Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Amund Bergland Kvalsvik

Investigating the Usefulness of Simulators and Prototypes as Research Tools

Master's thesis in Computer Science
Supervisor: Magnus Själander
September 2021

Norwegian University of
Science and Technology

Amund Bergland Kvalsvik

Investigating the Usefulness of Simulators and Prototypes as Research Tools

Master's thesis in Computer Science
Supervisor: Magnus Själander
September 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Contents

1 Introduction and Motivation **5**

1.1 Introduction 5

1.2 The motivation of this work 6

1.2.1 Desired Properties for Research 7

1.2.2 Using Simulators in Research 8

2 Understanding Existing Tools **11**

2.1 Processor Background 11

2.1.1 Terminology 11

2.1.2 Architectural and Micro-Architectural 12

2.1.3 Superscalar Processors 13

2.1.4 Out-of-Order Processors 15

2.2 The Berkeley Out-of-Order Machine 18

2.2.1 Parameters 18

2.2.2 Frontend 19

2.2.3 Branch Handling 20

2.2.4 Execution Stage 21

2.2.5 Load Store Unit 21

2.2.6 Reorder-Buffer 23

2.3 Hardware Tooling 23

2.3.1 RISC-V 23

2.3.2 Chipyard 24

2.3.3 Verilator 24

2.3.4 FPGAs 24

2.3.5 FireSim 25

2.3.6 Hardware Tools for the BOOM 25

2.4 The gem5 simulator 26

2.4.1 The origin and purpose of gem5 . . . 26
2.4.2 A cycle-accurate simulator in software . . . 27
2.4.3 Parametric, components and statistics . . . 27
2.4.4 Pipeline Design, the O3 CPU and Design Considerations . . . 28
2.4.5 The O3 CPU Load Store Unit . . . 30
2.4.6 The gem5 memory system . . . 32
2.5 Speculative Attacks and Delay-on-Miss . . . 34
2.5.1 Speculative Execution and Attacks . . . 34
2.5.2 Spectre . . . 35
2.5.3 Mitigations against Speculative Attacks . . . 35
2.5.4 Challenges within Spectre Research . . . 37
2.5.5 Delay-on-Miss . . . 37

3 Detailing the Implementation of Delay-on-Miss 41

3.1 Implementing Delay-on-Miss on the BOOM . . . 41
3.1.1 A note on working with hardware . . . 41
3.1.2 Implementing Delay-on-Miss on Scalar BOOM . . . 42
3.1.3 Implementing Delay-on-Miss on Superscalar BOOM . . . 46
3.1.4 Supporting a 3-wide Processor . . . 49
3.1.5 End Result and Challenges . . . 51
3.2 Implementing Delay-on-Miss on gem5 . . . 54
3.2.1 Creating Tracking Structures . . . 55
3.2.2 Updating Support Structures . . . 57
3.2.3 Handling Squashing . . . 57
3.2.4 Delaying Loads . . . 59
3.2.5 Retrying Loads and Statistics . . . 60

4 Gathering and Presenting Results 61

4.1 Methodology . . . 61
4.1.1 Notable Caveats . . . 61
4.1.2 SPEC2006 Suite . . . 62
4.1.3 Data Results Collection . . . 63
4.1.4 Data Collection on the BOOM . . . 64
4.1.5 Data Collection on gem5 . . . 64
4.1.6 BOOM configuration . . . 64
4.1.7 The gem5 simulator configuration . . . 64

4.2 Results 66
4.2.1 Showing a successful mitigation 66
4.2.2 Comparing Performance 69
4.2.3 BOOM Results 70
4.2.4 gem5 Results 75

5 Discussion and Evaluation **81**
5.1 Discussion 81
5.1.1 Result Trends from Hardware Prototype and gem5 81
5.1.2 Evaluating Delay-on-Miss 82
5.1.3 The Validity of this Research 83
5.1.4 The Implementation Problem 84
5.1.5 Minimizing inaccuracies 86
5.1.6 The Value of Simulators 87
5.2 Future Work 89
5.2.1 Finalizing the FPGA implementation 89
5.2.2 Improving the gem5 implementation 90
5.2.3 Other Simulators 91
5.2.4 Small Delay-on-Miss with only C-Shadows 91

6 Conclusion **93**

Bibliography **95**

A Results of Running Spectre Attacks **99**

B Hardware Size Analysis **105**

C Timing Analysis **113**

Abstract

Processors design has become highly complex over the years, which has complicated computer architecture research. To simplify the research process, many researchers therefore utilize simulators, such as gem5, to be able to more quickly model and explore design changes. Simulators model simplified versions based on real designs, and although they provide many benefits, using them while unaware of their limitations can introduce pitfalls that reduce the accuracy of research. Hardware prototyping remains an alternative to using simulators, but is generally too time-consuming for research, despite providing further insights.

This thesis implements a state-of-the-art security technique, Delay-on-Miss, on both the gem5 simulator and the Berkeley out-of-order machine, and uses the two implementations to investigate the benefits and limitations simulators and hardware prototyping provide as research tools. Notably, we find that hardware prototyping is suited to giving accurate estimations of key properties such as size overhead, and is vital to discovering design limitations that are not intuitive in a simulator design space. For simulators, we show that it is possible to achieve near-identical quantitative results to a hardware prototype under certain circumstances, all the while achieving greater insight into the mechanisms by which design changes affects performance.

In addition to investigating the usefulness of simulators and hardware prototypes, this work furthers research of Delay-on-Miss. To the best of our knowledge, this is the first work to verify Delay-on-Miss on a hardware prototype. We find that the hardware prototype incurs a size overhead of 2.5% and a performance slowdown of 20%, results that align closely with previous research results from the authors of the technique. We also verify that the implementation of DoM does not affect the critical path on BOOM.

The key contributions of this thesis are showing that simulators can achieve a high level of accuracy for numerical results when limitations are addressed, and how to reduce inaccuracies when performing research using simulators. This thesis therefore strengthens the credibility of simulators as a valuable cornerstone of the computer architecture field, and provided a nuanced view on the value of simulators and hardware prototyping.

Acknowledgments

It is said that it takes a village to raise a child, and the same could easily be said of this thesis. I am sure there will be others whose contribution I will have missed, but nevertheless I wish to attempt to thank as many as possible.

First, my colleagues: Thanks to David Metz for his continued support and guidance in getting familiar with most of the Chipyard ecosystem, for helping me set up debugging, and so much more. Björn Gottschall also provided incredibly valuable help in setting up the benchmarks and providing valuable input to earlier discussions. Finally, Bart Iver van Blokland provided great aid in providing a platform for me to run my experiments on, and some helpful comments on the introduction.

I wish to thank both my parents for reading through this far too long work. Their comments were invaluable for improving readability and fixing a myriad of language issues. I am also grateful to my mom for providing a much-needed change of scenery during July.

The comments of my supervisor, Magnus Själander, were invaluable to elevate this work to the current level, and I am especially grateful for him taking so much time out of his vacation to aid me. The technical level of this thesis would be far lower had it not been for his many detailed comments and suggestions.

Finally, I must thank my significant other, Karina Lofquist. Without her continued support and motivation, this work would simply not be. Between the many encouraging messages and cups of coffee, you truly have my gratitude.

Chapter 1

Introduction and Motivation

1.1 Introduction

Computers have never been more important and present than in the 21st century, in which the ubiquity of technology has reshaped both the prevalence and variety of computer systems available. Modern processors' performance are orders of magnitude greater than those of a few decades ago. This performance leap has however also resulted in a great growth in complexity. Processors are now composed of several billions of transistors, and although tools for design and research have also improved, the increased complexity has made working with processor design considerably harder.

The complexity growth of processors has been gradual, and is owed to a series of developments within materials, lithography, tools, and design strategies. Both older and newer computer architecture techniques, such as pipelining, caching, out-of-order processing and register renaming, have provided considerable speedups, but also increased the baseline complexity of processors. This has resulted in processor engineering teams becoming larger and more specialized.

Nowadays, computer architecture researchers rely on simulators and more general modelling when performing research. It is not possible for academics to work directly with commercial processors, as they remain under secrecy by key industry actors like Intel and AMD, and there is only transparency into older processors and reverse-engineered design features. In addition, modern processors are too complex to modify to allow for efficient research. Simulators, especially those maintained by the academic community, such as gem5, have therefore taken a central position within the field, both for academics and industry researchers.

These simulators can create pitfalls when used without properly understanding their limitations. Simulators have to make a trade-off between design properties such as ease-of-use, modularity and realism. If researchers are not sufficiently aware of the trade-offs a specific simulator uses, they may end up with inaccurate results. This is partially due to the abstractions used by simulators: simplifications that are meant to aid ease-of-use and modularity, but may reduce realism. Reliance on these simplifications can in the worst case invalidate research, when incorrect or imprecise assumptions are made about the actual design of hardware systems.

Avoiding these pitfalls is not trivial. Academics are familiar with most modern techniques employed in processors, and regularly reverse-engineer techniques present in new processors. Despite this, knowing the intricate implementation details of all these techniques, and also understanding emergent properties of processor design, is not typically feasible. The lack of competitive open-source versions of out-of-order processors and industry transparency further complicates this.

These challenges highlight the need for developing intuition for the more practical aspects of hardware design, as this allows for more understanding of real systems and allows researchers to navigate around simulator limitations. We posit that it is necessary to perform hands-on work with hardware in order to develop such intuition. Prototyping state-of-the-art techniques in hardware can give invaluable insight into both hardware design constraints and the prototyped techniques.

This work reproduces a state-of-the-art security technique, Delay-on-Miss (DoM) [1], on the Berkeley out-of-order machine (BOOM) [2], as detailed in section 3.1, and uses this to explore the challenges and advantages of hardware prototyping as a research tool. We find that the hardware implementation of DoM alters its design and unveils new design modifications, including the need for misprediction support, as well as stalling branch dispatches. These properties were not discussed in its original presentation and highlight how hardware work can uncover new design limitations and insights, which we discuss in section 3.1.2 and section 3.1.5. In addition, we also propose an alternative design for DoM based on the architecture of the hardware platform, that we present in subsection 5.2.4.

To the best of our knowledge, this is also the first work to verify a functioning DoM implementation as a hardware prototype and perform size and timing analysis based on these properties. Hardware analysis tools give a 2.5% size overhead for DoM, and additionally demonstrates that it is possible to implement DoM without negatively affecting the critical path on the BOOM. We present these results in section 4.2.3, highlighting the unique value of hardware prototyping and the more accurate implementation properties it can provide as a research tool.

The insights gained from the DoM prototype are also utilized to create a simulator-equivalent implementation, which reports near-identical performance results. The performance results of both of these implementations, as shown in subsection 4.2.2, demonstrate that simulators can provide highly accurate numerical data, as long as the underlying hardware is adequately understood. Our performance results also corroborate the performance ranges that the authors of DoM presented in earlier works. The similarities in numerical data trends between simulator and hardware prototype are precise enough to be observable in individual benchmarks, and strengthens the credibility of simulators, as we highlight in subsection 5.1.1.

In addition to the aforementioned results, this work aims to demystify many hardware details and aid future research to be more precise. We accomplish this by detailing the work that went into the hardware prototype, and providing guidelines for creating hardware realistic research in subsection 5.1.5. We demonstrate many of the design challenges this work faced and provide guidelines for others wishing to understand hardware design limitations. Additionally, we present a nuanced perspective on the advantages and disadvantages of both hardware prototyping and simulators as research tools, and highlight how the two approaches can both be used to create better research.

As this thesis occupies an intersection of research considerations, hardware design, and simulator design, a considerable amount of background knowledge is required. In order to properly convey the motivation behind this work, we first present some properties of quality research according to our axioms in subsection 1.2.1, and then present and discuss simulators in subsection 1.2.2. We then provide the necessary background, first for processor design in section 2.1, for the hardware platform used in section 2.2 and section 2.3, and then for the simulator used in section 2.4. DoM is then explained in section 2.5, before we detail creating the hardware prototype in section 3.1 and how we created the subsequent simulator implementation in section 3.2. The methods used to evaluate the performance of these implementations is discussed in detail in section 4.1, including notable caveats due to differences between the two platforms. The results of our experiments are presented in section 4.2. The key insights of this work are discussed in section 5.1, before we explore avenues for future work in section 5.2. We conclude based on our findings in chapter 6.

1.2 The motivation of this work

This work aims to investigate, amongst other things, the precision of simulator-based research. As simulators are utilized frequently in computer architecture as a research tool, it is important to examine the extent to which simulators provide precise results, as to maintain the credibility of computer architecture research. To discuss the value of simulators as pertaining to research, we first need to present what properties we desire research to have, and then discuss what simulators are and how they are currently used. The following discussions relate to the computer architecture field, and should not be viewed as an analysis or judgement of research in general.

1.2.1 Desired Properties for Research

What makes for good research is a complicated question, and outside the scope of this thesis. Instead, we first highlight three key properties that we deem valuable to producing quality research, namely the ability to reproduce previous research, research being appropriately evaluated against other research, and research being well presented. These properties are axiomatic, but are argued for. Afterwards, we will introduce a fourth property that is helpful when evaluating the hardware feasibility of research, namely *transferability.*

We define *reproducible* according to the ACM standard [3], which states that reproducibility is when *"The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts."* For this work, measurement would be numerical results, such as performance results, as well as size overhead. In addition, the standard also defined *replicability*, as being able to obtain the same measurement using other systems. For this work, we attempt replicability as we are measuring the results of Delay-on-Miss on a new platform, namely a hardware prototype. For the rest of this work, we will be referring to both of these concepts as reproducing previous work. To encourage reproducibility, researchers should share their research artifacts, such as experimental configurations, utilized benchmarks, and most importantly source code. Being able to reproduce previous research is critical for the credibility of a research field.

For research to be *comparable*, it has to compare itself with and perform data collection using methods similar to related work within that research field. If the research in question radically alters the entire design of a processor, it would not be fair to compare its performance gains to a minor modification that is design-friendly. Similarly, when working within a field or around a specific problem, such as mitigating speculative execution attacks, it is important to use the same benchmarks as used by other research in the field. When using non-traditional benchmarks, these should not be cherry-picked to highlight only the strengths of their developed method, but to provide a fair and balanced comparison.

Well presented means that dissemination is comprehensible and unambiguous. Research within computer architecture is often very complex, and it can be difficult to understand objectives the research had. It is even harder to properly understand the nuances of the results research provides. Therefore, it is imperative that all research is presented in a manner which minimizes confusion, and presents both strengths and weaknesses fairly. This is challenging in academic ecosystems, in which publication accepted is usually motivated in part by the results of the research. Highlighting the limitations of a research project can therefore lower chances of getting published. It is critical that research is presented in a fair light from those conducting it, so that the work may be understood appropriately within the field.

With these considerations in mind, we introduce one additional property for research, namely *transferability.* Within computer architecture research, there is a large amount of research being conducted, both within well-explored areas and more experimental areas. Both of these fields are valuable research fields, but have different prioritizes and challenges. When working within well-explored areas, the research is typically more focused on solving a single problem efficiently. In these cases, it is important that the design details that underpin a work of research is transferable to real processors, as the main contributions are usually intended to be limited to this scope. In order to achieve transferability, it is therefore important that research models reflect real hardware designs accurately.

In order for research to be transferable, research tools have to provide platforms that are transferable themselves. The most precise platform would therefore aim to accurately model competitive hardware designs. However, as previously discussed, these designs are not easily available. In addition, when a research tool becomes more detailed, it also tends to become more complex to use. This can create constraints that limit design flexibility, increase development time, and limit scope of research. Therefore, research tools need to strike a balance between transferable designs and ease-of-use.

1.2.2 Using Simulators in Research

A simulator is a research tool that utilizes varying amounts of abstractions to enable research, while attempting to emulate the behavior of real-world components, such as processors. Simulators attempt to strike a satisfactory balance between the trade-offs of ease-of-use, modularity, and realism. Currently, they are one of the most common tools by which modern computer architecture research is performed. It is therefore important to understand the differences between simulators and hardware, and why simulators are designed in the way they are.

Research will always have limitations, but simulators generally aim to provide a realistic platform on which research can be performed in a reproducible manner, that allows for insightful comparisons to similar work. Simulators do strive to be realistic, but lose some realism as they necessarily prioritize ease of use. The userbases for simulators will have different requirements for desirable properties. While one user might desire a precisely modelled instruction dispatch, this might add unhelpful complexity to another user's desire to investigate caches. As such, a balance has to be struck between the wishes of the different users.

Simulators employ various amounts of abstractions in order to simulate an underlying system. As long as all parts of a design are not physically realized, some amount of the design is being simulated. When we talk about simulators in this work, we refer to tools in which the designed system is approximated in a manner that gives an acceptable loss of precision. Simulators try to capture key properties and interactions within a design, giving a realistic overview over how changing these components, interfaces, and properties would affect the overall performance of the system.

When considering only the highly accurate simulators, there are still notable differences between the most detailed and the simplest, and there is no universal answer as to what level of abstraction is the most useful. Simulators have to base their design on existing designs, either theoretical or real, and there is no perfect design by which to do so. Within hardware development, there are huge design differences between products developed by market leaders. Many of their design decisions for those products can be based on considerations such as backwards compatibility, stem from oversights, or otherwise be the result of a limitation that provides no benefit for the general purposes of computer architecture research. Additionally, current leading designs might still be limited in ambition compared to what the next generation will entail, or be overly tuned for performance. This might limit design space too much to allow for meaningful research. Therefore, no simulator will ever truly be able to capture all characteristics and design considerations of all possible systems, nor would it even want to. The level of abstraction used in simulators has to be considered likewise, and an acceptable middle ground found.

For this work, we will often be referring to levels of abstractions, in which a high level of abstraction indicates a simplified modelling of a hardware interaction, while a low level of abstraction indicates a more detailed modelling of a hardware interaction. For example, a high level of abstraction for performing a load memory operation would be to move that data directly from memory into the requested register. A low level of abstraction would be adding the request to a load store unit, issuing a request and waiting for a cache state update and writeback.

In contrast to using simulators, it is possible to conduct research by creating hardware prototypes or by using systems that simulate hardware. As discussed, there are no perfect systems to simulate, and this also goes for choosing which hardware platform to develop your prototype on. Different hardware platforms might have better or worse performance, and might make non-conventional design choices, depending on their intended purpose. Working on a hardware platform requires considerably more detailed work. This is because the much more constrained nature require implementing design modifications fully into a system, with a minimal amount of abstractions.

Designing and building a processor, especially balancing a processor in such a manner that no single part becomes a consistent bottleneck for the system, is very difficult. There are therefore multiple risks when using hardware systems: it is possible to use a system in which an unnatural bottleneck dominates performance in a system, either massively lowering or raising the impact of intended design modifications. In addition, other mechanisms that might be near-universal in processors might be ignored, as the researchers were not aware of these, or they were not present in

the chosen hardware platform for other reasons. Avoiding these challenges is not trivial either, as understanding any processor design is complicated, and the lower amount of abstractions increases the cognitive load necessary to effectively work with the system.

As highlighted, neither simulators nor hardware platforms serve as a perfect research platform, and both provide notable benefits and limitations. There is no absolute answer to which tool should be utilized when, but as the rest of this work will demonstrate, there are notables challenges and advantages to both that can provide drastically different perspectives on the same problem. Ultimately, this work wishes to highlight both of these and present how a detailed understanding of hardware prototyping can improve the accuracy of simulators as a research tool.

Chapter 2

Understanding Existing Tools

This work requires a considerable amount of understanding within processor design, hardware design, simulators, and hardware tooling. Some of this material is readily accessible in terms of academic work such as papers and books, as well as documentation. A notable amount of the necessary material, presented in the following sections, is not easily available and requires a notable time investment to collect and organize. In addition to information pertaining directly to the systems we use in this work, this chapter also strives to present an understanding of many of their underlying features, design philosophies and intricacies. This understanding is necessary to comprehend the details in chapter 3, but should also be independently viewed as a valuable research contribution of this work.

2.1 Processor Background

This chapter covers an in-depth explanation of key components of modern processors. Although not every single topic can be covered as a matter of brevity, attempts have been made to give a sufficiently comprehensive understanding of how modern out-of-order (OoO) processors operate. The later parts of this section cover some key modern design strategies. Basic understanding of how a processor works, as described in Hennesy 2011 Appendix C [4], is expected knowledge.

2.1.1 Terminology

This work covers a lot of different processor designs with radically different layouts and modes of execution. Therefore, a lot of different terminology is used to describe the different designs.

For this work, a baseline in-order (InO) processor is considered to be non-superscalar and pipelined, consisting of the following five stages: instruction fetch (IF), instruction decode (ID), execution (EX), memory (MEM), and writeback (WB). This is the classic design for the reduced instruction set computer (RISC), and although a large simplification of even the required architecture, it is a useful baseline. Within the base InO processor, all stages are similar in design, and there is only one execution unit. All stages of the pipeline receive input, process these inputs, and ready them as outputs for the next stage in one cycle.

When describing superscalar processors, the terminology is mostly the same, except that it is also necessary to discuss dependencies between instructions which are entered into the pipeline in the same cycle. We will refer to these dependencies as same-cycle-dependencies (SCD) and we will refer to the instructions that are in the same pipeline stage at the same time as a bundle of instructions.

When dealing with an OoO processor, the previously discussed pipeline is split into two general stages, the InO and OoO parts, which we will refer to as frontend and backend. IF and ID are split into more stages, namely fetch, decode, register renaming and finally dispatch. These stages

together are referred to as the frontend of the processor and is in-order. The EX, MEM and WB are now changed into execution units and memory units, both having a writeback stage as part of their execution. There is also a commit stage after an instruction has completed. These stages are referred to as the backend of the processor.

For OoO processors, we also use the terms dispatch and issue. Dispatch is when an instruction is sent from the frontend to the execution units and the reorder-buffer, while issue is when an instruction is sent for execution from the issue queue of an execution unit.

Discussing the simulator we are using, gem5, can get confusing, as it is both a software system in and of itself, but it is also simulating the execution of another program. As such, we use the following terminology for clarity's sake: Control flow when discussing gem5 relates to the execution through gem5 functions and structures, not the system that is modelled underneath. As gem5 aims to mimic hardware interactions, but does not directly simulate them, it is deemed more valuable to examine how gem5 behaves than the processor design upon which it is based.

We also establish some of the terms that will be necessary for when we are discussing the execution of programs. Instructions which cause a speculative state based on a change in processor control flow, typically conditional branch instructions, are referred to as control instructions. They are not labelled as branch instructions because not all branch instructions are conditional and in some systems a jump return instruction might be speculative.

