fusion Rules

This subsection explains the framework for constructing fusion rules. The actual fusion rules that I implemented for the experiments can be found in Section 3.2.1.

A fusion rule takes in a block of instructions previously deemed to be fusable and one additional instruction. It determines whether the instruction is fusable with the existing block. As mentioned in Section 2.3.3, it outputs one of four possible outcomes: `NOT_FUSABLE`, `END_OF_FUSABLE`, `START_OF_FUSABLE` and `FUSABLE`. Additionally, the fusion rules I've implemented are chainable with each other[SI], so that rules can be composed with each other without having to be redefined.

In my implementation, a fusion rule is a wrapper around a user-defined function. It is implemented as a C++ class that has a `std::function` as a member variable, and `chain` and `apply` member functions.

The member function, initialised at construction, is the user-defined function that is passed in to the library. It has the type

```
std::function<FusableResult(std::vector<std::shared_ptr<Instr>> const&,
 ↪  Instr const&)>
```

Figure 3.2: Type signature of the user-defined function that is passed in to the library and stored as a member variable.

The `chain` member function allows a fusion rule to be defined with respective to two other fusion rules, and it is interpreted as one rule being applied after another. Implementation-wise, it applies both fusion rules and returns the worst result, where the following outcomes are ordered from best to worst:

1. `FUSABLE`
2. `END_OF_FUSABLE`
3. `START_OF_FUSABLE`
4. `NOT_FUSABLE`

It can logically be thought of as an "AND" operator which combines two fusion rules by outputting the worst result of both fusion rules, for example a fusable segment and a non-fusable segment chained together would be non-fusable. This allows for more complex fusion rules to be constructed as the composition of multiple simpler fusion rules. Examples can be seen in Section 3.2.1.

The `apply` member function takes in a block of instructions and the current instruction, and applies these arguments into the user-defined function.

3.1.4 Fusion Algorithm

The fusion algorithm is technically implemented as part of the instruction count experiment, explained in Section 3.1.7, but as both experiments rely on having fused instructions using this fusion algorithm, I will explain it here. The theory behind the algorithm was explained in Section 2.3.4. In this section, I outline the implementation of this algorithm.

The algorithm takes in a program and a set of fusion rules, corresponding to the sequence of instructions and the set of segments mentioned in Section 2.3.4 respectively. The pseudocode for the algorithm is shown in Figure 3.3.

It iterates over the sequence of instructions, and for each instruction it iterates over each fusion rule. It eliminates the fusion rules that get the worst results, and the round ends when there are no fusion rules left. The fusion rule which was eliminated last gets chosen, and a new round starts with all the fusion rules in contention again.

3.1.5 Output Data

The output data is automatically saved into a user-configured directory, with the folder name as a timestamp in the `YYYY_MM_DD_HH-mm-ss` format. This improves user experience, as experiments can be run in quick succession while keeping it clear which results belong to which experiment runs.

The specific output data that is saved depends on the experiment that is being run.

3.1.6 Testing and Debugging

Testing is a very important part of the project, as it is crucial to guarantee that the fusion behaviour is correct so that users are able to get correct results.

Throughout development, I made sure to write unit tests using the C++ **Catch2** test framework. Unit tests were written for all major components of the project. Functionalities tested include:

- testing that the fusion algorithm behaves as expected, along different parameters and fusion rules
- testing that multiple fusion rules can be specified within the same experiment run, and the output of the program is as expected

In addition, as part of the experiment workload, I checked for uncategorised instructions. Users would be able to run a script to check for uncategorised instructions as well, so that the fusion algorithm would be able to categorise all the relevant instructions to maximise fusion opportunity.

```
FusionResults FusionCalculator::calculateFusion(
    shared_ptr<File> file,
    ExperimentRun const& run)
{
    auto rules = run.rules;
    for (auto const& instruction : file-> instructions) {
        if (startOfBlock) {
            // record the instruction count as the count of
            // the fusable block
            dynamicCount = instruction.count;
        }

        for (auto fun : rules) {
            auto result = fun->apply(currBlock, instruction);
            switch (result) {
                case FUSABLE:
                    break;
                case END_OF_FUSABLE:
                    rules.erase(fun);
                    endOfFusable = true;
                    break;
                case START_OF_FUSABLE:
                    rules.erase(fun);
                    startOfFusable = true;
                    break;
                case NOT_FUSABLE:
                    rules.erase(fun);
                    break;
            }
        }

        if (rules.empty()) {
            if (endOfFusable) {
                currBlock.push_back(instruction);
                startNewRound();
            } else if (startOfFusable) {
                startNewRound();
                currBlock.push_back(instruction);
            } else { // NOT_FUSABLE
                startNewRound();
                currBlock.push_back(instruction);
                startNewRound();
            }
        } else {
            currBlock.push_back(instruction);
        }
    }

    return fusionResults();
}
```

Figure 3.3: Pseudocode for the fusion algorithm

3.1.7 Experiment Types

In the following section, I explain the 2 types of experiments that I have constructed in HistoSim. These experiments are available for use by the computer architecture researcher on their trace data, but they can also create new experiment types by fitting into the experiment manager framework^{MO}.

Instruction Count Experiment

An instruction count experiment is one that takes in a list of trace histograms and a set of fusion rules, and outputs the number of effective instructions before and after fusion (applying the set of fusion rules on each trace histogram).

It is useful for getting an initial idea of the effectiveness of certain fusion rules, not taking into account hardware limitations or pipeline utilisation. It is also a useful first step to output fused blocks for future experiments.

For every trace histogram and set of fusion rules that are passed into the experiment as input parameters, the experiment runs the fusion algorithm detailed in Section 3.1.4 to obtain the effective instruction count after fusion, applying the set of fusion rules on each histogram successively. It then collects the results and saves it to a CSV file in a directory specified by the user, and the user can parse this CSV file to analyse the results.

The instruction count experiment outputs the fusion results as 3 separate CSVs, of different granularities. This is to aid the user to more easily parse the data for further analysis. These CSVs are:

1. **Aggregate**: fusion results averaged across all benchmarks
2. **Per Benchmark**: fusion results that are displayed per benchmark
3. **Fusion Lengths**: fusion results that show the specific fused blocks

Theoretically, all the data can be derived from the fusion lengths CSV, but the aggregate and per benchmark data is for easier parsing. The fusion lengths CSV can be very large, so the other two CSVs can be used for most purposes that doesn't require per fused block granularity. The detailed structure of each CSV can be found in Appendix B.1.

Pipeline Experiment

A pipeline experiment is one that takes in a list of trace histograms and a set of fusion rules, and outputs the number of cycles before and after fusion. It differs from the instruction count experiment in that it takes pipeline stalls into account, so it provides a more realistic view of the effectiveness of instruction fusion.

The reason why pipeline stalls is interesting is because fusion can potentially increase the number of stalls in the pipeline. By fusing multiple instructions together and executing them within the same pipeline stage, a stall could be introduced compared to the non-fused version. For example, given the following sequence of instructions:

```
ld     rs1, 0(rs1)
addi    rs2,rs2,#1
add     rd,rs1,rs2
```

Originally, there would have been no pipeline stalls in this instruction sequence, as the `add` instruction that uses the result of the `ld` instruction, the value in `rs1`, is 2 instructions after the load instruction. However, if the last 2 instructions were fused together, then the fused sequence of instructions would become

```
ld     rs1, 0(rs1)
fused add instruction using rs1 and rs2
```

which would require a pipeline stall, as the value of `rs1` is not available at the point where the fused add instruction is executed. As seen in this example, a fused sequence of instructions wouldn't have lower performance compared to the non-fused sequence of instructions, but the performance benefit of fusion could decrease due to the increased number of pipeline stalls.

The pipeline experiment optionally is able to take in the results of the instruction count experiment before it is run. This allows it to access the fused blocks directly without having to rerun the fusion algorithm. If no results are passed in, the pipeline experiment automatically calls the instruction count experiment first before computing the cycle counts.