We define squashing an instruction as equivalent to removing it from the processor and preventing it from affecting the visible state of the processor in any way. Similarly, a rollbacked instruction is an instruction that has affected the visible state of the processor, and then had this effect reverted. Squashed and rollbacked instructions are both used to guarantee eventual correctness in processors, which will be discussed in subsection 2.1.2.

All instructions that are issued by the processor and reads from the caches or from the memory are referred to as a load instruction. The distinction of whether the issuing unit was the LSU or a prefetcher is used to separate load instructions that are a part of the executed program from loads that are issued by prefetching units in order to improve performance.

2.1.2 Architectural and Micro-Architectural

This work will repeatedly be referring to architectural and micro-architectural state within processors. This section will aim to give a thorough description of what both of these concepts entail and why the distinction matters, as well as some discussion on what has been suggested regarding the many challenges that this distinction creates.

Architectural state is based on the set of rules created by the instruction set architecture (ISA). An ISA defines most of the details of a computer system, including the number of registers, what each instruction should or should not do and how control registers function, what sorts of memory reordering that is allowed, and other properties. The ISA is therefore a fairly comprehensive definition of all requirements that must be met for machine code to operate in a certain manner, independent of the system. If a processor operates in the requirements laid out in the Intel x86_64 ISA, then it is an x86_64 processor, for example.

However, by necessity, an ISA is not complete. It does not define the underlying hardware, as that would negate the point of the ISA, namely the ability to design different processors that can run the same machine code. As such, the ISA is an abstraction implemented on top of the actual underlying hardware, and more properties of the ISA are abstracted by hardware design than might be initially assumed. For example, although ISAs define registers, these do not usually correspond to the physical registers of the processor in modern designs. Rather, the ISA defined registers are referred to as architectural registers. As long as the chains of dependencies between operands are observed, it does not matter what actual physical register each architectural register is temporarily mapped to. This is the basis of the register renaming process discussed towards the end of this background section.

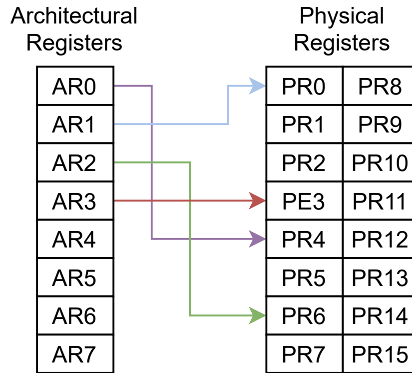

Figure 2.1: Architectural registers might map to physical registers. Some architectural registers not currently in use might not map at all.

Understanding this, we explain the differences between architecture and micro-architecture in terms of the state of a processor. We define state as the temporally current value of all parts of a processor, and the state of an individual component to be the value it holds at a given time. For example, the state of a register would be what value that register holds at the given time, while the state of the branch predictor would be its prediction history (assuming a history-based branch predictor). The architectural state is therefore the value of all objects that are defined by the ISA at a given point in time. The micro-architectural state is the value of all objects in the underlying hardware. Therefore, at a given point, architectural register ar0 might have the value 4, and if that architectural register is mapped to the physical register pr4, that register might also have value 4. However, pr4 might already have been reallocated to a new architectural register and if all dependent operations have already used ar0 and ar0 is scheduled to hold a new value, then there might be no physical register that holds the value 4. We show this relation in figure Figure 2.1, which shows both how architectural registers can map to various physical registers and how there can be a different number of physical and architectural registers. In addition, many of the registers might not be mapped at all.

These differences in state can be complicated, but understanding the nuances of these is typically not required for anyone except the engineers designing the system. The important thing is that the underlying micro-architectural state eventually satisfies all the requirements of the architectural state. Theoretically, the micro-architectural state can deviate drastically from the architectural state for long periods of time, but as long as all observable effects, such as writing to memory, are ensured to be architecturally correct, this deviation does not matter. This is the basis of speculation, in which an assumption is made to improve performance, and if the assumption is wrong, the micro-architectural state is corrected, but the architectural state never experienced visible changes based on the assumption.

However, these differences can lead to unforeseen complications. The micro-architectural state is not isolated from the architectural state, and is indirectly observable by the architectural state. An example of this are cache lines, which have different access times depending on the underlying micro-architectural state, which is not defined by the ISA. This means that any speculation that might affect the state of the caches can still be observed by the architectural state by using timing differences, despite the architectural state not being affected functionally by the micro-architectural state of the caches. As we will explain in section 2.5, this has created notable security weaknesses.

2.1.3 Superscalar Processors

A superscalar processor is defined by its ability to handle more than one instruction per cycle. Processors that are not superscalar can at most achieve an instruction-per-cycle (IPC) of 1, if it is pipelined. However, a superscalar processor can potentially achieve an IPC of more than 1, by fetching, issuing, executing and committing more than one instruction in a single cycle. This change to a processor is not trivial, however, and requires a series of changes to the entire processor.

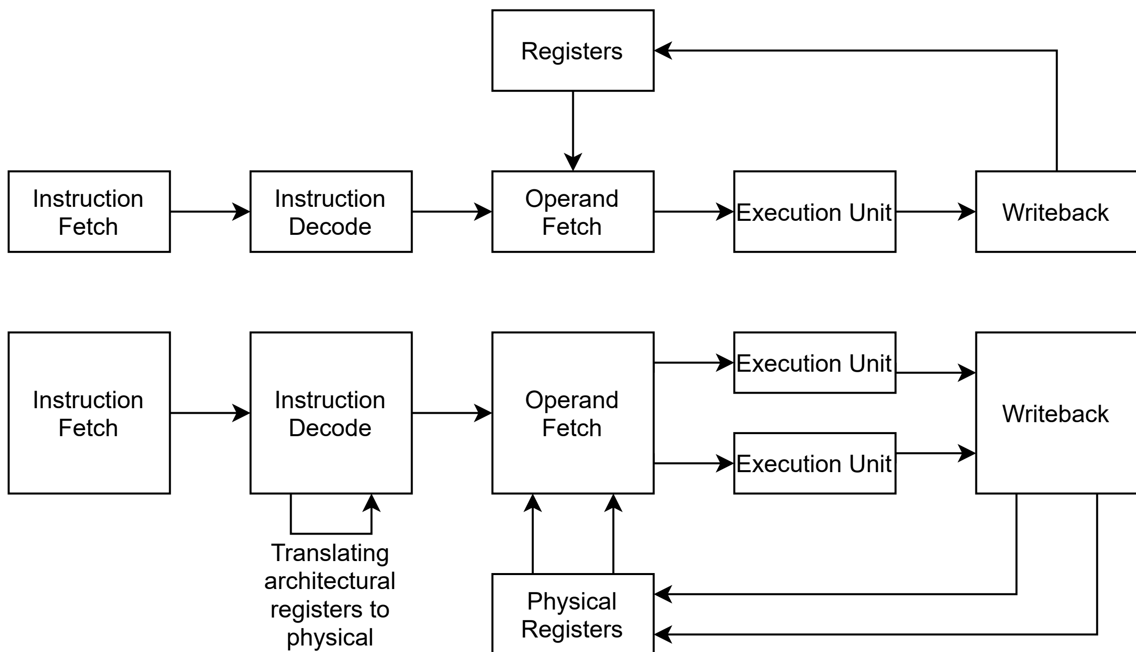

Figure 2.2: An example of the increased complexity necessary to go from scalar to super-scalar. Note that all the stages are now twice as wide, except for the execution stage, which now consists of two independent execution units.

Notably, managing dependencies becomes considerably more complicated. Although superscalar processors are also typically OoO, this is not a requirement for a processor to be superscalar. This section explains the key changes to make a processor superscalar.

Firstly, in order for a processor to be able to accommodate more than one instruction at a time, it is necessary for it to have the "width" to process these instructions at every part of the pipeline. This makes intuitive sense: if there is any bottleneck that is unable to accommodate more than one instruction somewhere in the pipeline, and this bottleneck is unavoidable, then the processor cannot handle more instructions in a cycle than that bottleneck is able to handle. In such a case, there is minimal reason in making the other parts of the processor capable of handling more instructions, as this one bottleneck will always be the limiting factor. Therefore, for a superscalar pipelined processor of any type, we need the ability to fetch multiple instructions, decode multiple instructions, execute multiplem and writeback multiple instructions in a single cycle.

Interestingly, it is not equally necessary to increase the width of both the memory unit and the execution unit, as they are not unavoidable parts of the pipeline. Some instructions do not use the execution units and some instructions do not use the memory unit. Therefore if we have a superscalar of width two and the two instructions we bundle are one that uses the execution unit and one that uses the memory unit, they will not stall the pipeline waiting for available resources. This is assuming that the system's memory operation is not dependent on the execution unit for address calculation. On average, around 25% of all executed instructions are memory instructions [5], so for a uniform distribution of instructions we would expect to see every other bundle of instructions use the execution unit and the memory unit. As a result, we would expect to see some higher amount of IPC, even without increasing the amount of execution or memory units.

However, the architectural design of this superscalar would be needlessly complex, as one would need to dynamically allow the two instructions travelling through to share resources. The decoder would have to explicitly address all stages of execution, and a large amount of information would have to be transmitted. Therefore, for a simple superscalar design, it might make more sense to have at least two execution units, as we display in a superscalar example in Figure 2.2, and rather not allow memory instructions to be bundled together to prevent congestion on the memory unit. This makes sense both because memory tends to be more expensive to wire together than execution

units, and also that since only one in four instructions are memory instructions, we would very rarely actually have to delay an instruction to prevent two memory instructions in one bundle. With a uniform distribution, we would only expect 1 in 16 bundles to result in this sort of delay.

When designing superscalar processors, it is necessary to consider the SCD between the bundled instructions. While pipelining can itself resort to stalling of the pipeline or otherwise more complicated data forwarding, in general it maintains a sense of atomicity, in that instructions will sequentially depend on each other. This makes checking for dependencies considerably easier than in a superscalar processor, where you end up with a register potentially being both the output register of one instruction and the input register for another instruction within the same cycle.

This issue can be solved in many different ways, depending on the needs of the processor. The simplest solution is to naively not put in more instructions if there are output-input chains between two instructions in a bundle. This is very simple to implement and maintain for simpler superscalar designs, especially when dealing with only a width of two. However, preventing dependent instructions from being bundled together scales poorly, and is therefore not feasible for larger designs. In addition, there is a large risk of this limiting the usage of the processor width, as output-input dependencies are very common. With a design-aware compiler, one could arrange the program order of instructions to minimize this limitation, but this would still not give good performance for sufficiently wide processors.

Another, more scalable approach is to differentiate between architectural and physical registers. This is typically what is done in OoO processors. This approach allows for much more flexibility when working with CPU cores with greater width. This strategy involves what is called register renaming, where a file is used to keep track of which architectural registers, the ones referred to in the ISA, that map to which physical registers, the actual registers implemented in the hardware. This strategy is part of Tomasulo's algorithm [4] (pages 145-149), which is used for full OoO processing and will be discussed more later. This gives a lot of flexibility, in the sense that input-output register chains of instructions can still allow for bundling.

However, without other methods, this improved bundling does not necessarily improve IPC in any meaningful way. This is due to the limitations of the sequential processor design: as the execution units are still in one specific cycle, one of the instructions would have to stall while the other instruction is executed. Depending on implementation, this could enable one instruction to continue on as the other executes after having received its updated operands, but this would only recover half of the lost IPC. However, based on the prevalence of memory instructions and memory instructions that require execution unit calculations, this might still recover parts of the lost IPC from input-output dependencies. This can especially help in systems with a larger core width in which this could potentially allow multiple instructions to move further down the pipeline. Care has to be taken to ensure that memory will be deterministic and correct when allowing instructions to run ahead, and this approach therefore approximates an OoO processor.

2.1.4 Out-of-Order Processors

Having looked at some of the complexities of the superscalar processor, we now explore changing a superscalar processor into an OoO processor and discuss associated design changes, noting also the increase in complexity. This section will show the progression of superscalar into OoO and how the two are to a large extent intrinsically linked, meaning that many of the discussed techniques for superscalar only make sense when also employing OoO execution.

Firstly, let us examine what OoO execution means and how it works in terms of program correctness. For a program to be considered to have executed correctly, we would expect its output to be both deterministic and correct, which means that regardless of memory changes, program contention or other hardware related issues, the program will always give the same results. OoO execution means that the internal order of instructions can be shuffled to improve performance, but this cannot result in creating incorrect architectural states. However, while incorrect states need to be rectified, this does not mean that it is necessary to prevent incorrect states from ever occurring: By allowing incorrect states to occur, but correcting them, it becomes possible to im-

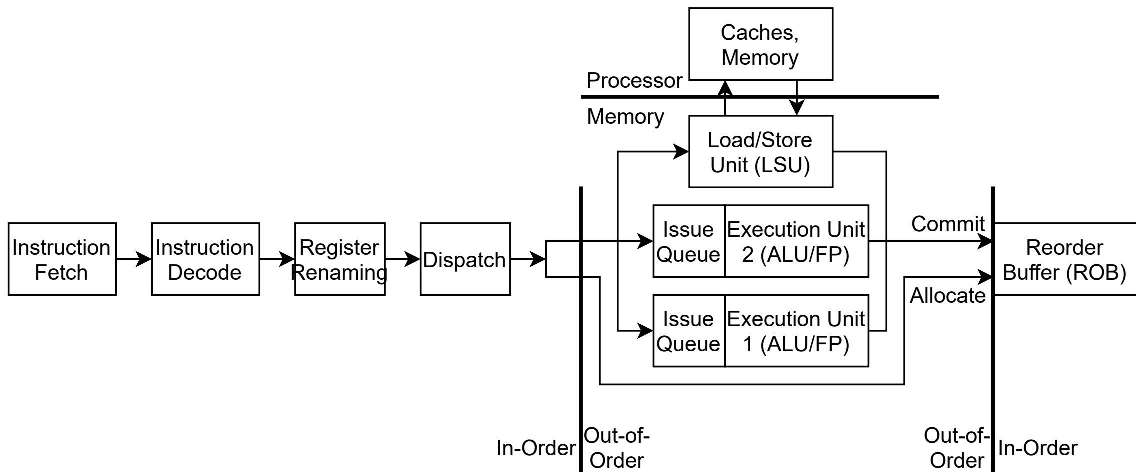

Figure 2.3: An example of a two wide out-of-order processor. Only execution is performed out-of-order. Orders are allocated and committed in the ROB in-order. Writeback is not shown to simplify the figure.

plement drastic performance improvements. Earlier designs could also enter a speculative state, by using branch predicctors, but due to InO execution, any incorrect branch prediction would be resolved before any results could propagate. As such, we do not consider the earlier designs to have achieved an incorrect state.

OoO processors use two key concepts: The ability to reorder the execution of instructions vis-à-vis each other to improve performance, and a promise of eventual correctness. Using these two concepts, performance enhancing techniques such as speculation become available, but there is also an increased overhead in order to achieve eventual correctness. We present now two of the major design changes present in OoO processors, namely the InO frontend and the reorder buffer (ROB).

Building on the details introduced in subsection 2.1.3, one of the main limitations preventing increased IPC from superscalar design are the chains of output-input registers between instructions that prevents them from being bundled together. Processors are never able to fully utilize all of their capacity at all times, but are usually built to ensure a mostly consistent throughput and high component utilization. When one of these chains occur, at least one instruction would have to stall, and as the pipeline does not allow for free movement of ready instructions or for them to be reordered, the superscalar processor is consistently unable to effectively utilize its width, even though there are notable size overheads. Superscalar is therefore a lot more enticing in specific use cases where stalling is very infrequent and the throughput is consistent.

OoO gets around the limitation of input-output dependency chains by separating the execution stage into independent execution units, while tracking dependencies for each operation independently and not reserving execution space before the operands for an instruction are ready. This both requires and enables many changes to the overall design of the processor. Importantly, the full detachment of the execution stage as being OoO and not restricted to single-cycle completion allows for more specialized units. Modern OoO processors usually have at least both integer execution units and floating point execution units, and usually also a few other specialized execution units. Note that this detachment also requires more support and as such increases overhead. An example of a detached execution stage that allows for reordering is shown in Figure 2.3. Note that writeback is not shown to simplify the figure.

Starting from the frontend, the instruction flow is much the same as before, with a notable exception: the register renaming strategy discussed under superscalar now becomes a necessity and architectural registers are now mapped to physical registers. This ensures that there is a unique reference to every operand and output which guarantees that the OoO execution does not cause an invalid state. Secondly, once the frontend has completed register renaming, it needs to dispatch the readied instructions to an appropriate issue unit, as well as the ROB.

Issue units are one of the core new features of an OoO system. In an issue unit, an instruction waits until its operands and the execution unit are ready. This is what enables increased parallelism compared to the superscalar design: An instruction that is not yet ready does not necessarily prevent the processor from doing meaningful work. If there are tightly coupled dependencies between instructions they would still have to wait for each other sequentially and there would be little benefit to the OoO changes, but this is not typical for the work most OoO CPUs perform. An instruction waiting in the issue unit gets woken up when the relevant operands are cleared and can then be issued whenever there is an available execution unit corresponding to the type of the instruction.

Depending on processor design, an issue unit might have several available execution units or only one unit to issue to. OoO processors are usually required to be able to handle floating point operations, and therefore a separate issue unit and execution unit for floating point instructions is usually added. This creates three distinct issue units: one for integer operations, one for floating-point operations, and one for memory operations. Whenever these finish their operation, they write back their results to the register file, that keeps track of the status of registers. Once all the registers that map to the operands for an instruction are ready, the instruction can be issued to an execution unit.

To ensure eventual correctness and deal with a myriad of technical challenges, OoO processors use a reorder buffer (ROB). All instructions are entered into the ROB at the same time as they are dispatched to the issue units. The ROB therefore stores instructions in-order and additionally keeps track of their completion status. The ROB will only clear an entry once it is at the head of the ROB and it has completed execution, which keeps the promise of eventual correctness: the functional changes to state are committed in-order.

The ROB also serves a critical function for handling several other challenges, such as mispredictions, flushes and ordering failures. When a misprediction occurs, meaning that a branch is resolved to have been mispredicted, the processor needs to be restored to the state before prediction in order to ensure eventual correctness. This requires knowing the state at the point of misprediction and being able to restore the architectural state to that state. This can be accomplished by storing the mappings and values of registers for register renaming in a register file, each mapping to a specific branch prediction, which can then be retrieved and applied to restore register values and mappings.

For an OoO system, the memory operation, which in the classical InO abstraction is usually confined to a single stage in the pipeline (assuming target address is known), is separated out into a separate execution unit called the Load Store Unit (LSU). The LSU functions in many ways as a combined issue and execution unit for both load and store operations. It keeps a queue of both kinds of operations and perform them as efficiently as possible with regard to the needs of the processor. This usually means a drastic prioritization of loads over stores, as stores can only commit when they are at ROB head and are rarely on the critical path.

Both of these characteristics make sense: stores can only be written to memory when they are guaranteed to not be mispredicted, and since they produce no new operands, they cannot stall new operations beyond later loads. The later loads problem is further circumvented by relaxing the store-load ordering and allowing loads to run ahead with potential forwarding from the store queue in case of an address match, and the occasional rollback when such an address match is missed. Again, this performance benefit is another key area in which OoO processors gain performance advantages over InO processors: the ability to reorder memory operations and not only prioritize clearing potential blocks, but even achieving faster memory operations through operations such as store forwarding, gives a clear performance benefit compared to InO.

A few new properties become apparent in a full OoO system, compared to the earlier systems discussed. As the amount of speculation drastically increases and reordering becomes common, there is a considerably larger amount of wasted work performed. With a full OoO processor and with the relative frequency of branches, there is a larger likelihood that work will be performed along a mispredicted path, which will then have to be squashed. The amount of work wasted can be notable in certain circumstances, such as when a branch instruction has a long latency until being resolved and a large amount of work has to be squashed. However, these mispredicted paths

of work will not affect the architectural state as it will be rollbacked, only the micro-architectural state which might have had properties such as cache state inadvertently altered.

The mispredicted work is part of the reason why OoO processors are considerably less energy efficient than InO cores, but the main reason for wasted energy is attributed to the increased overhead. The increased amount of buffers for storing instructions and register files, as well as cross-wiring so that any execution unit can writeback to any issue unit, makes for a high amount of static electricity usage. After the end of Dennard's Scaling around 2006, this large amount of wiring has been a key factor in the increased energy usage of processors. The increased energy usage has also resulted in unique modern design challenges, such as dark silicon [6], however covering these topics falls outside the scope of this work.

2.2 The Berkeley Out-of-Order Machine

The Berkeley out-of-order machine (BOOM)[2] is a fully functioning OoO core, written in Chisel and integrated into the Chipyard ecosystem [7]. The BOOM is open source and is under continual development by a team of PhD students at Berkeley. The BOOM is not competitive in terms of performance, operating at around an order of magnitude lower frequency than modern processors. In addition, many parts of the BOOM remain greatly unoptimized, in terms of memory handling, instruction issue and more. Despite this, the BOOM remains the most valuable hardware platform alternative for computer architects without access to the resources provided by the big design houses such as Intel, AMD, ARM, etc. This section will highlight the key components of the BOOM and some of the features that will be used later.

Although the paper introducing and explaining the SonicBOOM [2] (the current version of BOOM) goes into some detail on the micro-architectural implementation, it does not cover sufficient details for this work. Many aspects of the BOOM are not represented, and much of the documentation [8] remains out of date. In addition, to help preserve the longevity of this work, it is helpful to have a frozen description of the BOOM for how it functioned at this time.

2.2.1 Parameters

The BOOM is built using a parametrized system, in which it can easily be adjusted for various configurations without designers needing to make functional changes. These configuration options are set using parameters defined in the Chipyard [7] configuration files in a cascading manner. For the most part, the BOOM manages to maintain a parametric style within its implementation, but some functionality is implemented differently depending on configuration parameters. However, these do not affect any of the components that we are interested in, and relate mostly to differences between handling scalar and superscalar processors.

Many parameters are used to configure the BOOM, but only a small selection of them are relevant for this work. For the rest of them, it is sufficient to be aware of the fact that these sizes can be configured using parameters, to prevent confusion when comparing different design sizes. The parameters which matter for this work are especially the fetchWidth, decodeWidth, numRobEntries, numLdqEntries, maxBrCount and the number of functional units we care about. Each of these will be explained in some detail.

FetchWidth and decodeWidth each define the two width sizes for the BOOM core. The fetchWidth is the maximum amount of instructions that can be fetched within a single cycle. It remains considerably larger than the rest of the width of the core. This is in order for the fetch part of the pipeline to always be considerably ahead of the rest of the pipeline, and therefore much more rarely cause a stall when a change in control flow or similar occurs. It is common for the fetchWidth to be noticeably larger than the rest of the core to ensure that the fetch buffer is only rarely empty. The size of fetchWidth varies from four for the smallest design, all the way up to eight for the largest design. The ratio of fetchWidth to decodeWidth goes from 4:1 to 2:1.

The decodeWidth parameters determine what we refer to as the width of the core. This is the maximum amount of instructions that the BOOM can complete in a cycle and sets an upper bound on the IPC for a processor. For the BOOM, this goes from scalar (one) for the smallest BOOM design, all the way up to superscalar of width four for the largest BOOM design. This is fairly small compared to leading CPU cores, which peak at around a core width of eight [9], but is still suitable for most research purposes.

The rest of the parameters exist for intuitive configuration options. The number of entries for the load queue (LDQ), the maximum number of ROB entries, as well as max branch depth all scale with the basic decode width. This is intuitive, as there is little point in having a larger ROB than you can ever fill, or more branches than can be issued. There will always be a balancing act between these parameters, in which various ones will appear to be the bottleneck, depending on the profile of the benchmark being executed. There is little research done on the optimal balancing for a system such as the BOOM, but it seems based on general assumptions about performance. As around 1/10 of all executed instructions are typically branches and around 1/4 of the instructions are typically loads,[5] it is expected for the max branch depth and the number of LDQ entries, respectively, to be roughly be their fractions of the ROB entries, which they are.

Finally, the number of functional units refers to the number of execution units that can perform either integer or floating point instructions. Each of these functional units are bundled together into groups, in which dispatched instructions wait to be issued as soon as their operands are ready and there is a ready execution unit. The exception to this is the load store unit (LSU), in which instructions are always issued as early as possible and the frontend instead stalls if there are no available slots in the LDQ (similar for stores). Together, the functional units must be able to satisfy instructions at the rate at which they are dispatched by the frontend. Since the vast majority of instructions are integer instructions (at least for the traditional CPU benchmarks), this usually means that there are an equivalent amount of integer ALUs to the core width. In addition, there is a smaller amount of floating point ALUs, usually at around half the core width. The memory width (number of memory instructions that can be issued concurrently) is also half the size of the core width.

2.2.2 Frontend

We cover the frontend only briefly, as it is not particularly relevant to our work. However, it is valuable for one specific phenomenon concerning the need to stall the frontend to prevent overwrites, which will be discussed in section 3.1. In general, the frontend can briefly be considered as being a 4-cycle stage, in which instructions are fetched, decoded, renamed and dispatched. The rest of this section will now delve into some more detail.

The BOOM utilizes an aggressive instruction fetch that fetches considerably more instructions than can be decoded, usually twice as many, in order to fill up the fetch buffer. The fetch buffer is filled and then time-wise arbitrated on for the next stage. The fetch buffer feeds a decoder, which can have a width of up to four for the so-called Mega BOOM, but for our work with the Large BOOM it has a width of three. The decode-stage decodes all the instructions in parallel, transforming them from instructions to micro-ops. These micro-ops are not detailed completely at this stage, as their operands are defined as architectural registers instead of physical registers. The next stage performs register renaming, which is followed by the dispatch-stage, which dispatches the decoded and renamed micro-ops to both the ROB and the issue queues for the various execution units. There are some more nuances to this process that we do not detail, but those are not be relevant for our work and are not discussed for the sake of brevity.

For the frontend stages, that is up to and including dispatch, it is most important to note that control instructions are decoded, and allocated slots in the branch mask. Information about control instructions is fully visible before dispatch, which will be relevant in section 3.1.5, in which we have to stall dispatching control instructions under certain circumstances.

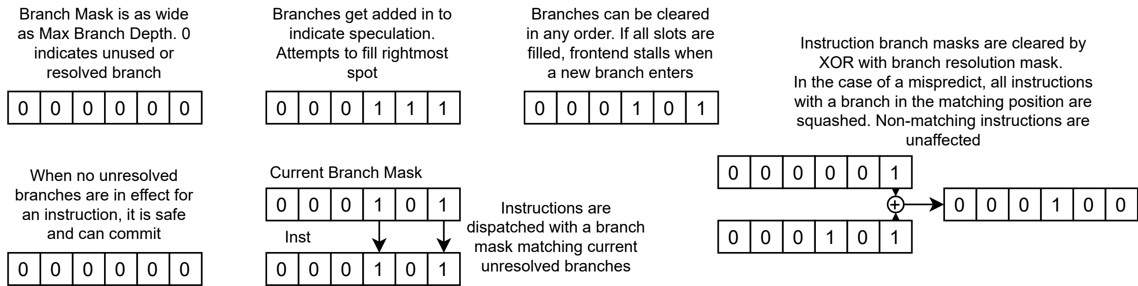

Figure 2.4: The behavior of the branch mask in the BOOM.

2.2.3 Branch Handling

The branch handling of the BOOM is core to this work, but is simple to understand. The BOOM uses a parameter to set the maximum amount of allowed branches, that is, the maximum amount of unresolved branches that can be dispatched at any given time. If the maximum amount of unresolved branches have been dispatched and remain unresolved, new branches cannot be dispatched and the frontend stalls. Whenever there are available branch slots, the frontend will attempt to dispatch as many branch instructions that it can fit in the current cycle.

Branches are handled and monitored through the use of a branch mask, as shown in Figure 2.4. This branch mask is a bit-array the size of the maximum amount of branch instructions, and indicates a 0 for a resolved or unused branch slot and a 1 for a currently unresolved branch. The branch mask is opportunistically filled and cleared, meaning that there are no guarantees regarding the structure of the branch mask. The branch mask can have an arbitrary amount of holes, and the only way to check if it is full or empty is to check each entry.

Within the BOOM, every inflight instruction is a micro-op. This micro-op has a branch mask with it, which indicates what branches this instruction is speculated under, if any. This branch mask is only used for two purposes in the default implementation of BOOM: When a branch is resolved to be safe, the corresponding branch entries in the branch mask are removed, using a standard XOR operation. This allows for new branches to be allotted for more speculation. The second purpose of a branch mask is killing mispredicted instructions. When a branch is resolved to be mispredicted, a mispredict mask is sent out and compared to the branch masks of all inflight instructions. If the branch mask contains the mispredicted branch, the instruction is killed.

We also note that although the BOOM sends out only the branch mask to the inflight instructions, extra information is sent to the ROB in order to reset its indexes on a mispredict. Included in this information is the ROB index of the mispredicted instruction, which will be the youngest instruction in the ROB after it resets its indexes. We say that this information is bundled with the branch mask, but it is issued one cycle after the initial branch mask is issued. In addition, when there was not a mispedict last cycle, and there was at least one branch resolved, this information is instead used to indicate the ROB index of the youngest resolved branch.

The advantages of the branch mask being available everywhere is mostly one of simplicity: a misprediction has a very short resolve time as all instructions can be killed within one cycle, and it is always easy to check whether an instruction has outstanding branches and whether it would be killed by a misprediction. In addition, this also makes it easy to make changes, as branches are always explicitly available and intuitive.

However, the globally available branch mask incurs a considerable storage overhead. The ROB can store up to 96 entries when working on the LargeBoom configuration. The same configuration allows for a maximum amount of 16 branches. This means that the ROB alone will require 1536 bits to store all the branch masks. However, these branch masks are also included wherever else the instructions are within the execution stage, resulting in twice this size at a minimum when considering the entire system.

The alternative is to use a sequence number counter and then compare the sequence numbers when

squashing. Since all younger instructions must be squashed during a mispredict in a typical processor, this can be accomplished by using a rolling counter with an extra single state bit, resulting in the much smaller cost of log2(Entries) + 1 number of bits everywhere. For the aforementioned example, this would instead be 7 + 1 = 8 bits per entry instead of 16 bits per instruction, resulting in a 50% reduction in size. This reduction in size becomes greater the more ROB entries there are and the larger the branch depth is. It is therefore fair to assume that this is used for modern CPUs where max branch depth is often more than 50 and ROB size is 256 or more.

For this work, the branch mask makes it considerably easier to make the BOOM to operate in the manner we wish, but there should be no fundamental difference between the approaches. We believe the insight produced here is to a large extent architecturally ambivalent, meaning that the insight is not limited to a specific architecture.

2.2.4 Execution Stage

The execution stage comprises the central part of the BOOM and is where instructions spend most of their active time. The BOOM possesses three primary types of execution units: the floating point units, integer arithmetic units and the LSU. The LSU is discussed in detail in the next section. For this work, there is less of a need to take a deep dive into the details of the other execution units, so we choose to give only a cursory overview of these.

The integer arithmetic logic units accept one instruction per cycle each and compute results within one cycle. As the majority of instructions within a typical program execution involve the integer arithmetic logic units, the BOOM is designed with an amount of these units equivalent to the width of the core, allowing it to satisfy coreWidth integer requests every cycle. Every unit also write back its results in the next cycle, allowing for fairly quick feedback for program code such as tight loops.

The floating point units are more complicated than the integer arithmetic units, as floating point operations are more complex. Within the BOOM, they are also pipelined, allowing for a new instruction to be issued each cycle, even if previous ones have not yet completed. This is critical for floating point benchmarks to be executed in a timely manner. It would be a poor design trade off to have as many floating point units as integer arithmetic logic units due to the larger size of the floating point units, as well as lower average prevalence of floating point instructions.

2.2.5 Load Store Unit

The LSU is the unit responsible for handling all memory operations within the BOOM. Both loads and stores get dispatched to the LSU and all memory requests in the BOOM are issued from the LSU. Within the BOOM, the L1 cache is custom designed and tightly integrated with the rest of the processor core, as are the MSHRs. However, the L2 cache and main memory are reached using the Tile and Link system [10], based on the Diplomacy model [11]. This means that the code for the L2 cache and onwards is not maintained by the BOOM team.

The LSU has been through several iterations and is still undergoing development at the time of writing this thesis, with a larger update to the memory system as a whole being well over a year into development. However, the current memory system is for the most part simple and reasonably intuitive.

The core part of the LSU is the arbitration unit. This decides at any given time what the LSU should spend be doing, based on pending requests and available resources. Incoming loads and stores are prioritized to be dispatched to the TLB, in that order. If there are no incoming loads or stores, any instruction that has previously failed to resolve in the TLB may be retried in the TLB, checking if the page walk has finished.

In addition to dispatching up to memory width instructions to the TLB, the arbitration unit can dispatch up to memory width instructions to the L1 cache, if there are loads that have translated

addresses. Recently translated loads from the TLB are prioritized for this purpose, but retried loads which have failed for various reasons will be dispatched opportunistically. If there are no loads to dispatch or retry, the LSU will attempt to write a store that is ready from the store queue. Although it might seem strange to issue stores when there is a performance benefit to holding them back as long as possible, this simplifies the logic of arbitration and makes the store queue less likely to overfill. Due to the lack of wakeup in the LSU, this is beneficial to prevent a bottleneck.

One of the aforementioned performance weaknesses of the BOOM memory system is the lack of wakeup for load. Memory systems usually keep track of the reason why loads fail, and set up a warning system to immediately wake up and issue loads when the reason for failure has been fixed. The reason for this is that loads often are on the critical path, and failed loads can have huge latencies. However, the BOOM does not keep track of the reason why loads failed, beyond some broad categories used for arbitration when re-issuing. Instead, it retries them with an age-based round-robin scheduling. This simplifies the logic considerably, but also means that one of the biggest bottlenecks of modern processing, the latency of loads, is not handled in a state-of-the-art manner. The section 4.2 discuss this and what it means for this work in more detail.

Loads and stores are respectively stored in the load queue and the store queue. There is no two-stage store queue for stores that have both address and data ready, as seen in earlier iterations, but a single unified queue that is updated with the incoming address and data. Both the load queue and the store queue have a large amount of fields that are used for memory arbitration, as well as exception handling and completion checking.

Stores are written into the store queue, and are attempted to be resolved in the TLB when they are incoming, unless there is also an incoming load. Data is written into the relevant entry through the store write port when it is ready. A store that has both data and address ready will be opportunistically issued to the L1 cache, unless the store queue is nearly full. If the store queue is nearly full, store write will take priority over most loads, except recent loads, to ensure that the store queue does not block the pipeline.

Loads are issued immediately to the TLB, and next cycle from the TLB to the L1 cache, assuming there was a hit in the TLB. If a load misses in the TLB, it will be frozen for a few cycles before being allowed to retry the TLB whenever there is opportunistically room for it. When a load is issued to the L1 cache, three possible results may occur: It can hit in the L1 cache, it can miss in the L1 cache and be put in a MSHR, or it can miss in the L1 cache and not be put in a MSHR. If the first event happens, then there is no issue and the data will be returned to the core and written into the relevant register.

However, if it misses and is put in a MSHR, the data will automatically be returned only after the MSHR acquires the data from the L2 cache (or lower). This will also automatically free the load from the load queue when the MSHR declares that it has acquired the data. If the load request misses in the L1 cache and cannot be put into a MSHR, either because the MSHRs are full or because the load has been killed for other reasons, then the load will need to be retried all the way from the LSU, at which point it might either hit in the L1 cache or be put into a MSHR, or delayed once more. A reason for the load being killed in the L1 cache is if a request for the same target address is already in an MSHR, but there are many other reasons that will not be covered here, as it is not conducive to this work.

Finally, a load can also be forwarded from the store queue if their addresses match. This check occurs concurrently with the dispatch to the L1 cache, and load requests are killed mid-cache access if a newer entry is found in the store queue. This allows for lower chances of load store ordering failure and allows for a lower strain on the memory system. This check is also performed whenever a store is written to the L1 cache, to see if there are any already completed loads that match an older store. If so, there is a load store ordering failure, and the affected instructions need to be replayed with the correct values. This causes the BOOM to begin rollback to restore the correct state.

2.2.6 Reorder-Buffer

The Reorder-Buffer (ROB) is the final stage of the BOOM pipeline and the stage in which the OoO micro-ops in the execution stage are committed in-order. The ROB is designed in a simplified manner, and is not well-optimized. Similar to other commit stage designs, it both keeps track of the original order of instructions and ensures that architectural state is restored in the case of erroneous execution. In addition to this, the ROB keeps some information on the current state of the core and is also in charge of handling certain special instructions.

The ROB is split into a number of banks equal to the width of the core, that can all be accessed concurrently. There is a ROB tail which indicates which row new entries will be fitted into. This tail is incremented every cycle if there are valid instructions that were inserted that cycle. Instructions are assigned into a bank without necessarily filling the banks in-order. This means that instructions can be fitted into any of the banks, with holes between them, if that is how dispatch has transmitted the instructions to the ROB.

The banks of the ROB are all constructed with the same properties and are automatically connected to the forwarded micro-ops according to coreWidth. Very little logic is uniquely implemented for each ROB bank, meaning that most of the logic of the ROB is unified and checks all banks simultaneously, while the logic in the banks only handle the entry and clearing of instructions. For all practical purposes, the banks are therefore identical and there are no dependencies between them. The chronological ordering of the banks however is critical, in that despite potential holes between the banks, older instructions will always be placed in a lower bank within one cycle. In other words, rows are in-order with respect to each other, and banks are in-order with respect to each other.

2.3 Hardware Tooling

In order to be able to understand later parts of this thesis, it becomes necessary to understand parts of the ecosystem and framework that were used to develop the hardware prototype. As understanding and experience with these systems also took up a considerable amount of time in the earlier parts of this work, we also deem it important to illustrate this for a sense of scope.

2.3.1 RISC-V

RISC-V[12] is a reduced instruction set architecture introduced in 2010, with the goal of learning from over 30 years of ISAs to create a more robust, logical and scalable ISA. RISC-V is now one of the most popular ISAs, with wide academic and hobbyist backing, as well as growing commercial interest. The advantages and design considerations of RISC-V are complex, but we choose to highlight the most important.

As by the name, RISC-V is considerable smaller in terms of supported instructions, compared to ISAs like x86_64, making hardware design using it considerably easier. In addition, the RISC-V design supports both 32 bits and 64 bits, with a wide variety of modular properties that allow the use of different instruction sets according to design needs. These include vector extensions (V extension), compressed instructions (C extension), and special design considerations for embedded systems, such as RV32E. These design features make RISC-V more universal than other ISAs.

In addition to the modularity and simplicity of the design of RISC-V as a whole, the design of its individual instructions and special rules are considerably more consistent, logical and simple than other designs [13]. Its instructions are designed to incorporate address targets, dependent registers and other properties in a consistent and logical manner, as to avoid complicated decoding. It uses the RISC-V weak memory ordering (RVWMO), and offers no macro-ops.

BOOM is built to satisfy the RISC-V ISA [13], and as such, all applications and programs that are to be executed on the BOOM must be compiled to a RISC-V binary. Therefore, certain

applications and systems that have not yet been ported to RISC-V are not currently available for use. There is currently no universal x86 to RISC-V translator.

2.3.2 Chipyard

Chipyard [7] is another project developed and maintained by a team at Berkeley, but having received very widespread support and usage, with dozens of other projects using it as their base platform. Chipyard is itended to work as a shared building dock for all chip related projects, allowing for relatively seamless integration between different systems. We outline the key properties of Chipyard.

Chipyard contains a large amount of functionality to support a variety of chips and chip-related projects. Most notably, it is the development platform for both the BOOM and the very popular RocketChip core. The RocketChip core is an open-source in-order core that has been the base building block of a variety of other core designs. In addition to the hosted cores, Chipyard employs an impressive system of configurations that enables users to define a cascading set of parameters that makes running the cores with various systems and for various programs considerably easier.

In addition to the cores, Chipyard is host to a wide variety of components, including the SiFive cache system, which is highly popular among chip developers. Developing all the necessary hardware components for a new system, especially cache components, is prohibitively time-consuming and expensive for most projects, so Chipyard allows for easy and free integration into a set of components for most general functionality. These components are integrated through a universal system known as Tile-and-Diplomacy, in which components function in an isolated manner, Tile, communicating through a well-defined set of interactions, Diplomacy. Tile and Diplomacy therefore allows for projects to quickly get off the ground and start prototyping, without having to invest expensive resources into peripheral requirements. Although the system and its components are unlikely to be viable for large-scale commercial projects, they provide a solid starting point, although comparable alternatives exist, such as GRLIB provided by Gaisler [14].

The RocketChip is similarly very popular as a baseline for expansion and modification. RocketChip was used as the starting design for the BOOM, and many of its features are still alive and well within the BOOM core now. RocketChip offers comparatively good performance for an open source processor, and as an in-order core it remains a promising choice for many embedded applications.

Chipyard also supports several of the most frequently used tools for debugging and verifying hardware designs. Although many tools remain proprietary, for smaller projects, these tools provide a sufficient base by which to start and maintain hardware designs.

2.3.3 Verilator

Verilator is a simulator, used to perform debugging and testing of RTL. Although Verilator is a simulator, we will refer to it as part of the hardware tools, similar to FPGAs, as it is dependent on hardware descriptions and simulates those. Verilator is several times slower than running natively on an FPGA, but gives many more options in terms of available data and configurability. For these reasons, it is usually used as the first step in the debug phase of development. However, due to its very slow speeds, it is usually not feasible to perform full testing using only Verilator.

2.3.4 FPGAs

FPGA stands for Field Programmable Gate Arrays and is a unique piece of hardware that can be used to model hardware. Although a full explanation of FPGAs is outside the scope of this work, it is necessary to understand both how FPGAs are instrumented, what their constraints are and how FPGAs function compared to actual hardware.

FPGAs can be used as prototypes of hardware implementations. FPGAs consist of a series of

lookup-tables (LUTs), that can be configured to mirror basic hardware components such as logic gates. By combining these, one can create a system with the same functional design as the actual underlying hardware, instead of simulating and abstracting. Typically, a complete description of the functional design of the hardware component is created in a hardware description language (HDL). This design is then transferred to the FPGA according to a process known as synthesis. Synthesis involves transforming the HDL into a level of abstraction known as register-transfer level (RTL), before placing, routing and bit file generation. RTL is a level of abstraction with explicit definition of gates and flip-flops. Placing, routing, bit file generation, and other steps not covered here, all work to make the design realized in the LUTs of the FGPA, as well as attempting to make certain parameters such as timing and size as optimal as possible.

FPGAs are a great resource for hardware design, as they can be used to implement most types of logical circuits. However, they are not as efficient in these implementations as actual taped-out hardware designs: The generic nature of the design of FPGAs require additional resources in terms of the individual components such as the LUTs. In addition, considerable resources have to be put into connecting and simplifying these components. This is done through routing and placing algorithms, that are designed to be considerably faster than traditional hardware placement algorithms, but give a weaker guarantee on optimal placement. The process uses some amount of randomization and as such is non-deterministic in terms of both placing and timing. Generally, the timing results from these algorithms are similar between iterations.

2.3.5 FireSim

FireSim [15] is a platform developed by a Berkeley team to allow for easy instrumentation of FPGAs, with a focus on the Amazon web platform. FireSim has many features, but most importantly provides a stable and configurable platform to manage and execute FPGA runs. This saves us a considerable amount of effort when having to instrument the FPGA cards directly through other proprietary tools such as Vivado, as we are instead able to instruct FireSim to do it for us according to the configurations provided within Chipyard, with execution being directly available through a shared interface.

As mentioned, FireSim is based upon a specific FPGA card that is available and offered as part of the Amazon web package, impressively offering an affordable platform by which to experiment without having to have the capital necessary to buy FPGAs. However, if FPGA hardware is available, it is possible to adapt the card-specific limitations of FireSim to work for a series of other cards. David Metz from the EECS group [16] at the Norwegian Unversity of Science and Technology has configured this for a cluster maintained by the Department of Computer Science at the same university. This cluster is what we use for running the BOOM on an FPGA card.

Together with FireSim, we also introduce FireMarshal. FireMarshal is a build tool which helps build workloads to execute on FireSim. For our purposes, we will be using it to build an image that launches a minimal operating system based on a Linux distro, and then lets us manually direct it to conduct workloads. FireMarshal is designed to work with FireSim and aids in launching images in a stable manner, with minimal user input. With an operating system to interface through, we are able to execute experiments in custom manners, to collect specific data.