The experiment is structured as a general pipeline experiment that takes in a pipeline object and computes the cycle count for that pipeline^{EX}. In my project, I've opted to implement the most simple in-order pipeline where the only pipeline stalls are load-to-use stalls, and they stall the pipeline for one cycle. All other potential stalls are assumed to be circumvented using data forwarding.

The pipeline experiment then looks into adjacent fused blocks and inspects them for possible pipeline stalls. This approach is valid given 2 assumptions:

1. the only pipeline stalls are load-to-use pipeline stalls of a single cycle
2. branch instructions are not fused with load instructions

Given these assumptions, only pipeline stalls between consecutive fused blocks in the address space would exist. We don't need to take into account stalls between a branch instruction and its branch target, as there is always a branch instruction in between a load fused block before the branch and a use fused block after the branch.

The pseudocode to compute cycle count is simple:

```
    for (auto const& block : blocks) {
        // test for shared operands in the block
        if (test_for_operands(block)) {
            results.stalls += block.count;
        }

        // add load operands for the next block to test from
        loadOperands.clear();
        find_ld_operands(block);
    }
```

Figure 3.4: Pseudocode for computing cycle count given a one cycle load-to-use pipeline stall, for an in-order pipeline.

The algorithm iterates through all the fused blocks, output as a result as the instruction count experiment, and sees if it contains any operands that are the result of a load operation in the previous fused block. If it does, then we take the pipeline stall into account when calculating the number of cycles.

The pipeline experiment outputs the fusion results as 2 separate CSVs, of different granularities. The CSVs are:

1. **Aggregate**: cycle counts averaged across all benchmarks
2. **Per Benchmark**: cycle counts that are displayed per benchmark

The structure of each output CSV in the pipeline experiment can be found in Appendix B.2.

3.1.8 Use of the Library

The user first installs HistoSim by running the command `sudo cmake install` on a cloned version of the fusion library. The user then links to the library in their own project. On CMake, this involves adding the line `find_package(fusion REQUIRED)` in the `CMakeLists.txt` file at the base of the project.

Using HistoSim consists of creating an `ExperimentRun` consisting of sets of fusion rules and a list of files that the user wants to perform the experiments on.

I created a series of example fusion rules within the library that the user could use for reference and also for their own use. Some of these rules are also rules that I have used in my own experiments, which I will explain in Section 3.2.1. They consist of the 4 base rules mentioned in Section 3.2.1, as well as 5 more rules that are meant to be chained with other fusion rules to limit them further. These consist of:

1. `Independent`: the instructions in a fused block must be independent, i.e. they must not share any operands.

2. `SameCount`: each instruction in a fused block must be executed the exact same number of times.
3. `MaxLength`: this rule takes an integer parameter n, and a fused block must not consist of more than n instructions.
4. `MaxReadPorts`: this rule takes an integer parameter n, and a fused block must not use more than n register read ports.
5. `MaxWritePorts`: this rule takes an integer paramter n, and a fused block must not use more than n register write ports.

3.2 Limit Study

In this section, I detail the implementation details relating to the limit study I ran to evaluate the effectiveness of instruction fusion for a RISC-V architecture.

3.2.1 Running Experiments

In this section, I detail the specific experiments that I ran in order to evaluate the effectiveness of instruction fusion for RISC-V.

For this project, I ran 2 sets of experiments:

1. Experiments that replicate the fusion pairs mentioned Celio et al.'s 2016 paper on instruction fusion for RISC-V. This allows to validate the correctness of HistoSim by comparing it to external results.
2. Experiments using the 4 base fusion rules mentioned in Section 3.2.1, and additional modifier rules that are applied to the base rules to mimic hardware limitations.

Celio et al.'s 2016 Paper

In addition to these 2 classes of fusion rules, I also replicated the results of Celio et al [2] to validate HistoSim.

The fusion rules in this paper are pairwise or three-wise fusion rules that take into account the most common fusable pairs of instructions. They consist of the following fusion rules:

- **Load Effective Address (LEA)**: this matches the instruction pattern of

```
slli    rd, rs1, {1,2,3}
add     rd, rd, rs2
```

  where `slli` and `add` can alternatively be any of their variants as well. The curly braces {} mean that the value of the second operand can match any of $1, 2$ or 3.

- **Indexed Load**: this matches the instruction pattern of

```
add    rd, rs1, rs2
ld     rd, 0(rd)
```

- **Fused Indexed Load**: this combines LEA and Indexed Load to form a three-instruction fusable block.
- **Clear Upper Word**: this matches the instruction pattern of

```
slli    rd, rs1, 32
srli    rd, rd, {29, 30, 31, 32}
```

- **Load Global**: this matches the instruction pattern of

```
auipcc    ca4, 321
clc     ca4, 882(ca4)
```

The fusion rules detailed above are paired with the `SameCount` fusion rule, which matches the limitations in the original paper.

The limited combinations of instructions and the short length of fused blocks mean that the fusion opportunity available is much lower than the fusion rules we propose in this project. Nonetheless, it is a good method to validate the results of our fusion algorithm.

Custom Fusion Rules

There are four main base rules that I experimented with in my project:

1. `ArithmeticOnly`: a sequence of arbitrary arithmetic instructions
2. `ArithmeticEndBranch`: a sequence of arbitrary arithmetic instructions ending in a branch instruction
3. `ArithmeticEndMemory`: a sequence of arbitrary arithmetic instructions ending in a memory instruction
4. `ArithmeticEndMemoryOrBranch`: a sequence of arbitrary arithmetic instructions ending in a memory or branch instruction

These base rules are chained with other fusion rules like `Independent` or `MaxLength` to limit the fusion to independent instructions and to a maximum length respectively.

The reason these rules were chosen is because of the high fusion potential of arithmetic instructions. Arithmetic instructions cannot throw an exception, making them prime candidates for instruction fusion as they can be logically combined into large instructions that are executed atomically without violating the architecture.

They also demonstrate the effectiveness of fusing more general, longer sequences of instructions compared to only fusing frequently occurring pairs or triplets of instructions, while remaining in the realm that is realistic for a computer architect to want in their processor. More drastic fusion rules can be explored as part of future work, to gain even more benefit out of instruction fusion, but I deemed this to be sufficient to as a proof of concept and potential, to encourage more work in this field.

The rule that encompasses the other rules is `ArithmeticEndMemoryOrBranch`, which fuses a sequence of arithemtic instructions of any length, optionally ending in either a memory or branch instruction.

Next, I list the sets of rules that I experimented with as well as the motivation behind each of them.

i) Base + SameCount + MaxLength Base rules chained with the `SameCount` and `MaxLength` fusion rules. Following the example of the `ArithmeticEndMemoryOrBranch` base rule, for a maximum length of n, the algorithm fuses a sequence of up to $n - 1$ arithmetic instructions followed by a single memory or branch instruction, all of which are executed the exact same amount of times. The experiment is repeated with various values of n to gain insight on how fusion rate varies with the maximum length of a fusable block.

This set of experiments gives us an idea of what a processor might be able to achieve in the limit given an intuitive fusion rule of fusing mostly arithmetic instructions, with the correctness properties being enforced with the same count fusion rule. This gives the main results for the effectiveness of instruction fusion for a RISC-V architecture.

ii) Base + SameCount + MaxLength + Independent Same rules as (i), additionally chained with the `Independent` fusion rule so that all instructions in a fused block are independent of each other, in addition to the `SameCount` and `MaxLength` requirements.

These rules are meant to simulate the effectiveness of a traditional superscalar pipeline. A superscalar pipeline can execute multiple instructions within the same pipeline stage as long as they are independent, i.e. the operands of one instruction doesn't depend on the results of another.

iii) Base + SameCount + MaxReadPorts Same rules as (i), but chained with `MaxReadPorts` instead of `MaxLength`. This allows us to gain insight on how many register read ports fusable blocks typically require, which is important because register ports are very costly to add within a processor.

iv) Base + SameCount + MaxWritePorts Same rules as (iii), but chained with `MaxWritePorts` instead of `MaxLength`. The reasoning for implementing this set of experiments is similar to that of (iii).