2.3.6 Hardware Tools for the BOOM

In this section, we cover some of the other properties that will be referred to later in this work, namely the differences between debug runs and FPGA runs, as well as properties such as SynthAsserts, and what a run-harness is.

When developing for hardware, there are different ways in which to run the HDL code that has been developed. At the least abstract level, it is possible to convert the code into components, generate schematics and develop a physical ASIC, but this is a very complicated process. For testing purposes, it is very common to run the code on FPGAs to ensure that performance is as expected and no emergent errors appear. While still in development, it is typically more beneficial to utilize

Verilator to perform trial runs, as this avoids the overhead of synthesis. When running on Verilator, it is also possible to run in a debug mode, in which waveforms holding the state of components are available. This can allow engineers to figure out which components are malfunctioning and why, while FPGA runs typically have a smaller amount of information available.

SynthAsserts are a feature utilized in the BOOM to allow for more efficient debugging while running on an FPGA. When running in Verilator, there are certain methods for sanity checking the state of the system. One of these is asserts, which verify that a certain statement is valid, and will stop the system with an error message if not. These usually have no functional bearing on the design when it is synthesized on an FPGA as they are removed as part of the syntheis process, but synthesize asserts (SynthAsserts) maintains the functionality also when running on an FPGA. However, the assert statements utilized within the BOOM are more used as a guide for debugging than for a consistent indicator of invalid state. As a result, some assert statements are imprecise and may trigger when running longer benchmarks, despite no error having occurred..

A run-harness is a wrapping script that manages interactions during execution, in order to alleviate the need for extensive user input and configuration. If most of the configuration options are static across all tasks performed on a system, then it makes sense to create a run-harness that sets these statically and leaves only the variable options configurable. Similarly, a run-harness can be used to manage interactions between Verilator, FPGAs, synthesis and more, to chain together several steps of a single process, such as compiling new RTL and executing a benchmark.

2.4 The gem5 simulator

The gem5 simulator [17] is the leading computer architecture research tool and has been under development for many years. It consists of a suite of tools, features, extensions, collaborative projects, instruction sets and architectures. Covering every aspect of gem5 is outside the scope of this work, but we will provide a relatively comprehensive discussion and explanation of the relevant features for our purposes. This section will cover what gem5 is and how it works, and discuss some of its well-known strengths and weaknesses.

2.4.1 The origin and purpose of gem5

The gem5 simulator was built as part of a mutual effort between the developers of the m5 simulator and the developers of GEMS [18]. The simulator was an attempt to bring together simulator efforts to better create one powerful tool, than many smaller half-finished ones. The gem5 simulator has since 2011 received regular updates [19] and development, and is now at the forefront of computer architecture research. This is in large part due to community favor and support, as there are large amounts of community projects both utilizing and developing the gem5 simulator every year. Common architectural research involves tool development, such as gem5x [20], which also get publicity with some regularity.

The ability of the community to develop and maintain gem5 has created a plethora of resources to allow for development and research, but like many research tools, it is only utilized by a few thousand users and requires a large amount of experience and technical understanding. The design considerations for gem5 also remain incredibly complicated, as the tool is utilized for a wide variety of purposes, such as academic research, design exploration, early-stage prototyping and iterative workflows. In addition, these features need adequate support across a wide variety of components and ISAs, resulting in a very complex code base. The maintenance of gem5 is hefty as well, as hundreds of thousand of lines of code require a huge amount of library dependencies, that may have updates that break gem5 functionality.

For the purposes of this work, it is only necessary to look at a subsection of everything gem5 offers. We focus on the purposes of academic research, especially realism and implementation details. The purpose of gem5 is to strike a competent trade-off between ease-of-work and realism that allows for interesting design exploration and research to be performed without the need for complicated

hardware prototyping. In addition, the software nature of gem5 enables it to more easily monitor complex states of execution and create statistics, a process which is considerably harder and more expensive on hardware and hardware-like systems.

2.4.2 A cycle-accurate simulator in software

The gem5 simulator is a cycle-accurate simulator, which makes it both deterministic and highly accurate. The basis of a cycle-accurate simulator is to imitate a design by simulating every interaction between its components, usually by a token-passing system and an atomic invocation of functions. This is in contrast to more general models, that correlate relationships between properties and their impact on a system. The advantages of cycle-accurate simulators are multifold: It is considerably easier to trace and follow what impact any modifications to the system have; the state of execution is available and consistent; and the work is considerably more likely to catch any unintended side effects or emergent properties.

The disadvantages of a cycle-accurate simulator comes in the form of its slower execution speeds and the increased complexity in designing and extending components within such a system. Although gem5 is considerably easier to work with than making fully functioning hardware prototypes, it still requires a greater amount of technical know-how than conventional modelling systems. There is no ability to simply make a correlation or change model parameters based on intended changes to the system and see the resulting changes. A component has to be implemented functionally in order to change the system.

There are, however, huge benefits to working within a software context instead of a hardware context for these implementations. Useful data structures such as lists, queues, and stacks are all trivially implemented in all but the most basic of programming languages. As gem5 utilizes C++, there is a large suite of features available providing robust and easy-to-use implementations of these data structures. No equivalent exists within real hardware, in which trivial implementations such as a cyclic array can prove to be complex. More information is available in section 3.1 on these sorts of challenges, which, although not difficult for experts, simply do not exist when working at a higher level of abstraction like C++.

Additionally, there is less need to model the detailed hardware implementations. An example of this is the availability of useful features such as searches, vectors and logging enabling more efficient development and debugging of new features. This prevents system crashes in situations which would normally cause a hardware to not function, and the nature of software development also gives useful stack traces that indicate where an error occurred in the event of a crash. These features combined enable researchers to approach new designs with a problem-oriented mindset, instead of having to focus on details and complicated hardware interactions. The ability to make a component act as it would as a hardware prototyped while being implemented entirely in software methods can give a significant reduction in development time.

2.4.3 Parametric, components and statistics

The gem5 simulator uses a system of configurations to more easily allow for varied testing and exploration. Based on configuration files, it is possible to have a wide varietyt of components, such as caches and memory units, and also configure the parameters for these components, such as size, clock speed, and even voltage. This combination of component selection and parameter configuration makes it trivial to explore a wide variety of design configurations and see how changes to the system are affecting performance and other results.

Parameters are normally defined with a default value that can be overwritten with a set value if desired. This removes the need for users to create a full configuration file to get started, but can also cause erroneous assumptions about how the system is really configured. An acceptable default value will naturally assume the most common use case and if the current work does not fit this common use case, there is a risk for inaccurate configurations. This is usually only mildly worrying, but can cause inaccurate reproductions of previous research, as comprehensive lists of

configurations and research artifacts are not usually available.

Parameters can be very different without causing dramatic changes to the design of the processors due to the software design method. While hardware implementations would require more complicated forwarding and dependency checks when dealing with larger core widths, within gem5 these are all defined as software methods. As the execution of a bundle of instructions does not happen concurrently across all instructions that are set to execute in a cycle, but rather atomically, the complicated dependency chains are considerably easier to manage within gem5.

Finally, gem5 can produce a large amount of statistics that can be used to monitor detailed performance. In a typical execution context, there is restricted insight into the performance of a program outside explicitly configured and manually collected data, or otherwise using performance counters [5]. Within gem5, there is the ability to get detailed information on the performance of various units, their average utilization levels and the occurrence of rare events. It is also trivial to extend and add new statistics, both to existing and new components added to the simulator.

These statistics can also be used to model more complex phenomena, such as memory-level parallelism (MLP). These can give deeper insights into the impact functional changes has on performance and can therefore drive better design. Since anything that can be accurately described in a software context can be used to drive a statistic, this allows for practically limitless and fine-grained data collection, which is not easily available in neither modelling systems nor hardware prototypes.

2.4.4 Pipeline Design, the O3 CPU and Design Considerations

This section details the key pipeline stages of gem5, as well as some details of their design, including some properties that do not transfer well to hardware. The main purpose of this section is to give a detailed understanding of gem5 as required for this project, and explain in what ways these designs differ from both the theoretical underpinnings and actual hardware based implementations.

Design-wise, gem5 is challenged by supporting processor designs with notably different design considerations. Currently, gem5 supports in-order cores and atomic cores, as well as the out-of-order CPU (O3 CPU). The O3 CPU is designed to support multiple cores and threads. However, as we do not use these, for our purposes it is essentially a CPU-core, with some extra arbitration for some functions. This arbitration will always forward to the hardware components for the one active thread. The O3 CPU does not define any of the other processor settings such as caches, memory or similar, and is as modular as the other cores in respect to these properties. Each of these have different design outlines that are not compatible with each other, or at least superfluous. An in-order CPU has no need for a dispatch stage and as such, forcing this stage into the shared design space would weaken the realism and performance of that CPU. Due to the lack of overlap between these sorts of processors, gem5 defines only a few, general design rules at the highest level of shared design space.

The gem5 simulator chooses to limit most of the shared resources between processor designs to a series of interfaces to other components and a unified system for describing instructions. All other design considerations for the processor, such as the number of pipeline stages, other supporting structures and internal consistency, are left to the specific CPU implementation. The shared design includes a BaseCPU class that each of the CPU implementations must implement and extend, and is the manner by which the external simulator controller allows for simulator progress.

It is also important to note the shared design of instructions: Within the CPUs, all instructions exist as a memory object with a shared pointer that is passed through the pipeline. By accessing this instruction pointer, properties of the instruction can be altered instantaneously, and all parts of the processor that references an instruction always gets its latest state. This is in contrast to hardware, where such information would need to be transmitted across components and there would be no guarantee that instruction information is fresh. The shared instruction pointer consists of both a few default functions and a large amount of properties that give information about the nature of the instruction and its dependencies.

This work focuses on the O3 CPU, which aims to be a realistic approximation of a modern OoO

processor. OoO processors dominate the computing market for everything with higher performance than embedded hardware, and as such the O3 CPU is often the core of modern research considerations within computer architecture research. For the rest of this section, we will be examining the O3 CPU and its specific design properties.

The O3 CPU is the most detailed processor available within gem5 and is loosely based on the historically important processor Alpha 21264 [21], which has for a long period of time been a research object for the computer architecture community. Although the O3 CPU is impressive in its design, it is also in many ways antiquated and is as such unable to compete in design complexity with the modern processors that are released regularly in commercial markets. Thankfully, due to the advantages of software-based design, the O3 CPU can still approximate similar performance and behavior, although this might fail to catch some phenomena that occur within modern OoO processors.

The O3 CPU consists of 5 stages: fetch, decode, rename, issue/writeback/execute, and commit. The stages of issue, writeback, and execute are combined into one stage for simplicity. The number of stages in the O3 CPU is very low in comparison to modern OoO processors, which typically employ several more stages, pipelined to allow for higher frequency of the core. However, each instruction stage is essentially atomic in the O3 CPU, timing is handled with delays to responses instead of as a natural consequence of an instruction moving through the pipeline.

The fetch stage handles acquiring new instructions, as well as thread arbitration in multi-threaded runs of gem5. Multi-threading is not relevant for this project. The fetch stage is where the instruction reference is initially created. This reference is globally used whenever information about the instruction is needed, and is freed only when the instruction is completed, committed and no more copies of its reference are used. Fetches also handle incoming branches, making a decision to branch based on the branch predictor. The branch is resolved in the execution stage. Fetch can be squashed as a result of signals further down the pipeline and can also stall when a later stage is no longer able to accept instructions.

The decode stage adds more information to the dynamic instruction, as well as doing some minor work with unconditional branches. Similar to fetch it can stall and squash depending on later stages of the frontend. Although decode is typically an important stage of modern processors, it is much less relevant within a software context, as the translation between binary signals and instructions is completely trivial, while it is a complex challenge of mixing wires and format when dealing with physical components.

The rename stage ensures that there are available registers to translate between architectural and physical registers, and to handle the serialization of certain dependent instructions by stalling the front-end until the back-end has drained the relevant instructions. The rename stage is therefore the first stage that can stall the rest of the front-end. Renaming functions similar to how it would function in hardware, but the physical registers are still an abstraction within gem5.

The issue, execute and writeback stage is by far the most complex stage in gem5 and incorporates all the necessary complex logic involved in checking for dependencies, scheduling instructions and handling memory related operations. We will discuss this in more detail later.

It is also important to note the O3 CPU's handling of squashing and rollback. Within every OoO CPU, there needs to be a system for handling mispredictions and exceptions. For the O3 CPU, there is a less consistent design philosophy around how both of these should be handled, and the process by which squashing occurs is less clear-cut than the BOOM. In the BOOM, mispredictions kill all in-flight instructions, and exceptions are handled at the head of the ROB. The O3 CPU employs more forms of squashing, and it is less instantaneous. Within the O3 CPU, instructions are only squashed at certain checks, meaning they might still propagate through a system for a while after having been squashed. Although they are unable to affect state while they are squashed, their continued execution makes it considerably harder to fully understand the state of the processor. In addition, they can continue affecting statistics in the simulation, although their influence should be minimal.

In addition to squashing due to mispredicts, there are a many ways in which instructions can be

squashed and many sources that can initiate squashing. These include a need to squash threads or reissuing specific instructions. These result in a less clear understanding of how squashing occurs and increases the likelihood that certain instructions may seem arbitrarily squashed. This is not unrealistic in terms of how an actual high-performance OoO CPU would operate, as it would have a need to be able to squash instructions opportunistically without necessarily waiting for ROB head or similar.

2.4.5 The O3 CPU Load Store Unit

The O3 CPU employs a very complicated control flow whenever a memory operation is handled that passes through several layers in the processor. This is to allow for the highest amount of modularity with regard to both ISA and the memory requests. This leads to a highly unnatural execution flow that heavily obfuscates the process by which memory requests are selected for execution, created, issued and written back. This does, however, not matter from a timing perspective, as a complicated control flow in the software causes no changes on the actual time it takes for a load to be considered executed by the system. This section describes how loads are handled in the load store unit (LSU) and clarifies which implications this has.

For the O3 CPU, the naming of the system and its design is based around the two structures LDST and LSU, respectively meaning load store (queue) and load store unit. These structures are also referred to internally as LSQ and LSQ_unit. The reasons for this split has to do with thread handling, in which there is an LSU for each thread, but there is only a single unified LDST queue for the entire CPU core. The CPU core will most commonly offload instructions to the LDST queue when it needs to handle them, but the LDST queue will simply forward these instructions to the appropriate LSU depending on the thread ID of the current instruction. However, for memory operations, when a request is pushed through to the CPU from an initiated access (more on this later), the LDST queue does arbitrate on how to handle the instruction. In addition, the LDST queue defines the structure of all memory requests, as these definitions are shared for all LSUs.

Every memory request within the system contains a series of properties that indicate both what kind of request it is, what sort of properties it has, and where in the process of execution it is. For example, a load might either be a single or a split data request, depending on the amount of data it is attempting to access. These requests also contain properties indicating whether they have been issued by a prefetcher, whether timing requirements should be enforced, and if they are strictly ordered or similar.

The O3 CPU employs a separate load queue and store queue, but a unified interface to access the cache and memory system. At its most basic, the LSU receives instructions from the execution context, which it then installs into its queues before attempting to execute them. Similar to other memory systems, the execution of a memory instruction consists of two main steps: the translation of the virtual address to the physical address and the actual memory access. Both of these steps are timing-dependent and can take a variable amount of "time" to resolve, which is handled by delaying responses to the relevant request based on information such as parameter configuration and TLB status.

The memory access is the second half of the memory request. It involves dispatching the translated request to the cache system using the interface. The memory system can then handle the request in a variety of manners, depending on the type of access and the information in the request. More specifically, if a timing request is sent, it will calculate delay, update internal state and write back a request to the receiver stored in the request. More information on the memory system is found in the next section.

It is important to be precise about what exactly a load request is. In gem5, the instruction is what is originally transmitted to the LDST and inserted into the LSU, which creates a default LSQRequest, with a reference to the instruction. This means that technically, the instructions are not what is stored within the LSU, but rather requests that possess a reference to the instruction. This also means that the LSU can discard a request and let a load instruction be reissued to the LDST and LSU. The LSU will discard requests when it is necessary to resolve special behavior or

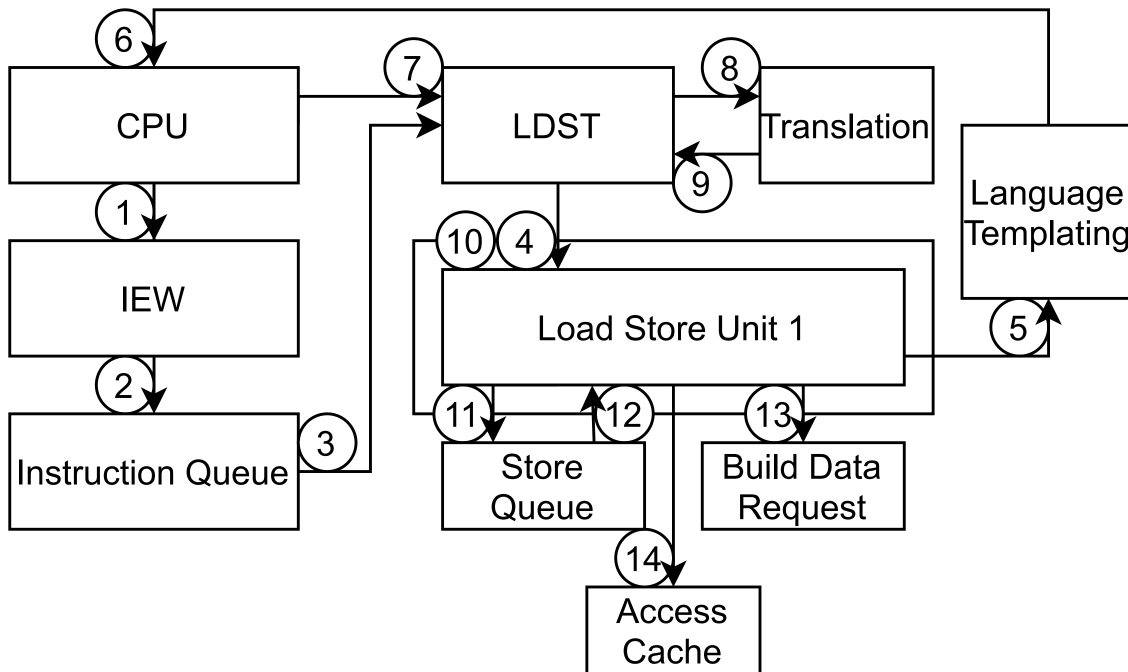

Figure 2.5: Most of the control flow for a load request in gem5. 1: CPU decides to execute. 2: Instruction selected from Instruction Queue. 3: Instruction is load, forwarded to LDST. 4: Load forwarded to thread-correct LSU. 5: Access initiated, language templating invoked. 6: x86 language templating pushes request back to CPU. 7: CPU pushes request to LDST. 8: LDST initiates translation. 9: Translation is returned. 10: Request is issued to thread-correct LSU. 11: Store queue is checked for forwarding. 12: If matched, result is forwarded, else, proceed. 13: Data request with sender state is built. 14: Data request is issued to cache.

when correcting for store forwarding.

In Figure 2.5, the control flow of a load is outlined. This flow is highly complicated, but it is necessary to understand it as we will be referring to it in detail section 3.2. We will now go through this step by step.

- The IEW (execution stage, functionally the core of the CPU) arbitrates which instructions should be executed and can decide that a load instruction should be issued. This only starts the first part of the memory request and does not need to be initiated for later stages after a successful translation.
- The LDST forwards the load to the relevant LSU for the correct thread.
- The LSU initiates the access and also communicates back whether any faults were encountered that indicate the need to reschedule the load
- The initiated access then goes through several references to allow for language templating. For x86, the three steps of initiate access simply forwards to initiate mem read.
- Initiate mem read pushes the request back to the CPU, which pushes it to the LDST once more.
- The request is checked to see if translation has been started. If it hasn't, it is now initiated. If the translation has started but not completed, no action is taken here. If the translation is completed, the next step is taken and a reference is made to the LSU to read.
- The LSU read checks the store buffer for any potential forwarding, i.e. if any known addresses of stores matches the load address. If it does not detect any forwarding, the request is assigned

a sender state to receive data responses, the request is built into a data packet and initiates sending a packet to caches.

- The LSU checks for availability in the cache and then issues the memory request. Depending on the response from the interface, the load will either be noted as having been successfully sent, or needing to be retried when the cache is no longer blocked.

As can be seen from the description above, the path that loads take is complicated and is made more difficult by gem5's commitment to being modular. The templating language in the middle can technically be of nearly any supported architecture, but the nature of having a single CPU that is shared between vastly different ISAs such as x86 and RISC-V makes for some unrealistic assumptions regardless of implementation. In addition to this, the repeated switching between the various execution stages does not necessarily realistically reflect how a processor would handle this, as the amount of wiring would be excessive.

Our presentation of the load control flow still leaves out some details in order to show the full pipeline in a comprehensible manner. Notably, it does not mention how faults and delays are handled, nor how strictly ordered loads work. Faults occur whenever a load is unable to proceed for an unexpected reason. This can be as simple as being a strictly ordered load that is not currently meeting ordering conditions, or a successful load later realizing it aliased with a store and therefore violated load store ordering. Notably, faults can occur during nearly any part of the execution. The reasoning behind this is that faults typically require some special handling by the processor that is performed when the faulting instruction is at the head of the ROB. Faults that the processor are not capable of handling or is not expecting to handle cause a panic exit from the gem5 simulator and results in program termination with the panic message.

Although a lower level might be returning a fault, a higher level part of the process might expect this fault and handle it or remove it without the fault ever reaching the top of the control-flow chain. The lower levels can also create a fault if the instruction fails a sanity check, even if it is not faulting and not going to when interacting with memory. Sometimes, these faults require that the load request be rebuilt, at which point the O3 CPU will discard previous work done on the request and wait to reissue until the load instruction reaches the head of the ROB.

In addition to faults, the loads can be rescheduled for a variety of reasons. The most common ones are that a load is aliasing with an older store that has not yet retrieved its data. In this case, the load needs to be reissued for data forwarding whenever the store operation is ready. The other common case is the cache not being able to receive a request, at which point the request must be delayed until the cache is unblocked once more. Both of these cases are handled by moving the requests to their own separate queues in the instruction queue component, which ensures that when the instructions are ready to be executed, they will once more appear in the list of ready instructions the IEW stage can choose to execute from.