Base + SameCount + MaxReadPorts + MaxWritePorts Combining the rules of (iii) and (iv). This allows us to take into account the interaction between limiting the number of read ports and the number of write ports, as limiting one might cause the other to also naturally be limited.

3.2.2 Data Analysis

For analysing the output data, I opted to use a series of Python notebooks to plot the CSV that was output as a result of the fusion experiments. This was deemed to be sufficient as the user is expected to conduct their own data analysis, so this part of the project is not part of HistoSim itself.

I wanted to analyse the fusion results in three different granularities:

1. **Aggregate**: fusion results aggregated across the entire benchmark suite. This allows us to evaluate the overall effectiveness of instruction fusion for a general purpose RISC-V architecture.
2. **Per-benchmark**: we can analyse the effectiveness of instruction fusion for specific benchmarks, which might give us some insight on the types of workloads that instruction fusion might be more beneficial for.
3. **Fused blocks**: we analyse patterns within specific fused blocks to gain an insight on the trends in fusion. This is specific to the instruction count experiment.

Most of the work involved reading the CSVs and constructing graphs using `matplotlib` to visualise the results.

3.3 Repository Overview

The code repository is split into the HistoSim fusion library and the user code that runs the experiments, and the fusion library is further split into 3 parts:

1. **common**: common functionality and data structures between the two experiments that were implemented.
2. **core**: the instruction count experiment. This part is named core as the pipeline experiment also relies on first running the instruction count experiment to get the fused blocks.
3. **simulation**: the pipeline experiment, which computes cycle counts.

Following the convention of C++ project structures, each module in the library is split into an **include** directory for header files, a **src** directory for source files, as well as a **tests** folder for unit tests. As I've used CMake for the project, each directory also has a `CMakeLists.txt` file.

The full directory structure is listed below. For clarity, I don't include the `CMakeLists.txt` files in each directory.

```
instruction-fusion
├── user-code ............................ contains user code for running the expriments
│   ├── input-data ....... contains trace histograms for the SPECInt2006 benchmark suite.
│   ├── output-data ................................... contains HistoSim output CSVs.
│   ├── rules
│   │   ├── celio-2016.cxx ................ contains fusion rules replicating previous work.
│   │   └── my-own-rules.cxx ................... contains fusion rules for the limit study.
│   ├── main.cxx
│   └── python-data-analysis .. contains Python notebooks that analyse HistoSim output.
└── fusion-library .............................. contains the HistoSim library itself.
    ├── common ...................... contains code common to multiple experiment types.
    │   ├── include
    │   │   ├── csv-handler.h
    │   │   ├── data-representation.h
    │   │   ├── experiment.h
    │   │   ├── fusion.h
    │   │   ├── instructions.h
    │   │   └── macros.h
    │   ├── src
    │   │   ├── csv-handler.cxx
    │   │   ├── data-representation.cxx
    │   │   └── experiment.cxx
    │   └── tests ...................................... contains test data and unit tests.
    ├── core .............................. contains instruction count experiment code.
    │   ├── include
    │   │   ├── example-rules.h
    │   │   ├── fusion-calculator.h
    │   │   └── instruction-count-experiment.h
    │   ├── src
    │   │   ├── fusion-calculator.cxx
    │   │   └── instruction-count-experiment.cxx
    │   └── tests
    └── simulation ............................... contains pipeline experiment code.
        ├── include
        │   ├── in-order-pipeline.h
        │   └── pipeline-experiment.h
        ├── src
        │   ├── in-order-pipeline.cxx
        │   └── pipeline-experiment.cxx
        └── tests
```

Chapter 4

Evaluation

In this chapter I detail the results that I obtained from running the fusion experiments using HistoSim. We also prove the benefits of using instruction histograms by evaluating the performance of HistoSim compared to other existing tools. These results were presented at the RISC-V Summit Europe 2025 [1].

4.1 Experimental Setup

In the next sections, I will describe the set of experiments that I ran using HistoSim. They provide novel results and insights that are useful to the community, but they are also examples of how a computer architect or researcher might use the library to run experiments and analyse the output data for meaningful results.

As mentioned in Section 3.2.1, there are two main sets of experiments that I ran:

1. The pairwise fusion rules detailed in Celio et al.'s paper [2].
2. The four base fusion rules and various additional constraints.

4.2 Preliminary Analysis

Using `ArithmeticEndMemoryOrBranch` as an example rule, I ran an instruction count experiment with no limit on the maximum fusable length, and observed the lengths of the fusable blocks that resulted from the experiment.

Figure 4.1 below plots the dynamic count of fusable blocks against their length. The maximum length of a fusable block is 143 instructions long, but the fusion results are dominated by blocks with a relatively short length. Fusable blocks that are over 10 instructions long only consist of 0.07% of the total instruction count, which is why I've decided to focus on fusable blocks that consist of 10 instructions or fewer in future experiments.

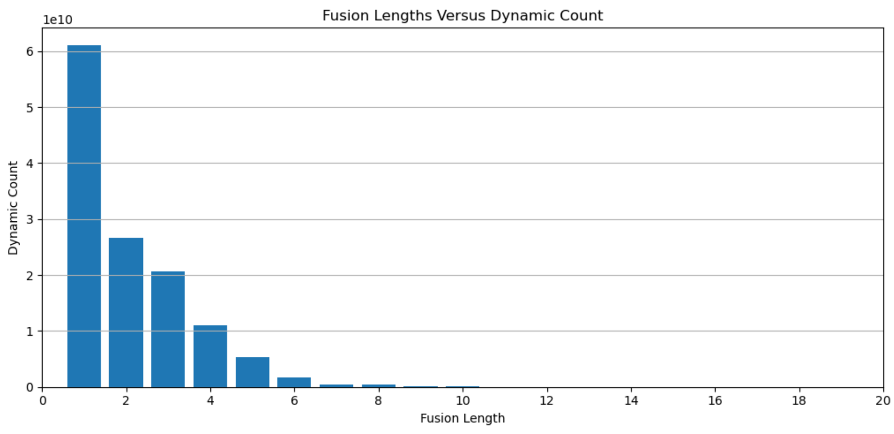
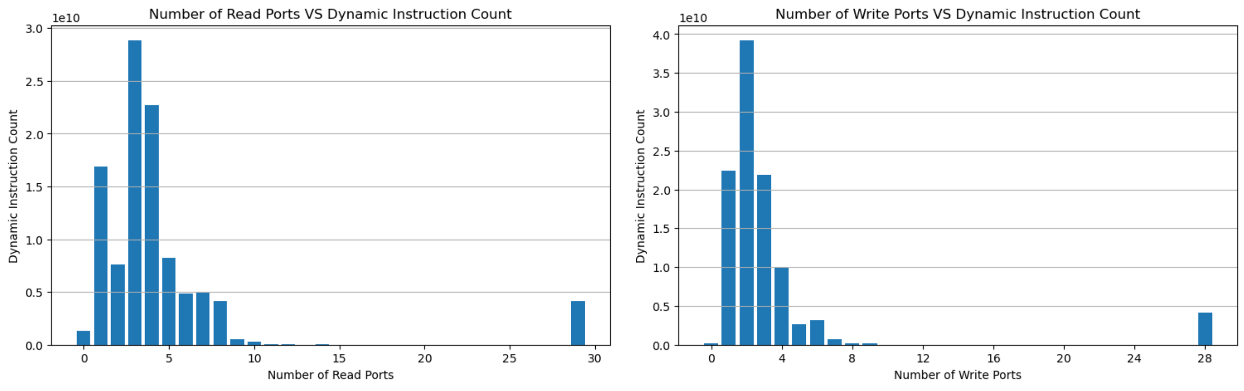


Figure 4.1: The lengths of the fusable blocks using a fusion rule that had no maximum length limit. The vast majority of the fusable blocks have a relatively short length.

Similarly, Figure 4.2 below plots the dynamic count of fusable blocks against the number of register read and write ports required. The maximum number of read ports required is 29 and the maximum number of write ports required is 28, but this is entirely due to the `h264ref` benchmark that makes heavy use of `memcpy`. Considering the other benchmarks, the vast majority of the fusable blocks make use of fewer than 10 register read and write ports. In addition, extra register ports are costly, and it is currently impractical to expect a processor to have more than 10 register ports per pipeline. This is why I deemed it sufficient to limit the register port experiments to a maximum of 10.



(a) The number of fusable blocks plotted against the number of register read ports.

(b) The number of fusable blocks plotted against the number of register write ports.

Figure 4.2: The number of fusable blocks plotted against the number of register ports they used. The majority of the blocks use less than 10 ports.

4.3 Replicating Celio et al.'s Paper

The original paper produces a result of a fusion rate of 5.4%. However, they also mentioned that 57% of the fusion pairs were found manually, as the compiled instructions were not necessarily optimised to maximise fusion opportunities. Therefore, we can infer that an automatic process had a **fusion rate of 2.3%**.

Our trace histograms differ from Celio et al.'s in a few key aspects which might impact our results:

- **different compiler**: our traces were compiled using LLVM instead of GCC. This means that the number of fusion opportunities might vary. In addition, as their paper was from 2016, compiler optimisations would have improved since then, so we might expect to see an increase in fusion opportunities.
- **CHERI** [15]: our traces are traces of running the SPECInt2006 benchmark on CHERI RISC-V, as these were the most readily available form of traces that we were able to obtain. As a result, some of the instructions were CHERI specific.

  However, we don't anticipate a large change compared to the original due to this reason. This is because there is usually a one-to-one correspondence between the CHERI and non-CHERI instructions that achieve the same effect, and they are usually instructions of the same category and therefore would be affected by fusion rules in the same manner.
- **linking**: the benchmarks they used were statically linked during compilation, while the traces for our project were obtained by running benchmarks that were positionally independent, which is closer to dynamic linking. Therefore we might expect to have more fusion opportunities in our version of the code, as there are more address lookups.

Based on the reasons above, we might expect our results to lie somewhere in between their automatically compiled fusion rate and their manually found fusion rate.

From Figure 4.3 below, we can see that the fusion rate of our replicated experiment across the benchmarks is **2.80%**, which is within the expected range of 2.3% and 5.4%. This gives us confidence that the fusion library works as expected, and it also gives us more confidence in our later results when we experiment with more general fusion rules.

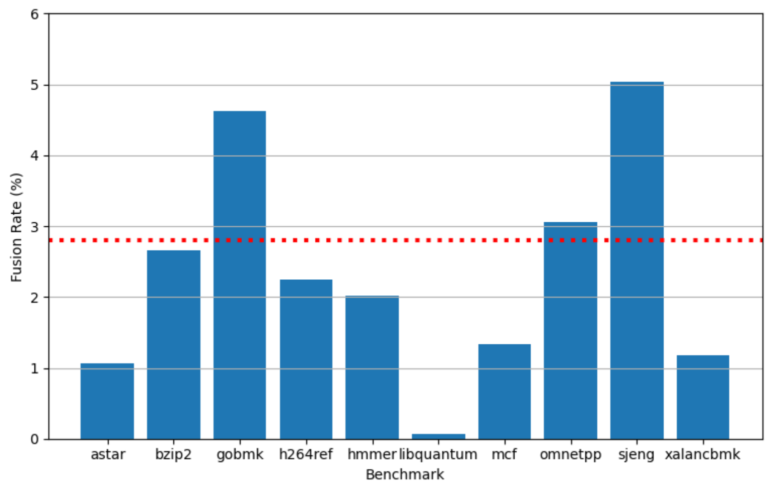


Figure 4.3: Fusion rate obtained from replicating the pairwise fusion rules mentioned in Celio et al.'s paper [2].

4.3.1 How Common are the Common Pairs?

One interesting thing to know is how dominant the common pairs of instructions explored in this paper are compared to all possible pairs of fusion rules with 2 register read ports and 1 register write port.

The fusion rate obtained with the fusion rule `MaxCount(2) + MaxReadPorts(2) + MaxWritePorts(1)` gives a fusion rate of **3.02%**. Fusing common pairs only leaves 0.22% of benefit on the table, which suggests that it is an effective and practical method for applying fusion when we're fusing pairs of instructions at a time with a limited number of register read and write ports.

4.4 Custom Fusion Rule Results

As mentioned before, the four fusion rules that were used for this experiment were:

1. **arithmetic only**: an arbitrary sequence of arithmetic instructions
2. **arithmetic end memory**: an arbitrary sequence of arithmetic instructions, optionally ending in a memory instruction
3. **arithmetic end branch**: an arbitrary sequence of arithmetic instructions, optionally ending in a branch instruction
4. **arithmetic end memory or branch**: an arbitrary sequence of arithmetic instructions, optionally ending in a memory or branch instruction

More detailed explanations for these rules as well as why I chose them can be found in Section 3.2.1.

For each of these fusion rules, I measure four main metrics:

1. **fusion rate**, which I defined to be the percentage reduction in effective instruction count.
2. **percentage reduction in cycle count**.
3. the variation in **fusion lengths** across fused blocks.
4. the variation in the number of **register ports** across fused blocks.

The results for all four base rules are summarised in Table 4.1 below. We can see that they all outperform the previous work done on instruction fusion in RISC-V by a large margin.

Taking pipeline stalls into account decreases the calculated cycle benefit of fusion, but the results are still significantly positive. It is likely that this result would change depending on the assumptions made in the pipeline simulation. We assumed a simple case of a single cycle load-to-use stall, but the number and lengths of stalls differ greatly between different processor designs.

We hope that this work puts draws focus to instruction fusion as a powerful tool to decrease the effective instruction count, and encourage both computer architects and researchers alike to model fusion more closely in more sophisticated simulators or hardware for RISC-V architectures.

Fusion Rule	**Fusion Rate (%)**	**Cycle Count Reduction (%)**
arithmetic only	29.2	25.1
arithmetic end memory	45.4	40.4
arithmetic end branch	36.6	31.0
arithmetic end memory or branch	52.9	46.2

Table 4.1: Fusion rate and cycle count reduction by implementing each of the 4 base rules on the SPECInt2006 benchmark suite.

In particular, `ArithmeticEndMemoryOrBranch` subsumes all of the other fusion rules and therefore gives the best results. We will focus on this outcome in our analysis.

As seen from the table, the potential upside of instruction fusion is very high, with a fusion rate of over 50% being achieved in the limit, meaning that the resulting effective instruction count is less than half of the total number of instructions.

4.4.1 Fusion Rate Versus Maximum Fusable Length

From our preliminary analysis in Section 4.2, we can see that there is a very long tail when it comes to the length of fusable blocks, but the vast majority of fusable blocks have are comprised of fewer than 10 instructions.

This experiment computes the fusion rate against the **maximum fusable length**, which is the maximum allowable number of instructions that form a single fusable block.

As seen from Figure 4.4 below, using pairs alone gives close to 30% fusion rate, suggesting that instruction fusion might be able to provide a big benefit without adding substantial amounts of complexity. The main difference between this and the work done by Celio et al. [2] is that this approach isn't restricted to specific types of instructions like an add or shift left, and instead applies to all arithmetic instructions, potentially also ending in a memory or branch instruction.

In addition, we can see a diminishing return in the fusion rate when the maximum fusable length increases, with the vast majority of the benefit being reaped by the time the maximum fusable length gets to 5 or 6. It is up to the computer architect what maximum fusable length to support, but this is a good reference, and shows that a very limited fusable length can still provide significant benefit.