The dependencies between instructions are tracked by the memory dependence unit (MDU), which keeps track of all memory operations. The MDU is separate from both the LDST queue and the LSU and functions as a simple solution to allow for memory operation reordering and status tracking for the CPU. The MDU keeps a complete track of all memory instructions currently in flight and manages their state, i.e. whether they need to be issued, rescheduled, are waiting or are ready to be executed. The MDU also checks for possible violations of load-store ordering.

The actual reissue of load requests happens in the instruction queue, which keeps track of all instructions that are not currently being handled. Notably, both loads that are blocked and need to be rescheduled are added to their own queue in the instruction queue and can be added to the list of valid candidates when their respective conditions are met.

2.4.6 The gem5 memory system

The memory system of the gem5 is the last central component necessary to understand for this work. The main thing to remember about the gem5 memory system is that it is incredibly modular,

as gem5 components rarely make specific demands regarding caches and memory structures. In addition, even when specific configurations are required, such as the use of at least one level of cache, there is a large amount of flexibility in many properties of this configuration, such as the chosen replacement policies used for the caches. Making a general statement about these properties is of little use, and we restrict ourselves to exploring the most relevant of the available replacement policies at the end of this section. For the rest of this section we explore how the memory system is built up and how it works for our specific considerations.

For this work, only the gem5 classical cache system is considered. The gem5 simulator also has support for the Ruby cache system, which models coherence protocols and other cache properties in more detail, but the extra complexity is not necessary for the purpose of this thesis.

Firstly, the memory system is connected to the CPU, or more specifically the LSUs, through the use of ports. These dictate a common interface between the radically different internal structures that are employed within the various gem5 cache systems. This is a necessary part of maintaining such a modular system, but means that there is more ambiguity and less direct control over memory requests once they have been issued: the CPU is dependent on the signals transmitted back from the memory system to make appropriate responses and is unable to monitor even the L1 cache at a glance. This differs from more realistic implementations, where the L1 cache is usually quite tightly coupled with the CPU.

The interfaces between the CPU, the cache hierarchy and main memory supports three types of requests: atomic, timing, and functional, in which atomic and timing are mutually exclusive and cannot be supported in the same design. Atomic handle memory requests without timing considerations and immediately returns results, while timing calculate an approximate delay based on the events of the memory access and use that to decide when to writeback to the LSU. Functional requests are somewhat more complicated and performs a variety of features that are necessary to maintain cache state. Functional requests do not experience timing properties and are used within timing caches to imitate snooping for accessed cache lines as well as maintaining cache coherency. In addition, functional accesses can be used for other maintenance purposes.

An atomic request is designed for atomic processors, in which timing is not a consideration. These requests are not employed within the O3 CPU system, and we are unable to use them, as they are system-wise mutually exclusive with timing requests.

A timing request is the most common type of request for research, as it is the only one to realistically mimic how cache requests would properly work in cached systems. A timing requests initially receives the memory requests and tries to find a matching block for it. If it has no matching block, it checks its MSHRs to indicate whether a request has already been issued for a matching address. If not, it attempts to allocate an MSHR for this new request to eventually load the data into memory. If it is unable to allocate an MSHR because all MSHRs are in use, it writes back a failed memory request to the LSU, indicating that it needs to retry at a later point.

The timing requests propagate in this pattern downwards in the cache hierarchy until eventually reaching memory. The basic flow of a timing request involve arriving at a cache level, checking if it has any hits either in the cache lines or in the MSHRs, updating timing and then propagating downwards. The updated timing is a system in which each step calculates a current latency to delay writeback by, to more realistically mimic proper hardware caches.

Once a timing request has successfully found the cache line it is looking for, it schedules an event to writeback its value to the sender state registered with the processor. All timing requests are required to submit a sender state to function as writeback for loads, and the delayed timing on the writeback is the main mechanic by which the classical caches implement timing differences.

In addition to propagating and extracting the required data, the timing requests also interact with the replacement policy at each level of the cache hierarchy. Within the memory system, replacement policies are managed by a system of tags that are associated with each unique cache level. Similar to real processors, this allows for different replacement policies at each cache level, so that lower level caches that receive drastically different information about load history does not waste their performance attempting to follow an outdated LRU scheme. Rather, the lower level

caches can easily implement RRIP [22] while the higher level caches implement more conventional policies. As the tag is a separate entity from the blocks themselves, the user has full control over choosing how to program them. For example, under special circumstances, avoiding updating the tags and thereby the cache replacement policy is as simple as not invoking the method.

A final type of request utilized in the memory system is the functional request. Functional requests are used for everything else within a cache system that would not require using the memory bandwidth. For example, within cache coherency protocols, functional requests are used to snoop between caches and ensure that the properties of potentially shared cache lines are updated correctly. Functional requests can have a wide variety of properties and associated purposes, and covering them in detail goes beyond the scope of this work. However, whenever changes are to be made to the memory system with regard to functional behavior, a functional request is the natural solution, as it has less overhead and is more akin to using special signals.

2.5 Speculative Attacks and Delay-on-Miss

For this work, we have chosen to create a hardware prototype of a computer architecture technique in order to investigate the usefulness of hardware prototypes as a research tool. The continuous slew of speculative attacks and their mitigation strategies remain a hot topic that will likely stay relevant for the next decade. As such, we decided to focus our effort on one of the state-of-the-art mitigations and aim to reproduce it.

2.5.1 Speculative Execution and Attacks

Speculative attacks rose to prominence following the reveal of Spectre and Meltdown in early 2018 [23]. In the years following, several more varieties and attacks have been discovered and the battle to find the best mitigations continue. Although Meltdown is largely considered to be an erroneous implementation, Spectre exploits a fundamental aspect of modern processors and as such cannot be easily fixed. Spectre attacks continue to be discovered with regularity, and several of the proposed mitigations do not offer the amount of security that is desired.

The core premise of both Spectre and Meltdown has to do with the difference between micro-architectural state and architectural state. As discussed in subsection 2.1.2, the architectural state gives a guarantee of eventual correctness, but the micro-architectural state can leak secrets that the architectural state would eventually squash. This results in secrets being accessible after squashing and therefore information leakage. Both Spectre and Meltdown exploit this, although in different manners.

Meltdown was not as universal as Spectre and both Meltdown and its successors exploit a weakness with how exceptions are handled in Intel processors. When an abnormal memory request was handled, i.e., one that would cause an exception, that instruction would be marked as causing an exception, but it would still receive the data it desired. The exception would only be handled when the instruction reached the head of the ROB, giving a huge window for execution during which the secret could easily be stored into the micro-architectural state for later retrieval. This is an implementation error, as it is logically inconsistent: A failed memory request should not return data, there is no benefit to it doing so, and it leaks information. Meltdown was subsequently patched and although some later attacks exploiting similar exception-delayed responses have been created, they are much less ubiquitous than the Spectre attacks.

Spectre on the other hand exploits a much more fundamental property of modern OoO processors. These OoO processors are highly dependent on speculative execution to achieve their high performance. Within a typical program, around one in every ten instructions [5] is a branching operation that diverts control flow. Resolving a branch can typically be done within a few cycles at the fastest, and several hundred cycles in the case of depending on a long-latency memory requests. Thus, it is imperative to predict what the result of the branch will be and execute accordingly. On a wrong guess, all speculative changes must be squashed and execution must resume from the

correct branch target, but on a correct guess the processor has done a large amount of valuable work where otherwise it would be forced to wait.

However, this produces a key challenge exploited by Spectre: Boundary enforcement, such as those preventing out-of-bounds accesses, are implemented using these branches. Therefore, a misprediction of these barriers would technically violate the security guarantees that the architecture provides. From an architectural standpoint, this is not an issue, as the processor will eventually realize its mistake and prevent any of the changes from being observable in the architectural state. However, as discussed, this does not handle the micro-architectural state, which can be used to hide the secret that can then later be hoisted back into the architectural state, resulting in a successful attack.

2.5.2 Spectre

The most basic version of Spectre works in the following manner: A program mistrains the branch predictor to guess that a certain branch index, which maps to the same index as a boundary check on an array, is to be predicted as correct. The attack then attempts to access the desired secret by calculating the offset from an attack-controlled array and performing an access to that value. The processor does not yet know whether the desired address is inside or outside the array, and due to mistraining, it assumes that the access is valid as it uses the branch predictor to amke a decision. It therefore provides the secret to the attacker. The attack uses this secret as an offset to load a cache line in a region in which all cache lines exist in a lower level cache. The processor realizes it has mispredicted and squashes the execution, including the secret that the attack has used. However, the single cache line that was accessed remains in a higher cache than the other lines. By checking the timing delay on all cache lines, the user can figure out what the value of the secret is, by noting that the one faster cache line corresponds to the secret.

Although there are many technicalities and variations in Spectre attacks [24, 25], this remains the core of their behavior: Acquire a secret under speculative execution and store or transmit it in such a way that a squash will not hide it. Spectre attacks have not been observed in the wild yet, most likely as they remain an ineffectual form of attack compared to other attack variants that achieve remote code execution, and that Spectre attacks require a high level of technical understanding of processors. Nevertheless, if Spectre attacks remain unmitigated, they will eventually prove to be an excellent source of read gadgets that can be employed as part of other attacks.

The early Spectre and Meltdown attacks caused concern about their exploitability and rapid responses from developers of the affected systems. Software patches were quickly created and deployed on a variety of services. Web browsers, operating systems, Intel Management Engine [26] and compilers were all changed to help mitigate the impact of the Spectre attacks. Yet, it was clear early on that these mitigations would prove inadequate in the long term for two simple reasons: Firstly, they were very expensive in terms of performance, as they were entirely reliant on changes to software to prevent attacks. Secondly, they were unable to guard against all different attack variants to a sufficient degree, and new attacks would eventually over time therefore navigate around these rudimentary measures. Now, three years later, we know both of these reasons to be well-founded and some mitigations against Spectre are currently being employed in newer hardware generations [27], although details from industry are sparse.

2.5.3 Mitigations against Speculative Attacks

The mitigations against Spectre attacks have been varied, both in scope and strategy. The two core steps of the Spectre attacks are the access to the secret and the transmission of said secret. Therefore, some strategies aim to prevent the speculative access of the secret, while other strategies attempt to stop the transmission of the secret. Another approach has been to attempt to rollback any of the effects of the transmission as part of the squashing when restoring architectural state, i.e., rollbacking micro-architectural state as well, but this endeavor has not received much research outside CleanupSpec [28].

The first strategy was explored to a small extent. There were some promising results with stricter memory sequestering [29], which had been recommended multiple times before to mitigate against a variety of issues [30]. However, these changes would create a large overhead in terms of both processor design, and compiler and operating system support. In addition, they simply proved to not be comprehensive enough, as not all forms of Spectre attacks used these well-defined channels.

It was realized that most approaches of preventing access to the secret were simply not feasible, and those that were promising would not be comprehensive enough to put a stop to the Spectre problem. The resulting slowdown was comparable to that of reverting to in-order processing, which would never catch on in the consumer market, risks notwithstanding. Due to the design structure of OoO processors, it quickly became clear that the main approach would have to be to prevent the accessed secrets from affecting state in an observable manner, while maintaining the highest possible performance.

Therefore, the second strategy, which involves unobservable speculation, came to be the central focus of Spectre research. Even with this general strategy, there were a myriad of different approaches when deciding how to keep track of speculative state, secrets and transmission [31, 28, 32]. Although a full survey of the field is far outside the scope of this work, it is necessary to understand some core properties used to evaluate these solutions to appreciate what motivates this work. Notably, there is a wide variety within the field regarding how performative, comprehensive and transparent the various methods employed claim to be. We highlight these variations with examples of current strategies.

Initially, the original mitigations and also the focus of this work, Delay-on-Miss (DoM)[1], aimed to simply hide the transmission of secrets through the cache. This is not a comprehensive solution, as other channels are not mitigated, but the motivation for this approach is quite reasonable: Managing cache state is a lot easier than changing all possible channels of communication drastically. On most architectures, caches remain the fastest and most stable method for performing Spectre attacks. Theoretical attacks matter much less if, for a practical attack, they depend on minimal interference and full control over scheduling. Therefore, both InvisiSpec [31], Ghost loads [33] and later DoM aimed to do the best they could to limit caches as a method of transmission.

Other strategies aim to block all forms of transmission by monitoring all possible channels of transmission, and then prevent the processor from either transmitting while in a speculative context or handling values that were speculatively accessed and could therefore potentially be secrets. These strategies include work like Speculative Taint Tracking (STT)[32] and DOLMA[34]. However, although the appeal of a comprehensive solution as offered by both STT and DOLMA are highly tempting, they both require significant changes to the processor designs. In STTs case, it requires a full and thorough understanding of all possible channels, which is an incredibly hard guarantee to make, and indeed DOLMA highlights one of the channels that STT missed. DOLMA requires changes to how instruction scheduling occurs in all functional units by adding pre-empting. Although both of these approaches are comprehensive, their performance remains uncertain, as they have not been successfully reproduced with similar results. Performance metrics for STT, DoM, and also less comprehensive solutions such as InvisiSpec has varied greatly between reproductions.

In addition, there is some ambiguity in how to determine speculative state. Although most comprehensive solutions measure themselves against the benchmark of attacks known as Spectre-Futuristic, there is no guarantee on which forms of speculation can be exploited for attacks. Practically, there is a lot of speculation occurring within a processor, as there is a large set of instructions, especially floating point instructions, that can cause an exception that technically induces a speculative state. Yet, it is uncertain what properties can and cannot be used for attacks and as such, it is very hard to accurately and narrowly define what is a speculative state that might result in Spectre attacks. Too broad a definition causes lower performance, but too narrow a definition enables more attacks.

2.5.4 Challenges within Spectre Research

Having now looked at a few of the general ideas that are popular within this research, we look at two of the core problems facing these solutions. First is the matter of reproducibility: Although many of the solutions show great technical insights, the matter of performance is significantly more relevant for this research than is typical for computer architecture research. When the eventual hardware mitigations will be adopted by industry actors, they will to a large extent selected on which can guarantee the highest overall performance. Although computer architecture research is typically less focused on purely the numerical results, it is necessary to approximate how the actual solutions would impact the performance of a system. Currently, recent academic work show significantly lower performance when reproducing earlier mitigations [1, 32, 34, 35].

This issue is highlighted in works such as InvarSpec[35], which report drastically different performance metrics for DoM than the original paper. The explanation for this may be multifold, since the source code for the original implementation was not public, so the internal design is not available to examine, and later research has had little motivation in optimizing the performance results of earlier research. These two key challenges result in uncertainty regarding realistic performance and increases the need for more neutral reproducibility studies.

The second problem with these solutions is realism. As mentioned earlier, there has not been a Spectre attack in the wild and there is unlikely to be one in the near future. The mitigation strategies employed have raised the barrier enough that most computer systems have more promising attack vectors, and the scope of Spectre attacks has always been limited to reading secrets. To most actors, this is less valuable than remote code execution, which is still possible in a large amount of ubiquitous software. It is important to note that processors don't have to be entirely secure, just secure enough that Spectre attacks will not be considered a viable attack vector. This means reducing the amount of high-bandwidth, noise-resistant, and easy to develop attacks.

For this reason, there should be less focus on purely comprehensive solutions. Although Spectre attacks do provide a more universal approach to read secrets, the cases in which this can be sufficiently exploited to prove a viable threat model are more limited in scope. However, preventing the most dangerous channels is still necessary, as these can leak at sufficient rates to extract either a large amount of information or be reliable enough to steal important secrets such as encryption keys.

2.5.5 Delay-on-Miss

We now turn towards looking at one specific mitigation against Spectre attacks, namely the technique Delay-on-Miss (DoM), as presented by Sakalis et al. [1]. DoM is not a comprehensive solution, it does not aim to mitigate all the possible covert channels, and it does not eliminate indirect covert channels such as port contention and instruction reordering. However, DoM is a general technique that can be used for any side-channel that would leak information that leaks information through changes to observable states. This means that although we will only consider it as a means to prevent observable changes to the data cache, it could also be considered for other attack vectors, such as the micro-op cache employed on many x86 systems [36].

As a note on terminology here, whenever we refer to cache lines being transferred between levels of the cache hierarchy, we will refer to this as moving cache lines, and assume that cache levels are exclusive. For cache hierarchies, if a lower level cache is inclusive, when a request is made for a cache line in that cache, the cache line is copied to the higher level cache. For an exclusive cache, it is instead moved to that higher level cache. For simplicity's sake, we will consistently treat caches at all levels as though they are exclusive, and therefore refer to cache lines being moved instead of copied. Being aware of the distinction would be important for cache modifications, but the extra term does not add to the clarity of this work.

The basic premise of DoM is tracking shadows, as introduced by Ghost loads [33], and then preventing loads from making any observable changes while under a shadow. This prevents the changes to the cache state that allow for the retrieval of the secret after the squashing of loads. A

shadow is defined as any instruction that causes speculation that can result in a squash. Shadow tracking can be implemented to track any selection of these shadows, depending on which attack vector is most likely. For this work, we consider only the control shadows, that is instructions thatalter the control flow, but do so speculatively and have a chance to mispredict. These are called C-shadows and are what is exploited in the original Spectre v1 attack. By tracking C-shadows, we can therefore mitigate this version of Spectre.

Note that for the later sections of this work that implement DoM, we are *only* tracking C-shadows. Fully reproducing the original work would entail also tracking E-shadows, D-shadows and M-shadows. These shadows are for exceptions, memory dependencies, and data ordering, respectively. However, tracking them all would complicate the implementation work, and would also require an extensive investigation into less well-documented aspects of the BOOM, namely memory exception handling and excepting instructions. It is not necessary to pursue all shadows to mitigate Spectre v1, but tracking more types of shadows would give a more comprehensive defense against other variants. We discuss what implications this has on the value of our reproduction in subsection 5.1.2.

DoM is a complex system, and discussing it in a clear and unambiguous way is a challenge. To limit any misunderstanding, we therefore introduce the following terminology: The ROB holds all instructions, and stored with control instructions are a reference to its entry in a shadow queue (see Figure 2.6). An entry in the shadow queue is said to be active, if it holds a reference to a control-flow instruction (e.g., a branch) that has not yet been resolved, and inactive otherwise. A branch is said to be speculative until we know its correct path of execution, and is resolved upon knowing its path. A resolved branch can either be safe if it was predicted correctly, or mispredicted otherwise. We will refer to the head and tail of the shadow queue as SQ-Head and SQ-Tail, respectively. The SQ-Tail marks where a new entry in the shadow queue should go, and the SQ-Head marks the current head of the shadow queue, i.e., the oldest unresolved entry.

The release queue tracks loads and holds a reference that indicates when a given load will no longer be under any shadows, which we call the shadow tag. The shadow tag is equal to the SQ-Tail minus one at the time the load enters the shadow queue. This indicates that when the SQ-Head has reached the point where the SQ-Tail was at the time of entry for a particular load, there are no longer any shadows covering this load. In addition to the shadow tag, the release queue holds a reference to the index of the entry in the load queue belonging to the load being tracked. We will refer to this as the load queue index. Load queue entries are also extended to contain a value indicating whether the current load is speculative. We will refer to this as the speculative bit, or refer to the load as being speculative while this bit is high.

An entry is said to be freed when it is removed from any queue. This includes the shadow queue, the release queue and the load queue. An entry is said to be squashed if it is removed from a queue without being freed. An entry is removed from a queue if it is no longer observed by that queue, which means that it is not between the head and the tail of that queue. We often refer to shadow tracking being active: This means that there is at least one entry in the shadow queue that has not been released, resulting in new loads being entered into the release queue.

DoM is outlined in Figure 2.6. We will now go through how DoM functions in detail, using the figure as a reference. As mentioned, the system aims to track only control shadows, which can be considered to be all branch instructions for this work. Therefore, when LD_0 enters the ROB, the instruction is not put into the release queue, as there is no shadow cast over the instruction. The load is therefore not marked as speculative in the load queue, as shown by (a). When BR_1 enters the ROB, it casts a shadow. This is marked by it being assigned to the slot indicated by SQ-Tail (i.e., 3) in the shadow queue (b). The reference to the shadow queue in the ROB is set to the SQ-Tail (c), and then the SQ-Tail is incremented. The following load, LD_2, enters the ROB while a shadow is being cast. This assigns it an entry in the release queue (d), and the load is marked as speculative in the load queue (e). The shadow tag value the release queue entry uses is equal to the SQ-Tail minus one (f). This indicates that when the SQ-Head has moved past the shadow tag, the load should be released.

Shadow casting instructions enter the shadow queue in program order. When a shadow queue entry is cleared, we also say that its corresponding shadow tag has been freed. For a load that is dependent on that shadow tag, this means that all older shadow casting instructions have been

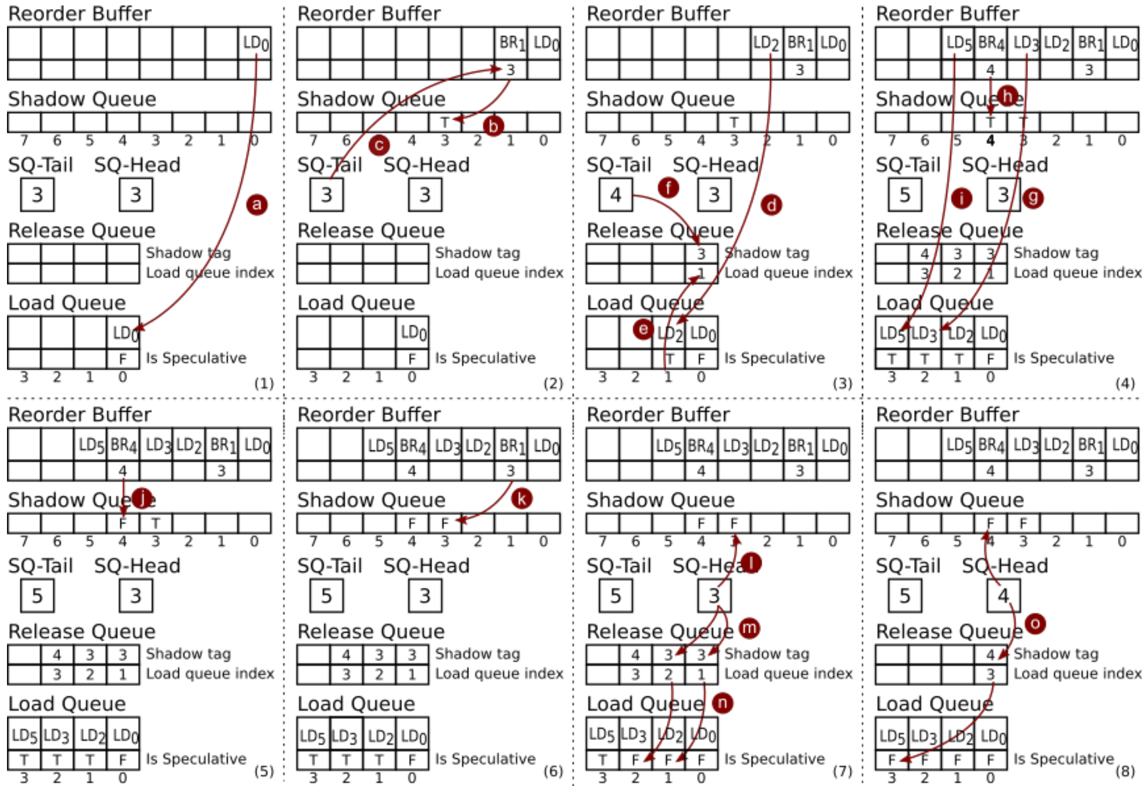

Figure 2.6: The core functionality of Delay-on-Miss, reused from the original paper with permission. The figure has been edited to adhere to our terminology, which deviates from the original paper only in that we refer to shadow buffer as a shadow queue. (a) shows a non-speculative load. (b) and (c) shows a control instruction casting a shadow. (d), (e), and (f) show a shadowed load. (g), (h), (i) show a second shadow being cast and a second shadowed load. (j) and (k) show the branches being declared safe. (l), (m), (n) and (o) show the shadow queue being cleared and loads being released.

resolved and marked as safe, so the load is no longer speculative.