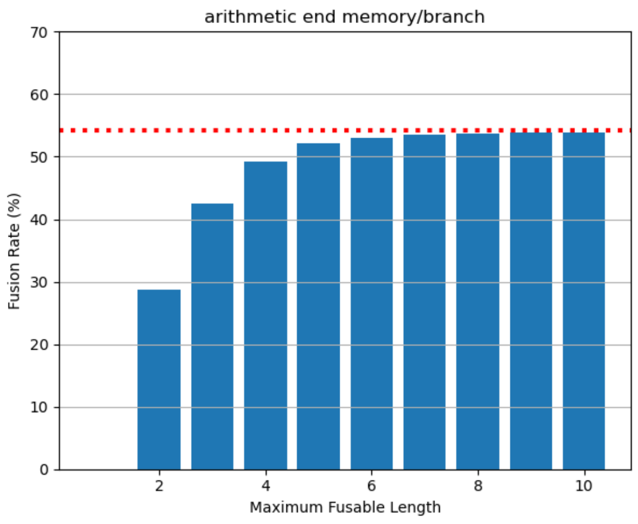


Figure 4.4: Fusion rate versus maximum fusable length for the `ArithmeticEndMemoryOrBranch` fusion rule. The dotted red line shows the fusion rate at the limit.

4.4.2 Fusion Rate Versus Number of Register Ports

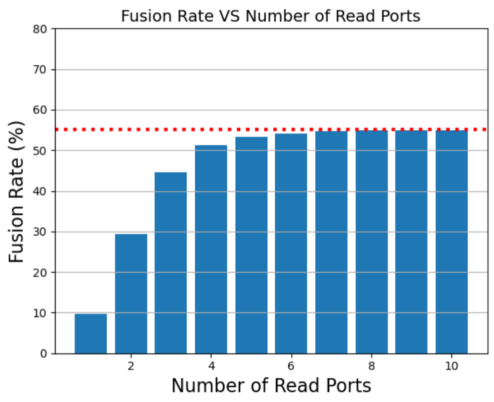
The number of register ports that a processor needs to support is also an important consideration for a microarchitect. This is because adding a single register port is costly both financially and in terms of area, so the decision on the number of register ports must be carefully made.

Instruction fusion in a hardware processor is limited by the number of register ports on the processor, and so it is worth knowing what the fusion rate is when we vary the number of read and write register ports. The results are shown in Figures 4.5a and 4.5b for limiting the number of read and write register ports respectively.

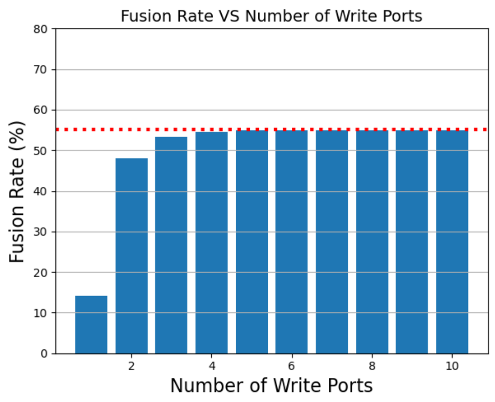
Compared to Figure 4.4, Figure 4.5a shown below has a much smoother increase in fusion rates following an increase in the number of read register ports. This suggests that a larger number of register read ports would be required to reap the full benefits of instruction fusion. With only 2 ports, which is common for a simple RISC-V processor, only a fusion rate of 7.87% is achieved. This can prove to be a significant limitation when implementing instruction fusion for a hardware processor.

However, it can also been seen that 8 register read ports are enough to reap the vast majority of the benefits of instruction fusion. This is commonly seen in high performance superscalar out-of-order cores and so it is not an impractical consideration, but the number of register read ports required would be even higher if we were to execute multiple fusable blocks at the same time by making a superscalar processor that additionally supports instruction fusion.

As seen in Figure 4.5b below, compared to the number of read register ports, a processor requires fewer write register ports to gain the same fusion benefits. Even without limiting the maximum fusable length, a processor doesn't gain much in fusion rate by going beyond 4 register write ports. This is useful for a computer architect to keep in mind when exploring instruction fusion performance relative to the number of register ports in their design.



(a) Fusion rate versus number of read ports in the processor. The dotted red line shows fusion rate at the limit.



(b) Fusion rate versus number of write ports in the processor. The dotted red line shows fusion rate at the limit.

Figure 4.5: Fusion rate versus the number of register ports in the processor.

4.4.3 Cycle Counts

As mentioned in Section 3.1.7, pipeline stalls are an important consideration when modelling the performance benefit of instruction fusion. As fused blocks are more likely to run into pipeline stalls, it is worth characterising the additional impact of pipeline stalls on instruction fusion.

Figure 4.6 below shows the impact of taking pipeline stalls into account. When considering just the effective instruction count, the effective instruction count of the non-fused version is 52.9% more than the effective instruction count of the fused version, but after taking pipeline stalls into account, the difference in effective cycle count drops to 46.2%.

Maximum fusable length, number of read ports, or number of write ports seem to have no significant impact on the difference between the instruction count and cycle count.

However, a fusion rate of 46.2% is still a very positive result and exhibits the strong potential of instruction fusion as a technique to reduce the effective instruction count of a program. It is possible that more aggressive fusion rules or more conservative pipeline stall estimates would further decrease the fusion rate, but further work would need to be done to estimate the impact.

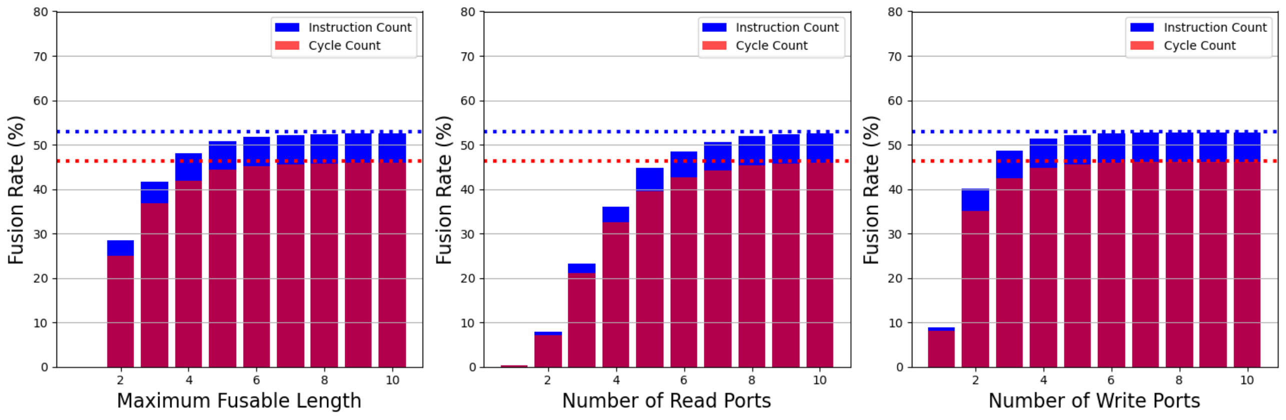


Figure 4.6: The blue bars show the percentage reduction in effective instruction count, and the red bars show the percentage reduction in cycle count. Both are plotted against the maximum fusable length. The dotted lines show the percentage at the limit.

4.5 Per-Benchmark Analysis

In this section we explore how the fusion rate differs across the benchmarks in the SPECInt2006 benchmark suite. This is important because each benchmark represents a different type of workload, and insights into whether instruction fusion works better for some workloads and not others might be useful for computer architects that are expecting more specialised workloads.

Celio et al. mentioned that the fusion rate differed greatly across different benchmarks due to the nature of the programs. Half the benchmarks exhibit less than 2% reduction while three benchmarks have a 10% reduction with `bzip2` showing a nearly 20% reduction [2].

However, in our case, as the fusion rules are much more general and widely applicable, we notice a proportionally narrower range of fusion rates across the benchmarks, suggesting that a more general approach to instruction fusion actually **decreases the variation in performance** and provides significant benefit to a large variety of programs.

The results for the per-benchmark fusion rates are shown in Figure 4.7. The minimum fusion rate across all the benchmarks is 34.4% for the `mcf` benchmark, and the maximum is 61.1% for the `bzip2` benchmark.

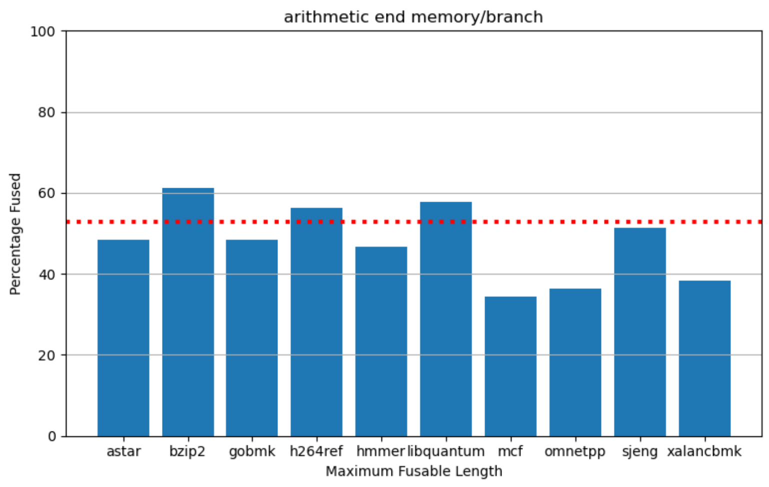


Figure 4.7: Percentage fused for each of the SPECInt2006 benchmark programs. The benchmarks `perlbench` and `gcc` were omitted due to lack of trace data.

As we can see by comparing our results with that of Celio et al. [2], the benchmark-specific fusion rate can differ greatly depending on the fusion rule that is being applied. Therefore a computer architect might be inclined to experiment with a variety of fusion rules to find out which one is the best for the expected workload for their processor.

4.6 Comparison with Superscalar Processors

Superscalar processors also try to speed up the instruction pipeline by executing multiple instructions at the same time. However, they have the restriction that the instructions can only be executed in parallel if they are independent, i.e. they do not share any common operands. This functionality can be mimicked using instruction fusion by specifying fusion rules that only fuse independent instructions. The results are shown in Figure 4.8 below.

Perhaps unsurprisingly, the additional restriction that fused instructions must be independent significantly reduces the fusion rate, with a value of **30% in the limit**.

Another result is that fusing pairs of independent instructions retains most of the benefit of fusing arbitrarily long sequences of them. One explanation for this is that longer sequences of instructions are less likely to be independent. This suggests that a dual-issue core would be able to capture most of the benefit to running independent instructions in parallel, not considering instruction reordering.

This might provide an explanation of the dominance of dual-issue processors, and why increasing the width of a processor past dual-issue seems to be giving diminishing returns. Instruction fusion might be a way to increase the width of a processor in a way that gives substantial benefit, as parallelisation is no longer limited to independent sequences of instructions.

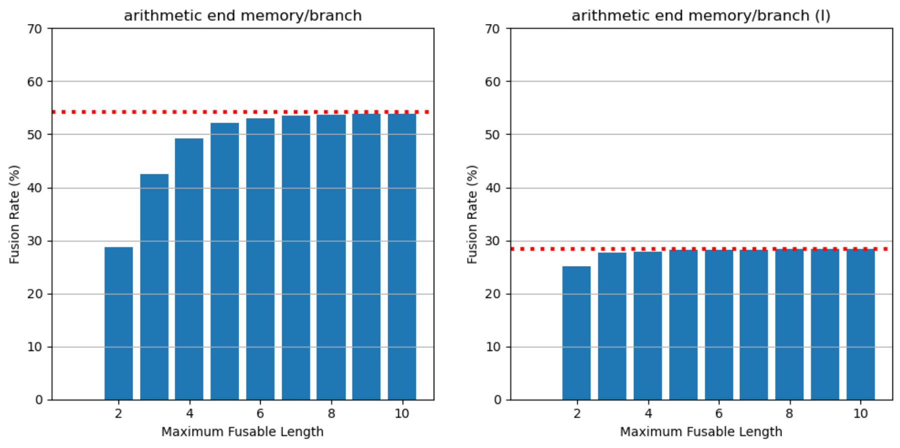


Figure 4.8: The same fusion rule, but with an additional restriction that instructions must be independent on the right.

4.7 Benefits of Using Instruction Histograms

In this section, I detail the performance of the fusion library compared to alternatives like QEMU, gem5 and ChampSim, and demonstrate why using instruction histograms is so empowering for computer architects and researchers.

Figure 4.9 below shows the relative performance between HistoSim and other simulation/emulation libraries, notably KVM [16], QEMU [17], gem5 [18] and ChampSim [19].

For HistoSim, I repeated the instruction count experiment and the pipeline experiment each 100 times, and calculated the mean, maximum and minimum instructions per second. For each of the other libraries, I searched through available literature for their best and worst case performances. Some inaccuracy is to be expected, but the difference in performance between each of them is sufficiently large that we can expect the overall trend to be preserved.

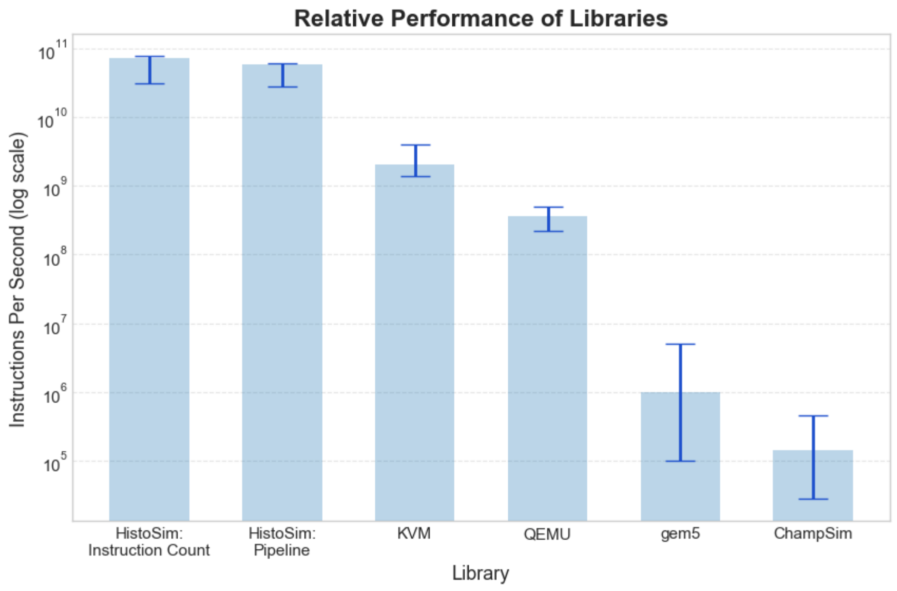


Figure 4.9: Comparison of the performance of different simulation/emulation libraries, measured in instructions per second. The error bars denote the maximum and minimum expected performance. Plotted on a logarithmic scale.

Notably, because of the histogram format^{SC} of the instruction trace data, it is possible for HistoSim to run **faster than native**. This is potentially revolutionary for computer architects and researchers that are used to waiting for days for a single benchmark to run, as it massively speeds up turnaround time and allows for faster iteration and rapid prototyping.

4.8 Success Criteria

The project met all of the success criteria of the core project, as well as a few extensions. Some of the extension goals have changed as the project evolved, as we determined that more insightful fusion results were more practical and useful compared to modifying an existing simulator to support instruction fusion.

The major components of work completed are as follows:

- I successfully created a **fusion library** for computer architects to use in their design process, and for researchers to use to gain insights on instruction fusion.
- I came up with a novel framework to define **fusion rules**, and a **fusion algorithm** that applies these fusion rules to instruction trace histograms.
- I performed a **limit study** by running various experiments to gain insight on the potential of instruction fusion for a RISC-V architecture.
- As an extension, I created another type of experiment, the **pipeline experiment**, to take pipeline stalls into account to compute the cycle count.
- I performed further analysis on the fusion results and fusion rules to gain insight on instruction fusion for RISC-V.