When LD_3 enters into the ROB, it is also assigned an entry into the release queue, and its entry is marked as speculative in the load queue (g). It shares the same tag as LD_2, as no new branches has entered into the ROB between the two. When BR_4 is entered into the ROB, it is assigned slot 4 in the shadow queue (h) and the SQ-Tail is incremented once more. It is followed by LD_5, which is assigned a slot in the release queue with a different tag than LD_2 and LD_3, as it entered after a different branch, BR_4. Its slot in the load queue is marked as speculative (i).

BR_4 is resolved and marked as safe, and its slot in the shadow queue is therefore marked as inactive (i). However, the SQ-Head is not incremented, nor are any loads released, as the branch at the head of the shadow queue, BR_1, is still not resolved. Afterwards, BR_1 is resolved and also marked as safe, and its slot in the shadow queue is also marked as inactive (k). This causes the SQ-Head to increment, as the head entry of the shadow queue is now inactive (i). When it increments, it frees all entries from the release queue that were marked with shadow tag 3, thereby freeing two entries (m). These two entries then mark their respective loads in the load queue as no longer speculative (n).

As the head entry at the shadow queue, entry 4, is also inactive, the SQ-Head increments once more. This frees the release queue entries with a shadow tag 4, thereby freeing the remaining entry in the release queue (o). When this entry is freed, it marks the last remaining load in the load queue as no longer being speculative as well. At this point, there is no longer any active shadow tracking, no active entries remaining in the shadow queue or the release queue, and SQ-Head is equal to SQ-Tail.

The reason DoM successfully mitigates Spectre v1, is that it blocks observable changes from its transmission channel. Spectre v1 uses the cache to transmit the secret, by using the secret as a key into the cache and then observing timing differences. The timing differences occur because all possible affected cache lines were in the L2 cache or similar, and only the cache line that was retrieved using the secret as an index would be loaded into the L1 cache. However, as the secret is speculatively accessed, the load access is preceded by a branch that casts a shadow. Therefore, DoM would only allow the load to proceed if the cache line is already present in the L1 cache, and would block any load that would attempt to move a cache line from the L2 cache to the L1 cache. By delaying this observable change until the state is no longer speculative, DoM mitigates Spectre attacks.

Delaying observable changes is a general approach to mitigating the transmission of secrets. The key insight here is that only changes to the state of a component will create observable changes, forwarding the value itself does not directly leak the secret. This can still cause execution reordering that can create observable changes, but this is a significantly slower and less reliable method of secret leakage. Therefore, DoM gives a significant performance boost by allowing the value to be forwarded from a component, as long as this does not create any observable changes. This means that the state of replacement policies should not be updated and that, for the caches, no cache line can be hoisted from a lower cache to a higher cache, as this would create an observable timing difference.

DoM is different from techniques such as InvisiSpec, which used a dedicated buffer for these loads, thereby increasing the size requirements considerably, in that it merely exploits current processor design to disable caches as a side-channel. The only size ovearhead that DoM has are the structures necessary to support the tracking of shadows and the tracking of loads.

Chapter 3

Detailing the Implementation of Delay-on-Miss

The following chapter details the work undertaken to implement Delay-on-Miss on the BOOM and the gem5 simulator. The primary motivation to implement a state-of-the-art technique on these platforms is finding the differences in approaches, opportunities, and limitations they each present. In section 3.1, we explore in detail the challenges of creating a hardware prototype, and uncover several necessary modifications that were not discovered previously for DoM. After the hardware prototype, we use our understanding of the underlying hardware components to make a hardware realistic implementation of DoM in gem5, as detailed in section 3.2.

3.1 Implementing Delay-on-Miss on the BOOM

The central goals of this work is investigating both simulators and hardware prototypes as research tools. We have stated that we believe simulators can give misleading results when hardware limitations are not properly understood. Additionally, we suggest that by working with an actual hardware implementation it is possible to increase the precision of computer architecture research, as well as discover valuable insights that would have been missed at the higher level of abstraction of simulators. To illustrate these points, we will now detail the work implementing the DoM technique on the BOOM, and the challenges and oversights this work revealed, which we believe strengthen these theories.

3.1.1 A note on working with hardware

Before we delve into the details of the BOOM implementation, it is necessary to highlight some unique aspects of developing prototypes in hardware. This thesis is written from a computer science background, and therefore some familiarity with software development is expected. Hardware development is less commonly studied within computer science, and as such we deem it beneficial to highlight some less intuitive differences between the two.

Firstly, hardware is dependent on timing requirements. Timing requirements demand that all values must propagate to their next logical checkpoint, typically a register, within a certain amount of time. This is to ensure that state is deterministic between each clock edge. If a design does not meet timing requirements, it is no longer deterministic and will produce random results over time. It is normally possible to lower the frequency of a given design, but doing so reduces the performance of the entire system. As such, hardware should be designed in such a manner as to not worsen the most constrained timing path, in order to maintain performance.

With timing requirements, we also highlight a related problem: Preventing long cascades of de-

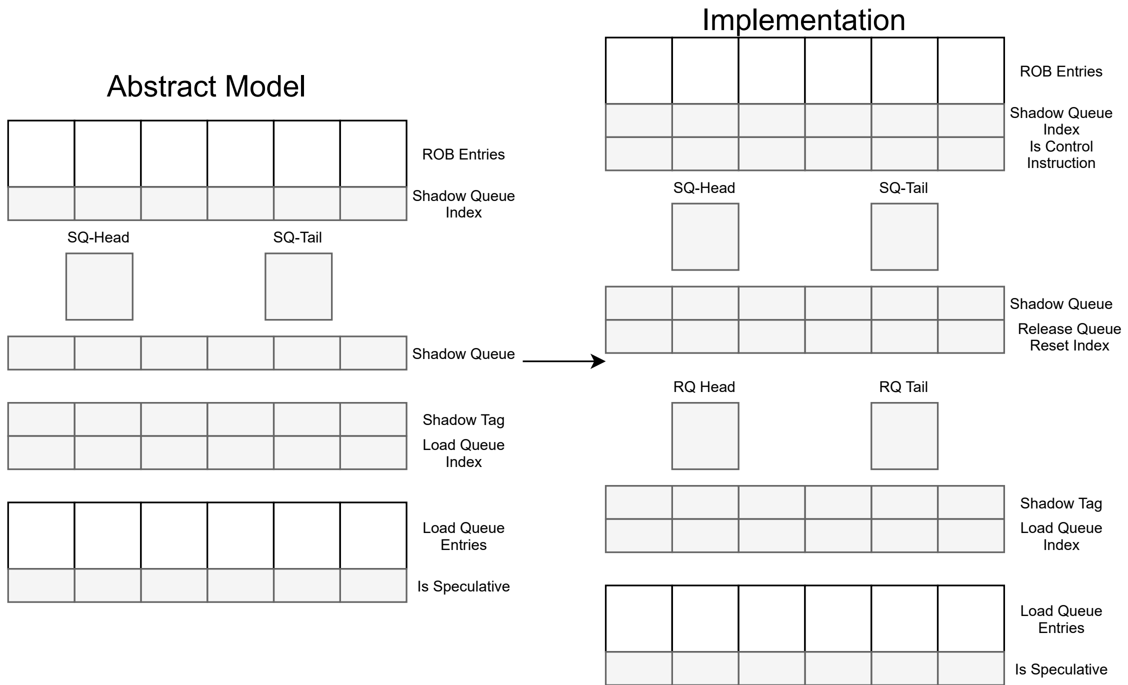

Figure 3.1: The original abstract model as envisioned by the Delay-on-Miss paper, and the eventual model actually implemented in hardware.

pendent information. In order to meet timing requirements, there needs to be logical checkpoints for operations that pass through a larger number of logical functions. For example, during a mispredict in a system, it may be necessary to rollback information structures. If one information structure possesses the data necessary to roll back another information structure, then that second structure is dependent on the first. If the second structure is rolled back in the same cycle as the first structure, then it might not meet timing requirements, as signals would have to pass through multiple structures. If the second structure rollbacks one cycle after the first, creating a logical checkpoint with the information, the system would spend one more cycle overall when handling a mispredict. Neither of these properties are desirable, but become necessary trade-offs within a hardware context.

3.1.2 Implementing Delay-on-Miss on Scalar BOOM

This section first presents the abstract technique and then the multiple steps necessary to use this technique to design a hardware prototype. It also discusses the necessary changes to prevent invalid execution states, and how to support other processor features such as flushing. The translation from an abstract model into a hardware implementation is particularly important, as it shows one of the key goals of this work: that simulator models do not necessarily model a realistic hardware implementation. Models developed from simulator research do not map one-to-one to a hardware prototype, even when assuming fairly detailed architectural properties. The last part might be of less interest to those familiar with the intricacies of processors, but highlights the advantages of following the work to the finish line, as such issues are unlikely to be intuitive to computer architecture researchers without a background in hardware design.

Transferring an Abstract Model

The first step of creating this hardware prototype is understanding how to best translate the abstract model into realizable hardware. Speaking generally, this can range from trivial to impossible, depending on how many abstractions are implemented and how easily they translate into actual

hardware. Some concepts that a researcher might erroneously assume to be easily available in hardware, can actually be very difficult or costly to implement. One example of this is checking if an element exists in an array. This check, usually trivial in higher abstraction levels, is very expensive to perform in hardware, as it requires many comparison units, or many cycles to check one element at a time.

Thankfully, the abstract model used as an example here, DoM, mirrors fairly common hardware concepts and can easily be transferred. It even makes overt references to the SQ-Head and the SQ-tail and how these should be updated. It is often wise to start from a simple implementation and move on to more complex implementation, as debugging functional errors in simple systems is considerably easier than in complex systems. Therefore, we perform a scalar OoO implementation first. There is no value in implementing these techniques for in-order or superscalar in-order, as the phenomenon they are attempting to prevent is only possible due to OoO execution. On the other hand, immediately attempting a superscalar OoO implementation is likely to be very difficult.

For the following sections, we will be referring to and explaining the components of Figure 3.1, including how and why we altered certain parts of its design.

Scalar Out-of-Order Implementation

To implement DoM for a scalar OoO processor, we translate the shadow queue and the release queue into cyclic buffers with separate heads and tails tracked. At its most basic, this results in three cyclic buffers: The shadow queue holds entries that indicate whether a given control instruction is active or inactive (True or False, in implementation terms). The shadow tag holds the value at which point tracked loads should be released, while the load queue index buffer holds the reference to index of the load in the load queue. The shadow queue uses the SQ-Head and SQ-Tail to keep track of its current state.

For the release queue to function, we need to release loads when they are no longer under a shadow. In Figure 2.5.5, we explained this as a result of an entry in the shadow queue being freed. However, if we free release queue entries as a result of shadow queue entries being freed, we might need to compare every single release queue entry at once, as they could all share the same shadow tag. Instead of doing this, we implement the release queue similarly to how the shadow queue was implemented: we check whether the current shadow tag at the head of the release queue is active or inactive, and free the load if the tag is inactive. This results in us needing a head and tail for the release queue, which we will respectively refer to as RQ-Head and RQ-Tail. In addition to the aforementioned queues, heads and tails, we also add a shadow queue index to ROB entries, in order to locate which shadow queue entry to mark as inactive after a successful branch resolution. As with the original DoM design, we add a speculative bit to the load queue entries.

Connecting this to a processor is simple: Control instructions entering the ROB cause an entry to be allocated in the shadow queue and SQ-Tail to be incremented by 1. Loads entering the ROB cause an entry to be allocated in the release queue, if the shadow queue is not empty. The speculative bit in load queue entries is set according to whether the shadow queue is empty at their time of entry. When a control instruction is resolved, the branch mask bundles with the mask the ROB index of the instruction. We use this ROB index to retrieve the shadow queue index, and use this to set the entry in the shadow queue as inactive. Every cycle, if the entry at the head of the shadow queue is inactive, the SQ-Head will increment, freeing shadow queue entries. It will increment a number of times equal to the width of the processor, or until it reaches an active entry, or the SQ-Tail. For a scalar OoO processor, the width of the processor is one, and the SQ-Head will therefore only increment by a maximum of 1.

Every cycle, if the shadow tag at the head of the release queue is no longer in use, the RQ Head will increment, releasing entries. As the RQ-Head increments and frees an entry, it sets the speculative bit of the associated load, as given by the load queue index of the entry being freed, to be false. The RQ-Head follows the same increment rules as the SQ-Head.

The load logic also needs to be changed to check whether a load is speculative before allowing it to proceed to the L2 cache. If the load is speculative, it is not allowed to proceed and is instead

delayed, which means the load queue has to retry it at a later point. When a load is no longer speculative, it can access the L2 cache upon being retried. A speculative load that hits in the L1 cache proceeds as normal. In order to achieve this, we transmit the speculative bit together with the load request, and directly alter the L1 cache: When a speculative load misses in the L1 cache, we transmit a nack response back to the LSU. This tells the LSU to retry the load at a later time.

We will now address the logic of checking shadow tags. Earlier, we discussed freeing release queue entries when the shadow tag of the entry was no longer in use. A shadow tag is no longer in use if it is not between the SQ-Head and the SQ-Tail. However, as we are using cyclic buffers for the shadow queue, checking this is not entirely trivial. When the SQ-Tail wraps, it will go back to a value of 0, meaning that it has a lower numerical value than the SQ-Head, despite being later in the shadow queue. As the numerical comparison for whether a shadow tag is between the SQ-Head and the SQ-Tail varies based on whether SQ-Head < SQ-Tail, we have to implement both of the checks, and choose which comparison to use based on a multiplexer of SQ-Head > SQ-Tail. This is not a complicated problem, and is a familiar solution to many who have worked with hardware before, but it adequately highlights unexpected difficulties when working within hardware.

We address here a general design strategy that will be recurring throughout this work. When designing the shadow queue and the release queue, we are constraining them to release no more entries than the width of the core, even if there are more entries that could be released. The reason for this is that parallelizing hardware components has a cost in terms of both design complexity and size. However, it is necessary to release enough entries to not bottleneck the processor. Due to the way the BOOM is designed, this is considered to be the core width for most components. As such, for this section we are only releasing one entry in a cycle, while in later sections, when the core width is two or three, we release two or three entries a cycle, respectively.

Supporting Branch Misprediction

The previous section describes a full implementation of the abstract model as presented by Sakalis et al. [1], but this model does not meet all the necessary requirements to be functional on the BOOM. Although this model does block and release loads, according to control shadow tracking, it fails to meet a guarantee the processor requires: the ability to restore state and achieve eventual correctness. This is because it does not explicitly support misprediction handling. This was not addressed in the original paper, but may have notable implications for the size and efficiency of suggested policies. DoM is a relatively minor example of this and requires only a small set of modifications due to its relatively simple state, but more complex systems might experience more dramatic changes. Further research is required to investigate this phenomenon.

In this work, supporting misprediction means rolling back the release queue and the shadow queue to their states at the time the control instruction that mispredicted was entered into the shadow queue. This means that all entries after the resolved entry in the shadow queue need to be squashed. The SQ-Tail must be set to be the index of the resolved entry + 1, as this would be where the next valid entry after rollback should be inserted.

This does however get more complicated when looking at the release queue. The release queue is not directly linked to the resolved branch, and there is no inherent logic behind how many loads one shadow tag can have. It might have zero, one, or many loads associated with it. Similarly, searching for the preceding load from a branch is also difficult: As branches could be located anywhere in the ROB, and would require hardware to check every entry. There might be many instructions to the nearest load. This load might not be speculative either, and as such give no information about the state the release queue should be in.

It is possible to use the updated tail from the shadow queue to invalidate entries in the release queue, but this comparison has notable limitations. An entry would be invalidated if it no longer resided between the SQ-Head and SQ-Tail. However, as the release queue is designed to only free a number of entries equal to the width of the processor each cycle, it might have entries that would be freed within the next few cycles. This occurs regularly when there are many release queue entries dependent on a single shadow tag. These release queue entries would now be squashed instead,

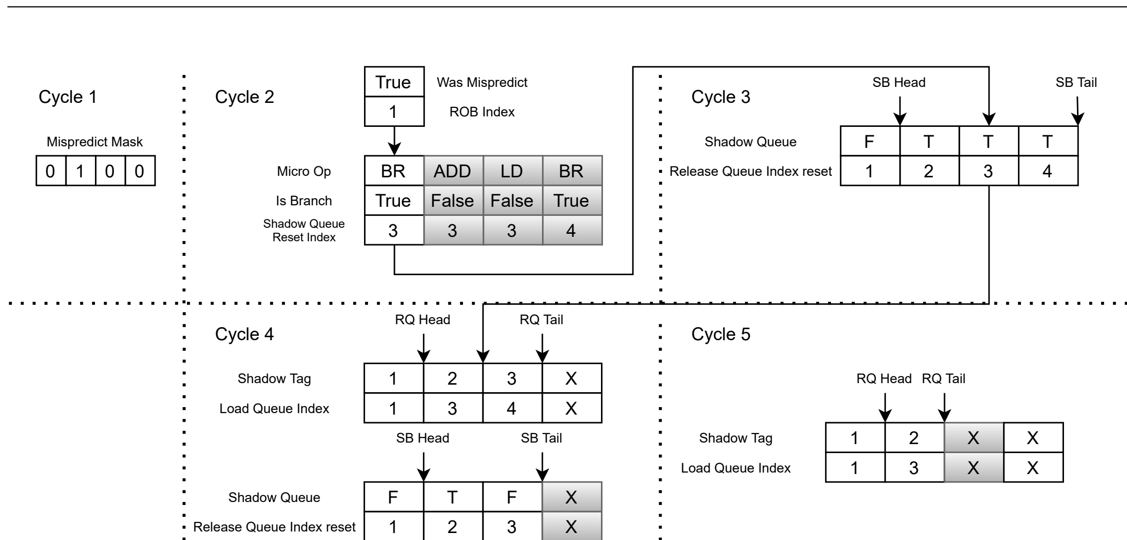

Figure 3.2: The cascading logic when a mispredict occurs. They gray entries are the ones squashed as an effect of the mispredict. Note that it takes a cycle for pointers to be updated (this is not shown for ROB, as we have not changed it). X indicates that we do not care about the value, as it is not used for this.

and leave speculative loads in the load queue that would never be declared non-speculative.

The previously discussed systems are viable, but they are not efficient. They add extra complexity that makes the system less intuitive, and as they require comparisons for all entries in the release queue, they have a large size overhead and scale poorly. Instead, it is beneficial to focus on the reliable indicator of the rollback state: the shadow queue entry. By extending the shadow queue to include an index into the release queue, there will be a reliable and simple mechanism to get the appropriate rollback state. We add this element as the release queue reset index. The release queue reset index is set to the RQ-Tail at the time of entry for the shadow queue entry.

When a misprediction happens, we therefore spend the first cycle to reset the SQ-Tail. In the next cycle, we emit the release queue reset index, and use its value to set the RQ-Tail. A visualization of the five cycles it takes to fully reset the shadow queue and release queue is shown in Figure 3.2, with the gray entries being the squashed entries due to the mispredict. Note that instructions are killed off in cycle 1 of the branch mispredict handling using the mispredict mask, while the cascading mispredict handling happens in cycle 2, which is when the ROB index of the mispredicted control instruction becomes available.

Supporting Rollback

In addition to the need for supporting mispredictions, the DoM implantation also has to be able to support rollbacks. For this work, we define rollback as all state in which the ROB squashes or rollbacks entries, that are not caused by mispredicted control instructions. Rollbacks occur for many reasons, including exception handling, and similar to mispredictions, they require the shadow queue and release queue to be restored to the state they were in at the time the rollback causing instruction was entered into the ROB.

The notable difference between mispredictions and rollback is that rollbacks occur from non-control instructions. Under misprediction, we knew the exact entry in the shadow queue we had to restore the SQ-Tail to, and we also had to mark that entry as inactive. However, for rollback, there is not an associated shadow queue entry. As such, we change the design of DoM in the following way: All instructions now have a shadow queue index that indicates their associated entry for control instructions, and their associated rollback entry for non-control instructions. In addition, all ROB entries now have a bit indicating whether they are a control instruction, named in Figure 3.1 as ”Is Control Instruction”.

With these two changes, we can now handle rollbacks in the same manner as by which we handled mispredicts: We set the SQ-Tail to the value given by the shadow queue index in the first cycle, and in the second cycle we set the RQ-tail to the value given by the entry immediately preceding the SQ-Tail. All entries following the updated SQ-Tail and RQ-Tail are squashed. Notably, we do not set the youngest remaining entry in the shadow queue to be inactive, as a rollback is not triggered by the control instruction being resolved.