Chapter 5

Conclusion

5.1 Achievements

This project was a success, meeting all core and extended success criteria. I created a versatile open source fusion library that could be used by computer architects and researchers to investigate the impact of instruction fusion for a processor design. I then used my own library to run a series of experiments which highlighted the potential of instruction fusion in a RISC-V architecture, which I hope will spark further research in this subject. This is demonstrated by achieving a fusion rate of over 50% using a simple fusion rule. I characterised a few limitations on the hardware processor design that would affect the fusion rate, including maximum fusable length, the number of register read and write ports, and pipeline stalls.

5.2 Lessons Learnt

Instruction fusion is a research topic that is frequently locked behind industrial walls, so there isn't much public work on fusion, especially not the more general approach to fusion that we take in this project. As such, I had to construct a lot of the theories and methods from the ground up, which honed my problem solving skills.

I also had to adapt my approach to the extensions as the project developed, and it became clearer what remaining research direction was the most insightful. This required both project management skill and research insight to know what new information would be the most helpful to the community, while prioritising limited time and resources.

Reflecting on the process of implementing the project, the data analysis process was more messy than I anticipated, as the analysis was done using Python notebooks and had to be catered to each of the experiment types that I was running and the metrics I was measuring. In hindsight, I might have implemented a separate Python library that could make fusion specific data analysis easier for the user.

5.3 Future Work

This project highlights the potential of instruction fusion for RISC-V, but there is much work to be done before it becomes commercially viable. This includes but is not limited to:

- More accurately characterising the limitations that the hardware enforces on the fusion rate.
- Supporting non-contiguous fusion, for example by implementing the fixed window algorithm. This could open up more fusion opportunities, and we don't have to worry about potentially missing out on very long fusable sequences due to the results shown in this work that the vast majority of sequences are less than 10 instructions long.
- Exploring more sophisticated fusion rules, for example fusing across branches, predication of loops or incorporating branch prediction. Many of these would require a rollback mechanism, so further research needs to be done on how to implement this.
- Instruction fusion could also enable further optimisations such as common subexpression elimination (CSE). Further work could be done to explore this.
- Exploring compiler optimisations that can be done to improve performance of instruction fusion, for example optimise instruction ordering to maximise fusion opportunities.
- Extending commonly used simulators such as gem5 to support instruction fusion, thereby lowering the barrier for computer architecture researchers.
- Further developing the idea of using instruction trace histograms to perform computer architecture research.
- Creating a hardware processor that supports instruction fusion.

In conclusion, we strongly believe that this work provides important insights on the effectiveness of general instruction fusion for RISC-V, and encourage further work on this front. We also wish to highlight the potential benefits of working with trace histograms in speeding up research and development in this field, and hope that this work provides the proof of concept necessary to spur further development.

Chapter 6

Bibliography

[1] E. Ho and J. Woodruff, “Instruction Fusion Limit Study for RISC-V,” RISC-V Summit Europe 2025, May 2025.

[2] C. Celio, P. Dabbelt, D. Patterson, and K. Asanovic, “The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V,” 2016.

[3] J.-Y. Shen and S.-W. Liao, “Evaluating and Enhancing Performance through Macro-Op Fusion Optimization with RISC-V,” in *The 53rd International Conference on Parallel Processing Workshops.* Gotland Sweden: ACM, Aug. 2024, pp. 33–37. [Online]. Available: https://dl.acm.org/doi/10.1145/3677333.3678150

[4] S. Singh, A. Perais, A. Jimborean, and A. Ros, “Exploring Instruction Fusion Opportunities in General Purpose Processors,” in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2022, pp. 199–212. [Online]. Available: https://ieeexplore.ieee.org/document/9923815

[5] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru, “Performance Characterization of SPEC CPU Benchmarks on Intel’s Core Microarchitecture based processor,” 2007.

[6] Y. Lu and S. G. Ziavras, “Instruction Fusion for Multiscalar and Many-Core Processors,” *International Journal of Parallel Programming*, vol. 45, no. 1, pp. 67–78, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s10766-015-0386-1

[7] B. Jeff, “A walk through of the Microarchitectural improvements in Cortex-A72 - Architectures and Processors blog - Arm Community blogs - Arm Community,” May 2015. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/a-walk-through-of-the-microarchitectural-improvements-in-cortex-a72

[8] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, “The Intel® Pentium® M Processor: Microarchitecture and Performance,” *Intel Technology Journal*, vol. 7, no. 2, 2003.

[9] C. Lam, “Hot Chips 2023: SiFive’s P870 Takes RISC-V Further,” Nov. 2024. [Online]. Available: https://chipsandcheese.com/p/hot-chips-2023-sifives-p870-takes-risc-v-further

[10] B. Baktha, “Real Systems. Real Traction. The Next Chapter in High-Performance RISC-V in Data Centers.” RISC-V Summit Europe 2025, May 2025.

[11] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: Association for Computing Machinery, Apr. 1991, pp. 176–188. [Online]. Available: https://dl.acm.org/doi/10.1145/106972.106991

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[13] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," Sep. 2020. [Online]. Available: http://arxiv.org/abs/2007.03152

[14] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," Oct. 2022. [Online]. Available: http://arxiv.org/abs/2210.14324

[15] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 20–37. [Online]. Available: https://ieeexplore.ieee.org/document/7163016

[16] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed," in *2015 IEEE International Symposium on Workload Characterization*. Atlanta, GA, USA: IEEE, Oct. 2015, pp. 183–192. [Online]. Available: http://ieeexplore.ieee.org/document/7314164/

[17] T. Spink, H. Wagstaff, and B. Franke, "Hardware-Accelerated Cross-Architecture Full-System Virtualization," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 1–25, Dec. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2996798

[18] A. Sandberg, S. Diestelhorst, and W. Wang, "Architectural Exploration with gem5," 2017.

[19] E. Domínguez-Sánchez and A. Ros, "MBPlib: Modular Branch Prediction Library," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Raleigh, NC, USA: IEEE, Apr. 2023, pp. 71–80. [Online]. Available: https://ieeexplore.ieee.org/document/10158076/

Appendix A

Proof of Greedy Algorithm

Proof. by induction

Invariant: at index i, the k segments chosen by the greedy algorithm are part of some optimal solution.

Base case: at index 0, when 0 segments are chosen, it is trivially part of some optimal solution.

Inductive step: assume we are at index i, and k segments have already been chosen and are part of an optimal solution. Show that the $k + 1$th segment chosen by the greedy algorithm is part of an optimal solution.

Let the segment chosen by the greedy algorithm be denoted by A. Consider interval from i to A. In this interval, no other solution can do better than choosing A, which adds only a single segment to the previous optimal solution.

Proof. by contradiction

Let A be the segment chosen by the greedy algorithm. Assume A is not in the optimal solution.

Let B be the segment with the latest endpoint in an optimal solution.

If we take the solution with B and replace segment B with segment A, either:

1. A overlaps with B and entirely another segment C that comes after B in the optimal solution. In this case, replacing B with A gives a strictly better solution.
2. A overlaps with B and partially another segment C that comes after B in the optimal solution. In this case, replacing B with A gives an equally good solution, assuming that we can take the suffix of C to still be a fusable block.

This is a contradiction, so A must be part of an optimal solution. □

Termination: when we reach index n, the entire optimal solution must be obtained. □

Appendix B

Structure of Output Data

B.1 Instruction Count Experiment

The aggregate results CSV contains the following data:

- **rule title**: each row in the CSV corresponds to the results of applying
- **rule description**
- **user defined key**: an optional field that the user can populate for easier data analysis. For my own experiments, I populated this field with the maximum fusion length, so that I can collect the data and plot them in a graph.
- **total instructions**: the total number of dynamic instructions, summed across all the files in the experiment. This is counted before any fusion is conducted.
- **instructions after fuse**: the number of effective instructions after fusion is calculated, summed across all the files in the experiment.
- **instructions fused**: the number of instructions that have been fused with another instruction. This is equal to the difference between the total number of instructions and the number of instructions after fusion.
- **percentage fused**: the percentage of instructions that have been fused. This is the percentage benefit of instruction fusion in reducing the effective instruction count, and is calculated by
$$\text{percentage fused} = \frac{\text{instructions fused}}{\text{total instructions}}$$
- **average fusion length**: the average length of a fused block, weighted by the dynamic count.

The per-benchmark results CSV contains similar data, except that it has an extra field **file**, so the fusion results are given per input file. Each file corresponds to a single SPECInt2006 benchmark, so we can use this to find out how the fusion results differ across different benchmarks.

The fusion lengths CSV contains information about specific fused blocks. It contains the **rule title**, **rule description**, **user defined key**, **file** for indexing, and the **dynamic count** and

fusion length for data. This file is typically very large, so it's not recommended to use this for analysis that can be done on the other two CSVs.

B.2 Pipeline Experiment

In addition to **rule title**, **rule description**, and **user defined key**, the aggregate cycle CSV contains the following data:

- **cycles without stalls**: total number of cycles it takes to run a program without taking stalls into account. This is equivalent to the dynamic instruction count of the program.
- **stalls**: the number of stalls that occur in the program, weighted by dynamic count.
- **total cycles**: the total number of cycles it takes to run a program, including stalls.

$$\text{total cycles} = \text{cycles without stalls} + \text{stalls}$$

- **cycle percentage**: the percentage increase in total cycle count compared to the dynamic instruction count.

$$\text{cycle percentage} = \frac{\text{total cycles} - \text{cycles without stalls}}{\text{cycles without stalls}}$$

The per benchmark CSV contains similar data, except that it has an extra field **file** and the results are given per input file.

Appendix C

Project Proposal

Introduction

Instruction fusion (otherwise known as macro-op fusion) is the idea of fusing two or more instructions together in the microarchitecture pipeline and treating them internally as a single instruction.

There are multiple benefits to doing this over introducing a new fused instruction in the ISA, namely that this doesn't clutter up the ISA with more instructions, and gives the microarchitect more finetuned control.

For an ISA like RISC-V which is meant to be as flexible as possible, this is a good way to prevent the ISA from being bloated while still allowing for optimal performance. Therefore RISC-V might be uniquely positioned to exploit instruction fusion, if it can effectively be used.

Parameterised Instruction Fusion Strategies

My project investigates what we've decided to call parameterised instruction fusion strategies. An instruction fusion strategy is a class of instructions that I might want to fuse, for example:

1. a series of arithmetic operations
2. a series of arithmetic operations that end in a branch
3. a series of arithmetic operations that end in a memory operation
4. a series of contiguous memory operations

My project aims to investigate the performance and cost tradeoffs for each of these instruction fusion strategies, varied across multiple parameters, for example:

1. the number of instructions to fuse
2. the number of input operands

3. the number of output operands

Changing the values of the parameters mean a tradeoff in cost (in terms of the number of hardware components needed to implement the change) and performance (in terms of the number of instructions we actually save).

My project aims to investigate this performance-cost tradeoff that instruction fusion has on the RISC-V architecture for each of these parameterised instruction fusion strategies.

Starting Point

I have not done any programming or implementation prior to the start of this project. I have no prior experience with instruction fusion, implementing software or harware simulations, or conducting performance analysis experiments. My experience with RISC-V is limited to the Part 1B Computer Architecture course and ECAD practical classes. I also have no experience working with simulators such as Gem5.

Prior to starting the project I have read one reference paper on instruction fusion for RISC-V [1].

Objectives of the Project

Core Objectives

The core part of the project gives us an idea of the effectiveness of instruction fusion, but is very simple. Most of the interesting work will be done in the extension part of the project.

Fusion Strategies Decide on a set of parameterised instruction fusion strategies to investigate. A starting point is the four fusion strategies I outlined in the Introduction section above.

Traces Convert the traces I obtained from my supervisor into a useful form for my project, namely a list of basic code blocks, their constituent instructions, and the number of times each basic code block was executed.

Experiments Run multiple experiments on the formatted traces, measuring the final number of macro-ops after fusing as well give a cost estimate for implementing the fusion mechanism. Experiments will be run by varying the values of the parameters mentioned in the Introduction section.

Results Write up the results in a comprehendable form, and reach a conclusion with regards to the effectiveness of instruction fusion for the RISC-V ISA.

[1]Celio, C., Dabbelt, P., Patterson, D., & Asanovic, K. (n.d.). The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V.

Extensions

Instruction Fusion Framework In addition to having specific parameterised instruction fusion strategies, I could come up with a more general framework to derive effective instruction fusion strategies. This allows us to adapt to changes in programming practices, or to different types of programs/applications/cost framework.

More In-depth Simulation Modify an existing software simulator (e.g. Gem5) to support instruction fusion. This allows us to more closely measure the impact of fusion certain instructions (e.g. precise execution time). I would come up with a list of metrics that are more extensive than the number of instructions fused, and run experiments on the software simulator to measure these.

Hardware Module Build a hardware module and repeat the experiments.

Alternative Hardware Architectures Experiment with alternative hardware architectures such as multiple issue, superscalar, or out-of-order cores.

Success Criteria

The project is a success if:

1. Parameterised instruction fusion strategies should be designed and documented.
2. Multiple values representing the effectiveness of each instruction fusion strategy on a scalar RISC-V architecture (e.g. the number of macro-ops after fusion was done on a trace) is obtained, varied across different parameter values.
3. The results are analysed and evaluated, and a conclusion is obtained.

Plan of Work

Michaelmas Term

Week 1 (10th Oct - 16th Oct)

- Continue working on the project proposal
- Review literature on instruction fusion, and find commercial applications of instruction fusion

Weeks 2-3 (17th Oct - 30th Oct)

- Submit the final project proposal
- Obtain traces for dynamic simulation
- Format the traces that I obtained

Weeks 4-6 (31st Oct - 20th Nov)

- Come up with a set of preliminary rules for instructions that can be fused
- Come up with a cost estimate of implementing a given parameterised fusion strategy
- Come up with experiments to run
- Write an engine that can consume traces for dynamic simulation, perform instruction fusion using the rules I defined earlier, and output the resulting number of macro-ops

Weeks 7-8 (21st Nov - 4th Dec)

- Run a series of tests based on the experiments I came up with, and measure the results
- Do preliminary analysis

I should be done with the core part of my project by the end of Michaelmas term.

Christmas Vacation

Weeks 1-4 (5th Dec - 1st Jan)

- Modify Gem5 to support instruction fusion

Weeks 5-7 (1st Jan - 22nd Jan)

- break, and slack period for the first cycle of results/Gem5 implementation

Lent Term

Weeks 1-2 (23rd Jan - 5th Feb)

- Write the progress report (due 7th Feb)
- Start preparing for my presentation

Week 3 (6th Feb - 12th Feb)

- Finish the presentation and rehearse it

Week 4 (13th Feb - 19th Feb)

- Come up with an extended set of metrics to measure the performance of Gem5 with
- Come up with a set of programs/benchmarks/workloads to measure the performance of Gem5 with
- Update the set of experiments I want to run

Weeks 5-6 (20th Feb - 5th Mar)

- Run the experiments on Gem5 and measure the results

Weeks 7-8 (6th Mar - 19th Mar)

- Writing up the results that I've obtained.
- Write chapters 1 (Introduction) and 2 (Preparation) of the dissertation.

Easter Vacation

Weeks 1-3 (20th Mar - 9th Apr)

- Conduct any extensions that I have the time for.

Weeks 4-5 (10th Apr - 23rd Apr)

- Write chapter 3 (Implementation).
- Write chapter 4 (Evaluation).

Week 6 (24th Apr - 30th Apr)

- Write chapter 5 (Conclusions).
- Share draft with supervisors and directors of studies.

Easter Term

Weeks 1-3 (1st May - 16th May)

- Address review comments and make changes to dissertation.
- Submission deadline: 16th May.

Resource Declaration

I will use my own personal laptop for the project. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. If my personal laptop fails, I will use a backup Windows laptop that I have. Git will be used for version control, and the project will exist as a repository on Github. The dissertation itself will also periodically be uploaded to Github.

I might require a FPGA for one of my extensions. I might also require access to the private lab network to get some traces which would be useful for dynamic analysis.