Other Details

This is most of the work necessary to make a functioning implementation of DoM on the BOOM, when the core width is one. Although the work is mostly straightforward, some complications are emerging: There is a noticeable increase in the size overhead due to the need to support a release queue reset index upon rollback. In addition, the original work does not cover how ROB entries are linked to their entries in the shadow queue for rollback, and this also increases the size overhead somewhat.

Another noticeable aspect here is the size of the various buffers and indexes. Logically, there is never any need for an index to have a larger value than the amount of possible entries, so these are allocated bit-efficiently according to log2 ceiling, which is the smallest amount of discrete bits necessary to cover all possible values when encoded in binary. However, a better question is on the sizes of the various buffers themselves. For the release queue it is most logical to use the same amount of entries as the LDQ as the basis for the release queue as there cannot be more speculative loads than there are loads themselves. For the shadow queue, there is a maximum amount of branch depth that the BOOM uses, usually limited by the number of backups required for rolling back register renaming. For simplicity's sake, we set the shadow queue to match this maximum branch depth. In the following sections, the problematic aspects of this will be explored.

3.1.3 Implementing Delay-on-Miss on Superscalar BOOM

Implementing a functioning prototype of DoM on scalar BOOM is a good starting point, but without superscalar support, OoO systems have a much lower performance gain than when there is a higher throughput in the system. Therefore, it is critical to move over to fully supporting a superscalar system, including the ability to handle multiple instructions going through the DoM in a single cycle.

In line with the reasoning for performing a simple implementation before moving to a more complex one, we implement support for a 2-wide core before we migrate to greater widths. This is a way to introduce and fix many of the problems that superscalar design introduces, without having to make all the design changes necessary for a generically wide design.

Superscalar Support

One of the critical considerations when designing superscalar processors are the bottlenecks. If every instruction has to pass through a part of the processor, and that processor can only support a certain number of instructions per cycle, then that becomes the limitation of the processor, no matter how wide the rest of the processor might be. DoM does not interact with every instruction, it only registers branches for shadow tracking and loads to hold in the release queue. However, it can still be a bottleneck if there is a high prevalence of these instructions in the executed program.

If loads are not released quickly enough to continually feed the LSU, certain memory operation heavy workloads might face performance degradation. However, if the release queue is capable of releasing more loads than the LSU can issue to cache, the design will be more complex and need more logical components than necessary. Ideally, the release queue can free as many eligible loads in one cycle as the LSU can dispatch in one cycle. This prevents the release queue from being a bottleneck and also guarantees that it can't release more loads than the LSU can dispatch. Of

course, the LSU might not handle the loads immediately, such as in cases where it has higher priority incoming loads, or not all the operands are ready for the load request yet.

Another challenge of superscalar is that all supporting structures need to be able to handle the worst case scenario for incoming instructions. As the pipeline now supports up to two instructions within one cycle in all in-order parts of the processor, the release queue and the shadow queue need to be able to potentially handle two loads or two branches, respectively. Even more complex is the ability to handle one load and one branch, in either order, and the challenges this creates will be a key aspect of the following subsections.

Before implementing full support for superscalar, let us first consider the potential alternatives to supporting multiple instructions of either load or branch in a cycle. It is fairly trivial to limit the dispatch stage to only dispatching one instruction within a cycle, if both instructions are either loads or branches, or a combination of these. This would also massively simplify the logic for adapting to the larger pipeline width, as very few modifications would be necessary. However, this results in stalling the rest of the frontend whenever there are two branches or loads in the issue unit. This creates notable limitations, as both branches and loads are estimated to be around 25% and 10% of the total amount of instructions in a typical program, respectively [5]. The chances of there being a load or branch in the first instruction is 35% and similar for the second one. Assuming these are independent, that gives a roughly one in nine chance of having to only issue one instruction instead of two, or effectively a 5.5% slowdown. This might seem acceptable for the current width, but the issue gets worse the wider the pipeline is. For a 3-wide pipeline, the slowdown becomes 28%. This is the reason why it is necessary to adopt other methods.

ROB and Branch handling

The first part of extending the previous work is understanding how the fundamental assumptions about behavior change when the pipeline width grows. The BOOM uses a banked ROB to support larger pipeline widths. The increased pipeline width complicates one part of allocating entries in the ROB. When two control instructions enter the ROB in the same cycle, if the SQ-Tail is used to set the shadow queue index, they would both have the same value, despite one following one more control instruction than the other. They should be pointing to sequential entries instead. Therefore, the ROB has to keep a running tally of previously allocated control instructions within one cycle and offset the shadow queue index based on this tally. This requires adding together the number of preceding control instructions for each incoming instructions, but this requires only minor overhead.

Control instruction resolutions are also handled differently when moving to a wider pipeline. Branches are still committed atomically, but their branch resolutions are now bundled together, as the BOOM uses a branch mask to cover all oversights from the branches. As discussed in Figure 2.2.3, when only one branch was cleared in a cycle, the additional information bundled with the branch mask could be used to determine the ROB index of this control instruction. However, for larger pipeline widths, this bundled information only indicates the ROB index of the youngest resolved control instruction. As such, it cannot be used to find the ROB indexes of all control instructions if there are more than one control instruction resolved in one cycle. Therefore, we have to manually add a bundle of signals holding the ROB indexes of *all* branches that were resolved the previous cycle.

Even though two control instruction can be resolved as safe within one cycle, multiple control instructions cannot be resolved as mispredicted in one cycle. This is because control instructions are dependent on each other, one mispredicted control instruction would have to happen before the other, and therefore invalidate the younger instruction. This theoretically simplifies misprediction handling, but a mispredicted control instruction can be bundled with a safe control instruction, creating complications. The safe control instruction has to be older than the mispredicted control instruction. The ability to handle clearing control instructions needs to be separate from handling mispredicts. As the SQ-Head only increments one cycle after an entry has been resolved as safe, at the earliest, control instructions being resolved as safe and misprediction handling can be detached from each other without issue.

Thankfully, the BOOM stalls the cycle it handles a mispredict. This means that it is not possible for new entries to be added in the same cycle as mispredicts are handled, and therefore the mutually exclusive logic of entry allocation and tail resets due to mispredicts never occur at the same time. Simply, the mispredict handling trumps the entry allocation logic, but even if it did not, there would never be any valid entries in the cycle in which the mispredict is handled.

Switching States

We now come to a complex aspect of extending pipelines for greater widths, the challenges of state switching during a single cycle. When dealing with a scalar processor, there could only be one type of relevant instruction handled in a cycle, either a control instruction or a load. In addition, based on whether there were entries in the shadow queue or not, it was easy to discern whether a shadow was currently being cast and therefore whether a load needed to be tracked or not. However, these simplified scenarios are no longer guaranteed.

When a control instruction and a load are entered in that order into the ROB in a single cycle, the previous implementation would erroneously not track the load if the shadow queue was previously empty. As the control instruction entered the shadow queue, it would correctly track it as part of the shadow tracking, but the state change would only be visible to the release queue in the next cycle. The release queue would therefore be observing the old state of the shadow queue when deciding whether to track the load. Erroneously seeing the shadow queue as being empty, it would not enter the load into the release queue, as there was no observable shadow being cast. This way, the security premise of DoM fails, and certain loads again become observable in side-channels while speculative.

This makes it necessary to add several changes to the DoM implementation in order to correctly and precisely track these state changes. There is no need to consider state changes for when the shadow queue has active entries. In that circumstance, we always add new loads to the release queue, but we still need to consider offsets: If the shadow queue has empty entries, but there is an incoming control instruction in slot 0, then the load in slot 1 should have a shadow tag that is equal to the SQ-Tail plus one, as that will be what the shadow tag would be if these instructions were handled atomically. We discuss this at the end of this section, before we discuss how to support greater widths in subsection 3.1.4.

The following details are relevant only when the shadow queue seems to be empty. This state only occurs when the shadow queue actually is empty, but detecting whether a structure is full or empty is not always simple. For the sake of simplicity, we therefore only examine the state changes when the shadow queue actually is empty here, and examine the other related challenges in section 3.1.5.

To handle release queue logic when the shadow queue is empty, it is necessary to implement a multiplexer to select how to handle incoming loads and control instructions. If the shadow queue is empty, entries should only be added to the release queue if there is a control instruction in slot 0. However, this information is by default not available to the release queue, which only has information about incoming loads. Therefore, the signal for incoming control instructions has to be extended to also go to the release queue. This information is transmitted with a signal that is as wide as the core, with each bit indicating whether a control instruction was entered into that slot.

Since the width is currently two, technically it is only necessary to check if there is a control instruction in slot 0 (i.e., checking older instructions), but as we will want to support larger core widths, we implement it as a generically wide signal. If there is a control instruction in slot 0, we add any succeeding loads to the release queue. This then also needs to be added with an offset of 1, as the SQ-Tail will increment by 1 in the same cycle. Fortunately, we don't have to implement an equivalent for the shadow queue, as it will simply always add all incoming control instructions.

On the other hand, if the shadow queue is not empty, it is necessary for the shadow queue to track incoming loads, as these contribute to changing the release queue reset index. If there is an incoming load in slot 0, and an incoming control instruction in slot 1, the shadow queue corresponding to the control instruction should have a release queue reset index equal to the RQ-Tail plus one, as

there would have been one unobserved release queue entry that is older. Therefore, the shadow queue will have to also observe incoming loads.

3.1.4 Supporting a 3-wide Processor

The LargeBoom config employed by BOOM has a width of three and is also the largest functioning BOOM configuration for our experimental setup. It is advantageous to have an implementation as representative as possible when dealing with research, and as modern CPUs support widths of six or more, a width of two would be considerably unrepresentative. Therefore, we present the work of migrating the DoM changes to function on a larger core. The key challenges to this involve multiple state changes and calculating offsets.

Complicated State Changes

When dealing with a 2-wide processor, one of the biggest challenges is that the state can change within one cycle, meaning that a control instruction causing a shadow can precede a load within the same cycle. This logic gets more complicated when dealing with a 3-wide core, as the number of configurations of control instructions and load instructions becomes more convoluted.

As an example, when dealing with a 2-wide core, if the shadow queue was empty, and there is one control instruction incoming and one load incoming, it is only necessary to check that the control instruction is in slot 0. If so, the load should be added to the release queue with an offset of 1 for its shadow tag.

However, when dealing with a 3-wide core, this becomes more complicated. The control instruction can now occupy either slot 0 or slot 1, and there can be one or two control instructions preceding a load that should be tracked, instead of always one. In addition, there could now be loads preceding and succeeding the one control instruction, in which one of them should be tracked while the other should not. Finally, there could be two control instructions preceding a load instruction or two load instructions succeeding a control instruction. The first case would require an offset of two for the load instruction instead of one, while the second would require filling in two entries in the release queue at the same time, both with the same shadow tag offset.

Ultimately, these state changes are a large part of what makes ensuring correctness in processors difficult, as we will explore further in subsection 3.1.5. For now, we will build on the previous work we did while migrating to a core width of three.

Calculating Offsets

In order to calculate how to consistently and correctly insert incoming control and load instructions into their queues, it becomes necessary to extend the input going into these structures. We will deal with the simplest case first, in which there are currently entries in the shadow queue, and there can be any number of branches and loads being inserted. Dealing with insertions in this state is different for the shadow queue and the release queue, but both require considerably more wiring than for smaller core widths.

The shadow queue has to, for each input, calculate how many control instructions that precede the current input. This will be used to insert the new entries into the shadow queue with the correct offsets. As entries should be inserted sequentially, without overwriting each other, each entry after the first needs to be entered into the queue with an index calculated from the SQ-Tail and the number of preceding entries in earlier slots. This offset calculation and addition has to consider wrapping, so that the shadow queue does not attempt to insert an entry outside the structure.

However, the shadow queue also has to keep track of the release queue reset index, in order to resolve mispredicts correctly. This means that for every input, it also becomes necessary to calculate the amount of preceding loads for any given entry. Again, we are only considering the situation

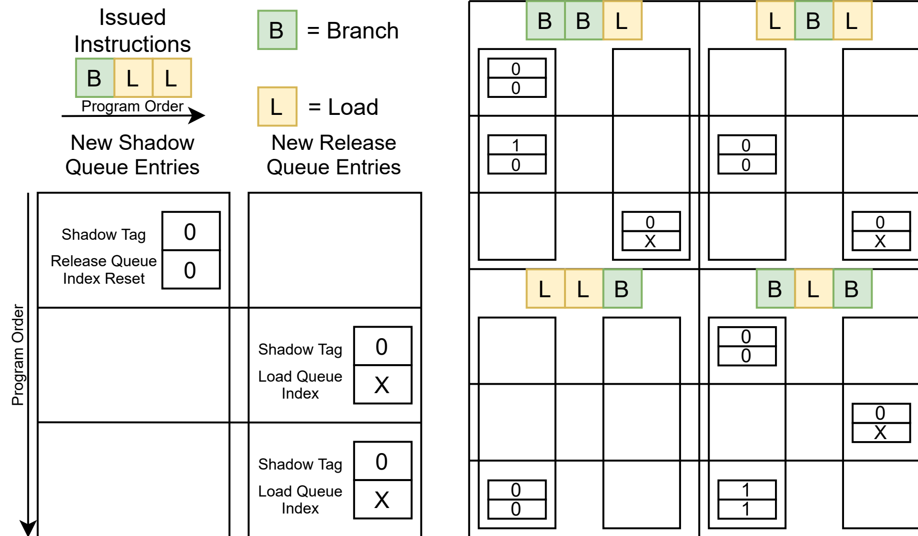

Figure 3.3: An example of created shadow queue entries and release queue entries when shadow tracking was not active at the start of the cycle. Note that ordering is only maintained explicitly within the shadow queue and the release queue. Order between queues is maintained through release queue index reset and shadow tag. Assume SQ-Head, SQ-Tail, RQ-Head, and RQ-Tail are all zero when the instructions were issued.

in which there are already entries in the shadow queue here, so loads preceding the activation of shadow tracking will be considered later. For every input channel, it is checked whether the channel is valid, and if so the entry is inserted with an offset equal to the amount of preceding control instructions and with a release queue index reset equal to the current RQ-Tail added with the number of preceding loads.

For the release queue, the situation is in many ways mirrored compared to the shadow queue. Assuming there are entries in the shadow queue, for each input slot, the release queue needs to check whether there is a load present and calculate the offset for the resulting entry by tallying the number of preceding loads for that slot. This will be used together with the RQ-Tail to calculate which index to create the entry in for the incoming load. The shadow tag will then have to be calculated by adding together the current SQ-Tail and the amount of preceding control instructions. Here too, the shadow tag has to be wrapped according to the size of the shadow queue.

As we see, these two approaches are mirrored, in that the shadow queue needs to calculate the offset by which release queue entries will be inserted, and the release queue needs to calculate the offset by which shadow queue entries will be inserted. Naturally, this means that designing the shadow queue and the release queue as separate structures has created a lot of extra wiring, as they are both dependent on complete information about the other. However, it would be too complicated to change the design this late in the process. We discuss an alternative solution later, in subsection 5.2.4.

We will now examine how these implementations get complicated by the state switching discussed earlier. Previously, we only looked at when there were active entries in the shadow queue. However, it is also possible that there are no entries in the shadow queue and then a control instruction and a load instruction enter. This requires different handling by the release queue, but also minor adjustments to the shadow queue.

We look once more on the shadow queue and see what extra changes have to be made. The basic challenge here is that the release queue index reset is calculated based on all preceding loads. However, if shadow tracking was not currently active, the release queue index reset should only be offset by incoming loads that are following at least one control instruction. If all loads are counted, the offset would be incorrect, as loads that precede the control instruction that activates shadow tracking should not be counted. This is because these will not be added to the release queue, as

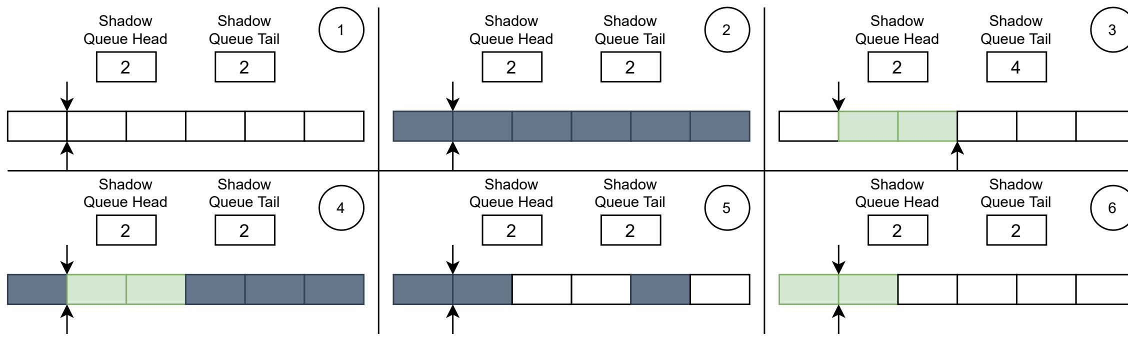

Figure 3.4: An example showing the challenges of calculating whether a queue is empty or full (or neither). White entries are unusued, gray entries are used, and green entries were freed last cycle. 1 shows an empty queue, with head and tail equal. 2 shows a full queue, with head and tail equal. 3 shows an empty queue, but the head has not yet incremented, making the head and tail different. 4 shows a not-full queue, but the head has not yet incremented, so the head and tail are equal. 5 shows a not-full queue, but the previously freed entries are not at head, so head and tail are the same. 6 shows an empty buffer, but head is technically a full queue rotation behind tail. This raises an additional problem of what the head should do in such a case.

they are not speculative, as indicated by them having no shadow tracking them.

Therefore, we now see that it is necessary to employ two different counting measures. In Figure 3.3, we see several examples of what inputs we can get when operating with a width of three, and the entries created as a result. We implement changes so that for every input channel, both the number of preceding loads and the number of preceding loads that follow at least one control instruction are counted. Then we select which to use based on whether the shadow queue is empty at the start of the cycle.

The check for loads following at least one control instruction is somewhat more complicated, but should still not result in large hardware components. It is important to avoid relying on designs that result in cascading chains of dependencies, as they are unlikely to meet timing requirements. Even though this wiring is more complex, as it is not handling many input slots, its design should still be manageable for a system of this width. However, the complexity of this is highlighted, as it is unlikely to scale well to larger systems that employ processor widths of six and upwards.

The check for release queue entries is now very similar to those for the release queue reset index, as they are both trying to predict where new entries will be issued. It becomes necessary to check whether shadow tracking is already active by receiving a signal from the shadow queue indicating if it is empty or not. We then either select the amount of preceding loads as an offset, or the amount of preceding loads that follow at least one control instruction. We see again that the logic structures for input are mirrored between the shadow queue and the release queue.

3.1.5 End Result and Challenges

Many varied and difficult fringe cases occurs when working within a complicated system such as the BOOM. This section aims to explore some of these challenges and see what they tell us about both the BOOM and the precision of it as a research tool.

Empty or Full

One challenge that may seem trivial, but requires some careful design changes, is detecting whether a queue is full or empty. At a software level, this is typically a trivial problem, but when working within hardware, it becomes considerably harder for the following reasons: Firstly, the main source of information about a queue are the head and tail of that queue. When a queue is empty, head

and tail will both be pointing to the same entry, as any incoming entry should be entered at the head. However, in cyclic buffers, as the queue is in operation, the head and tail will increment and eventually wrap around. When the queue is full, the tail will catch up to the head, and they will both be pointing to the same location, as seen in Figure 3.4. Secondly, these queues tend to have limited guarantees on which slots are filled. In a queue, several slots might have been marked as inactive, but because the entry at the head is active, the head has not changed. It is therefore not possible to get accurate information about the number of active entries from just the head and tail.

A logical approach would be to check whether the head and tail are equal and then check whether the entry at the head is active, as this would indicate that the queue is not empty. However, if both the head and tail are currently incrementing, as the head is clearing entries and the tail is adding entries, then a simple approach might give an incorrect answer, and adding in information about current changes to the queue would make the check complex. Similarly, depending on whether a queue should be defined as full when it has active entries and is unable to take more entries, or if it should be defined as full when all its entries are active entries, it is difficult to extract this information elegantly.

We decide to tally up the amount of active entries in the entire queue, instead of attempting a more elegant solution. This gives us accurate information about both the fullness and emptiness of the queue, and cannot give false positives. As such, our edge case handling is reliable. Alternatively, and for future work, a counter could be used to keep track of the amount of live entries.

Stalling Frontend Dispatch

The DoM method does not provide detailed implementation details, as it is system ambiguous and some modifications will be necessary for any system it is to be implemented on. However, there are some limitations of the DoM strategy that are likely to be universal and though workarounds exist, they tend to degrade size efficiency or performance somewhat.

Consider the differences in how control instructions are cleared from the shadow queue compared to the BOOM branch mask. Within the BOOM, there can at any point be a maximum amount of unresolved control instructions equivalent to the amount of slots in the branch mask. However, the branch mask resolves these control instructions and remove them as they are resolved, regardless of their age. In addition, it can fill in control instructions in any holes in the branch mask, not just sequentially as in a chronological, cyclic buffer. Therefore, control instructions can be cleared from the branch mask before they are cleared from the shadow queue. This creates a notable problem, as if a control instruction at the head of the shadow queue has a considerable latency to resolve, it could stall the SQ-Head for a long time. Eventually, as new control instructions are issued and entered into the shadow queue, the entry at the head of the shadow queue would be overwritten as the SQ-Tail wraps.

Fundamentally, this problem cannot be resolved entirely by reasonable size increases. There could naturally never be more unresolved control instructions issued than there are slots in the ROB, but increasing the size of the shadow queue to be equivalent to that of the ROB would make for large amounts of wasted space in the vast majority of execution contexts. However, if we allow the ROB to be larger than the amount of shadow queue entries, there is no guarantee that the previous issue will not occur, as a malicious entity could craft a program that would perpetually feed control instructions while under a long-latency control instruction resolution. This would be a functional error and would not be acceptable for the BOOM core.

Therefore, in order to properly avoid this issue, some design changes have to be implemented: It has to be possible to stall control instructions if issuing more control instructions would overwrite a valid shadow queue entry. In addition, this should happen as infrequently as possible. Although it is, as mentioned, impossible to fully prevent a potential stall from occurring, it is highly important to have the stalling occur as little as possible, as front-end stalls are detrimental to performance.

In order to prevent this issue from occurring, we therefore extend dispatch with a few new features. We add two registers that monitor how many control instructions were dispatched one cycle and

two cycles ago, respectively. This is necessary as there is a 2-cycle latency until when the SQ-Head and the SQ-Tail are updated. In order to keep an accurate tally of what state the shadow queue will be in after these have entered the ROB and shadow queue, we have to store this additional information. Then, we use the SQ-Head and the SQ-Tail and count incoming as well as previously dispatched control instructions, to check if we would overwrite the SQ-Head with the SQ-Tail. If so, we stall the frontend until this is no longer the case.

Interestingly, this work also uncovered the same issue causing challenges with entries in the release queue. As the head of the release queue also possess a shadow tag that can possibly be overwritten, we need to take the same precautions. Consider a situation in which the shadow queue is nearly full and there are several entries in the release queue, all with a shadow tag that would be inactive if the SQ-Head incremented once. If the current entry at the shadow head gets declared inactive, the SQ-Head will increment in the next cycle. This will cause the dispatch to no longer stall any incoming control instructions. With a 3-cycle delay, new control instructions will be entered into the shadow queue. The release queue will also release loads that were marked with the shadow tag correlating to the SQ-head, but only at a rate equivalent to the width of the core. Therefore, if there are more than three times the core's width entries in the release queue, all tagged with the previous SQ-Head value, there will still be release queue entries tagged with this when new entries get allocated to the shadow queue. If the SQ-Tail then increments past where the old SQ-Head was, these entries in the release queue would then observe that their tags are once again active and will stop being released. This creates orphaned instructions that will never clear, stalling the system.

Although the explanation of this phenomenon is complicated, the actual situation will occur with a relatively high rate. Typically, something as simple as leaving a loop and loading up a large amount of new values to perform new work would create this exact scenario. If the leave condition for the loop is dependent on the value of an operation in the loop, one could also expect there to be a considerable latency and for this control instruction to be blocking the shadow queue for quite a while. Even without the likelihood of this happening, simply the fact that it is possible for it to happen is a hardware design fault, and one that can cause a processor to stall indefinitely. Therefore, we need a way to prevent this from happening, while also setting our configurations so that stalling the frontend happens at a reasonably low rate.

Our approach is very similar to the previous work in preventing the shadow queue entries themselves from being overwritten, and we reuse many of the previous structures from that. We change it so that the release queue now emits what the shadow tag of the entry at RQ-Head is. Then we can check by adding all control instructions dispatched in the last two cycles and in the coming cycle to see if their dispatch would overwrite this valid tag. If it does, we stall the frontend until such a time as this is no longer the case. As the branch stall dispatch observes the SQ-Head and SQ-Tail directly, it will know with a one cycle delay when it can dispatch control instructions once more.

Final State of Implementation

As discussed earlier, it is very difficult to develop and verify hardware implementations, even on FPGAs. There are many issues that can occur, and some of them will be incredibly difficult to debug without proper tooling. Unfortunately for this work, the FPGA cards available to run experiments did not possess DMA and as such many of the most popular open-source tools were not available for debugging. By this state of the implementation, the vast majority of bugs had been addressed (as detailed), but some fringe cases that occur some billion instructions into a few of the benchmarks remain. These are very complicated to debug, as debugging approaches such as Verilator would take weeks to run that many instructions, and the complex internal state of the processor is not available when running the system on an FPGA.

As such, at this point in development, we have to finalize the current implementation in order to collect data and perform results analysis. The current implementation of DoM therefore has some bugs that causes the system to stall and stop under certain circumstances, but they are exceedingly rare. As documented under the methodology in subsection 4.1.3, we are able to run several benchmarks to completion on the FPGAs. This indicates that the error is very fringe and

Figure 3.5: Load requests after implementing DoM. The new steps are colored. A shows the speculative check. Non-speculative loads are issued straight to cache, as before. Speculative loads check for an L1 cache hit, as shown by B, and receive an immediate response, as shown by C. If the load hits, it is issued to cache, if it misses, it is sent to the delayed loads queue as shown by D. Once the load is no longer speculative, it is available for execution through the instruction queue once more, as shown by E.

should hopefully not have a large performance impact on the successful benchmarks. However, as we discuss under future work in subsection 5.2.1, future research based on this work would be prudent in fixing these bugs to improve the accuracy of their results.

3.2 Implementing Delay-on-Miss on gem5

This section details all the work necessary to replicate the implementation of the Delay-on-Miss (DoM) technique presented by Christos Sakalis et al. [1] on gem5. This research was originally performed using gem5, and has since been reproduced by other research within the field [35]. These works have however reported notably different results than the original work. This work therefore primarily aims to reproduce the original functionality with a focus on hardware realism, but also to investigate where these differences in performance can arise from.

These sections first describe the initial implementation of support structures, followed by the management of shadow tracking and finally the functional delaying of speculative loads. These sections are necessarily quite detailed, and some rudimentary comparisons are made to the BOOM work, to make the connections between the two systems more visible, in order to aid the later discussions in section 5.1.

We highlight that this design is based on the BOOM implementation, as described in section 3.1. This results in the following work making use of software shortcuts that we know mirror the behavior of the BOOM implementation, in an attempt to ensure accurate performance metrics. We discuss the advantages of this form of hardware familiarity further in section 5.1.

Figure 3.5 shows the complete set of interactions that DoM add into the load request flow used by the O3 CPU. The DoM implementation only affects the flow close to the final step, as the other steps do not affect cache state and as such, do not leak information for our purposes. The following sections describe the implementation details, how mispredicts are handled, and other design considerations.

3.2.1 Creating Tracking Structures

The first question when implementing this work is how to translate the original abstract implementation into a suitable model. With gem5, there is a lot of flexibility in how this can be done, as unlike BOOM, there are software based methods and shortcuts that may be appropriate. With these opportunities, we should still be selective about what approach we choose and focus our design so that it is both approximate of the hardware and easily manageable within gem5.

For example, implementing a cyclic buffer would lead to many of the same problems we experienced when working with the BOOM: There would be issues relating to wrapping and ambiguity over fullness. Neither of these properties are particularly valuable in terms of research, yet going in the opposite direction might lead to overly optimistic designs. For example, utilizing hashing functions would be quite difficult to develop in actual hardware, and research utilizing such a feature would ideally need to explain how its hardware equivalent would function.

For our design purposes, we want to emulate the realism of our hardware prototype, while still taking advantage of the flexibility that software offers. As we do not need to go through every complicating detail of a hardware implementation in order to have a realistic design, we use software shortcuts where appropriate and describe how the implementation equivalent would function in hardware. For example, there is no benefit to implementing a cyclic buffer, but we still wish to imitate this behavior and therefore need to store data in an equivalent manner. We therefore implement both the shadow queue and the release queue as a vector of tuples, and describe their properties in the following sections.

For the shadow queue, we ideally want to keep track of the relevant instruction as well as the rollback position of the release queue in case one of the branches mispredict. The release queue reset index is a carryover from the hardware implementation, that revealed that it would otherwise not be possible to roll back to the correct state without complex hardware or many cycles. This limitation would not be obvious if the work had been done in the opposite order. When in a software context, it would be quite easy to locate the right instruction through a search using the instruction sequence number. The same process would take many cycles to perform on hardware. Therefore, we deem it necessary to keep track of both the relevant instruction and the release queue reset index. Here we can also use the global pointer reference to the instruction we wish to roll back to instead of an index reference into the ROB, as the simulator design gives us more freedom.

In addition to keeping track of the release queue reset index and the control instruction, we also need to keep track of whether an instruction is active or not. As gem5 instructions do not have a speculative bit by default, we can choose to either extend the instructions with this tag or we can manually keep track of this by handling control instruction resolutions within the CPU. The latter approach is considerably closer to how this would work on actual hardware, and also closer to the approach taken with the BOOM. However, while the BOOM handles this through monitoring resolved control instructions as they enter the ROB and clear entries based on this, the gem5 implementation directly tracks the cleared control instructions from the CPU as they are issued to commit. This approach can also be taken with the BOOM, and is more likely to be reflective of how an eventual implementation would act like.

To keep track of whether an instruction is active or not, we add a third element to the tuple, namely a boolean, to indicate whether the instruction is still active. Whenever a control instruction is resolved and marked as safe, and is set to be ready to commit, we also update the corresponding entry in the shadow queue. If the instruction is in the shadow queue, we declare it to no longer be active.

For the release queue, we need a tuple with two entries. This is to keep track of what the shadow tag is for the entry in the first slot, and the second slot stores a reference to the instruction that the entry is tracking. This reference to the instruction is necessary both to later update it when it is no longer under any shadows and can be declared non-speculative, but also to handle any squashes that might occur. The O3 CPU has quite a lot of checks that might result in a load being squashed for any number of reasons, such as load-store ordering violations, and it is necessary that it is possible to remove entries from the release queue. The easiest way to do this is to keep the

global reference and check it with every squash that occurs elsewhere.

Now we look at the actual methods to implement this in gem5. As previously mentioned, gem5 uses a specific system of interchangeable components, and even when they are not interchangeable, they tend to be mostly isolated, with interfaces between them. This is particularly true for the core parts of the O3 CPU, as most of the internal communication occurs through ports that mirror how wiring and bundling would function in hardware. Some interactions in the processor, specifically those attempting to access a smaller component that is a part of a larger component, are often invoked directly with functions instead of communicating through ports.

For this work, a hardware identical implementation would utilize a port-like system, with connections to the relevant parts, but that would be overly constrained for our goals. The most important thing is that the abstract implementation does not exploit any properties that do not have a hardware equivalent. Therefore, we have the ability to set up the component in the most flexible way possible, at least for early iterations, and then either verify that our assumptions are valid or later migrate it to a more realistic design.

We develop the DoM component as a standalone component, meaning it is connected directly to the core of the processor, and not a subcomponent of another component, such as the ROB is to the commit stage. In addition, we interface with the component through function calls instead of ports, as this is considerably easier, and should have no effects on timing, which is only reliant on when the entries are freed. We start by implementing the shadow queue and the release queue, and we focus on the first three core aspects of DoM. Firstly, it has to be able to add incoming control instructions into the shadow queue. Secondly, it has to be able to add new loads to the release queue whenever there is at least one active entry in the shadow queue. Thirdly, it has to be able to declare shadow queue entries inactive when the originating control instruction has been resolved. We now implement these features stepwise within the O3 CPU.

For the first point, we have to design it so that we extract all control instructions and determine whether that control instruction is speculative. Thankfully, within the O3 CPU, all control instruction are considered to be speculative, and will eventually have a corresponding speculation resolution. Locating all control instructions becomes a matter of extending the commit handling. Within the O3 CPU, all instructions are sent to commit to be part of the ROB, so we check to see if the instructions are marked with the "isControl" property and then issue these to the DoM component. This ensures that all control instructions are entered into it and in program order.

For the second point, we split the capability of adding release queue entries into two separate issues: Sending all load instructions to the DoM component, and checking whether there is an active entry in the shadow queue. The first half of this is done similarly to how we handled control instructions. We add a check whenever the O3 CPU send instructions to commit to see if the instruction is a load instruction, issue it to the DoM component if it is. The second half involves checking whether the shadow queue is empty or not. As we have software abstractions, we make a function call to check if the shadow queue is empty and use that to determine whether an issued load should create a release queue entry. Even if the shadow queue only has inactive entries and is currently clearing itself, this would not delay loads, as the release queue entry would be freed quickly.

For the third point, we have to declare loads non-speculative whenever there are no unresolved control instructions older than them, and keep track of which shadow queue entries are active. Based on the previous design decisions with shadow tags, we implement a check that is similar to how we solved this in the BOOM. For every cycle, we check to see if we can free any shadow queue entries. If the leading entry in the shadow queue is inactive, we remove it from the shadow queue and increment the SQ Head. We do this a number of times equal to half the width of the core. After this, we check to see if the leading entry in the release queue has an inactive shadow tag. If so, we free the entry, and declare the load non-speculative. Freeing the entry here involves two steps: the load instruction has to have its speculative bit unset and the entry has to be removed from the release queue. We do this a number of times equal to half the width of the core.

3.2.2 Updating Support Structures

With the previous three points implemented, we have most of DoM functioning, but there are a few details remaining. Adding entries to the queues is an atomic event that happens in response to a new instruction being issued to the ROB, but freeing entries from the queues does not have a corresponding trigger. Whenever a control instruction is resolved, one entry in the shadow queue is marked as inactive. After this marking, it is then possible that the shadow queue possesses several entries that could be freed, and in turn, it is possible the release queue has several entries that could be freed, as shadow tags would be made inactive. Updating the structures could occur immediately after marking the entry in the shadow queue as inactive, but this would not be realistic in hardware. This is because there could be any number of entries freeable, and parallelizing the entire structure to be updated in such a manner would have a large overhead in complexity and size. Instead, we free entries from the queues a number of times each cycle, similar to the BOOM implementation of DoM.

The gem5 simulator progresses the execution of programs through the use of cycles. Every cycle, a signal is issued to the CPU core, indicating that it should proceed one step, meaning it should receive input, process it, and export output. These steps are referred to as a tick. The ticks are used to simulate many of the components in the CPU core that would normally update once every cycle in hardware. These hardware equivalents would be parallelized in order to process all their inputs simultaneously, and export them all within one cycle. However, in software, we are not restricted to handling all events in parallel, and instead handle atomic events, such as a tick being issued. Therefore, for freeing entries from the queues, we use a functionally equivalent solution to the parallel implementation on the BOOM, but this implementation takes advantage of the atomic nature of the simulator to be considerably easier to implement: For each tick, we try to free the entry at the head of each queue several times. Meaning that for each tick issues to the DoM component, we try to free the entry at the head of the shadow queue four times, thereby freeing up to four entries, and then do the same for the release queue. This emulates the level of parallelism that we want the component to have, without having to handle the complicated edge cases that occurs from parallelizing.

For DoM on the O3 CPU, it is reasonable that it should be possible to free at least half as many shadow queue and release queue entries as the width of the processor. This would free enough entries to supply the maximum amount of load requests that the LSU can issue, and would make only the most branch-intensive and load-intensive workloads result in stalling. Although for the BOOM we deemed it necessary to free as many entries as the entire width of the core, due to the considerably larger width of the O3 CPU, it is necessary to free less entries each cycle to supply the LSU. Therefore, we configure the shadow queue, and the release queue, so they check for freeable entries four times every cycle, as this is half of the core width we used and more than the max number of loads the LSU can issue in a cycle.

The order in which entries are freed from the shadow queue and the release queue with respect to each other matters. On a hardware system, the freeing of the shadow queue, and the release queue would be occurring in parallel, as the updated state of either would not be visible to the other until the next cycle. In gem5, if we were to free the entries in the shadow queue first, and then in the release queue afterwards, we would effectively be letting the release queue observe the state of the shadow queue as if it was one cycle further along. This could lead to it freeing entries faster than would be natural on a hardware prototype. There are faster methods that can be employed in hardware to allow for this same-cycle freeing, but noticing this difference in behavior is important. It is not a property that is immediately obvious without intuition for hardware design. As this is only a minor optimization and unlikely to skew performance results, we allow the release queue to free after the shadow queue.

3.2.3 Handling Squashing

For the DoM component to be functioning, there has to be support for handling squashes. The O3 CPU squashes for various reasons, and the system of squashing is not intuitive. The DoM

component does not need to adopt all forms of squashing in a manner similar to the rest of the CPU, but does need to ensure that it maintains a valid state, such that it drops entries that contain squashed instructions. For these reasons, we need to implement support for handling the most common causes of large-scale squashing and for removing squashed entries. This is to ensure that they do not affect the functional correctness of DoM. This is more robust than attempting to track down all sources of squashing within the O3 CPU.

We examine both the execute stage and the commit stage to identify the two largest sources of squashing: squash all and squash from mispredict. Squash all occurs for several reasons, including the need to to access sensitive registers, and to handle fence operations. Squash from mispredict occurs whenever a control instruction is resolved and is revealed to have been mispredicted. When a squash all occurs, all instructions after the current head of ROB need to be squashed. The need to squash all instructions originates from the commit stage. In contrast, when a squash from mispredict occurs, only instructions that are younger than the mispredicted instruction need to be squashed. Branches are resolved in the execute stage, and therefore mispredicts originate from that stage as well.

In order to properly support squashes, we have to understand how they should work within the DoM component. When squashing from a mispredict, we would want to use the shadow queue index to roll back to, and then use the release queue reset index to roll back the release queue to a correct state. When squashing from a specific instruction sequence number, we should instead squash all instructions from both the shadow queue and the release queue that are younger than the given sequence number. As the sequence numbers are chronological, this also ensures that neither the shadow queue nor the release queue would be out of sync: a control instruction cannot be dropped without any dependent load instructions also being dropped, as they are necessarily younger.

To squash from mispredict, we search the shadow queue to find the relevant instruction, and squash all later entries. Note that this is an advantage compared to the BOOM, which required us to store the shadow queue index in the ROB for this specific purpose. The BOOM implementation could be imitated by adding a shadow buffer index property to the dynamic instruction pointer in the O3 CPU. As the end result would be the same, a simple search is sufficient. When all younger instructions have been removed, we then need to update the SQ-Tail, by setting it to the index of the youngest entry + 1. Instead of storing a release queue reset index, we instead squash entries at the tail of the release queue until the entry has an active shadow tag. This gives the same results as the BOOM implementation, but is easier to implement.

We add a function to squash from a sequence number that will handle the squash all function. This reads the latest entry in the shadow queue, and checks if the instruction is younger than the sequence number, dropping the entry if it is. It repeats this process until the shadow queue is empty, or it finds an instruction that is older or as old as the sequence number, i.e., the instruction causing a squash. It does the same for the release queue to ensure that no squashed loads reside in the release queue entries. We note that this is not a feasible implementation for a hardware implementation. Rather, in a hardware context, it would be necessary to store a shadow queue index in the ROB, and then use that to reset the indexes of the shadow queue. As we have shown that this is possible to do in hardware, in a manner of two cycles, which is less than the time for the next instruction to enter the ROB, and as such we deem this shortcut suitable.

Finally, to finish off the implementation of the DoM component, we make a few changes to ensure that the component only carries valid entries. Every cycle, we search both the shadow buffer and the release queue for any entries that have either been committed or have been squashed and remove these from the structures, restoring as we did before. This ensures that any single-instance squashes will not negatively affect the performance statistics of the DoM component. This is a catch-all to ensure the DoM is always correct in unforeseen circumstances, but we also implement a counter to keep track of how often the DoM component actually removes these undetected squashed entries. This counter has been zero for all benchmarks, indicating that there are no forms of squashing we do not adequately handle.

3.2.4 Delaying Loads

With the supporting structures implemented, we look at how to implement the functional change of delaying loads. Loads should only be delayed if they are speculative, and they should only be delayed when they miss in the L1 cache, as this is what creates the observable changes. However, we must also ensure that any observable changes from accessing the L1 cache are prevented. It is important to consider where a load should be blocked, how the L1 cache should be checked if it has the load, and when loads that are delayed should be reissued.

We consider where the check for if a load is speculative, and whether it hits in the L1 cache, should occur. It is tempting to mirror the implementation performed for the BOOM, upon which the speculative information is transmitted alongside the load request and the loads are handled directly through the not acknowledged (nack) signal received from the L1 cache. However, this approach is considerably less feasible than in the BOOM for several reasons.

Firstly, the gem5 system does a lot less of its work in parallel, instead relying on sequential function calls to handle necessary inter-cycle dependencies. As such, checking for load-store forwarding occurs only in the second to final step of the processor side of the memory request. If the check for an L1 cache hit and delaying the load occurred earlier than this, it would be inaccurate to the intended functionality and affect the correctness of the memory system. The immediate next step after checking for load-store forwarding is to build and issue the request, so by injecting the DoM functionality after load-store forwarding, we can prevent unnecessary work while ensuring correctness.

Secondly, the interface between the processor and the L1 cache is a lot more clearly delineated than in the BOOM. While the BOOM has fully integrated the L1 cache into the processor, all interactions with memory in gem5 happen through the port interfaces, regardless of cache level. Therefore, changing this would require a notable amount of rewriting in order to make all the parts of the program that expect requests to be in a certain format.

Thirdly, the requests going to the cache do not get sent back at a predictable interval. Rather, they are responded to using the sender state the requester submits alongside the request. This creates uncertainty about when the request comes back and the relevant request might no longer have a valid reference in the release queue, as the release queue might have been partially or fully squashed and refilled in the elapsed time. This would also complicate the implementation on the processor side.

As such, we take a notable shortcut at this point. Instead of issuing only a single timing request, we split the checking of the L1 cache hit and the actual access into two separate requests. We first issue a functional request, checking whether the requested cache line is available in either the current cache blocks or in the MSHRs. If it is not available in either, we do not propagate the request, but rather send back a nack and delay the load. If the cache line is available in either, we issue a regular timing request, but prevent the access from updating cache state.

This change is not possible in hardware, as it is essentially performing an instantaneous check for whether the cache line is available or not. It is important to be aware of the potential implications this can have, as it reduces cache bandwidth contention somewhat unnaturally. However, we judge this to have a minor impact on performance, as long as the amount of delayed loads remains low relative to the total amount of loads. As we see under subsection 4.2.4, the amount of delayed loads is very low compared to total loads, i.e., speculative loads most often hit in the L1 cache or the MSHRs, in nearly all benchmarks.

To stop information from leaking, we need to prevent the speculative load from altering cache state, e.g., the replacement state. This makes it necessary to alter the requests being issued, as we should only prevent speculative loads from altering cache state. Therefore, our second caveat against wanting to make changes to the request interface has to be rescinded, as it is necessary to update all load requests to now possess this information. We alter the request in such a way to possess a shadowed property which defaults to false and needs to be explicitly set to true to prevent loads from altering cache state. We set this property explicitly after we know that a speculative load will hit in the L1 cache.

3.2.5 Retrying Loads and Statistics

With loads now being delayed when they are speculative and miss in the L1 cache, we need to ensure that they are reissued when they are no longer speculative. Unlike the BOOM, the gem5 employs a load wakeup system, which has better performance than opportunistically retrying loads. To comply with this, we need to ensure that loads which are nacked, when checking for the cache line in the L1 cache, are stored somewhere in the memory system and reissued as soon as they are no longer speculative.

In order to do this, we copy most of the logic by which the O3 CPU handles load requests that fail due to a blocked cache, but repurpose it to work for delayed loads. We extend the instruction queue component with a new queue, named delayed memory instructions, and whenever a delayed load is nacked, we issue it to this queue. Upon entry into the queue, its parameters are reset, but not its translation. This ensures that load-store forwarding is checked once more when it gets reissued, but it does not need to be retranslated.

The delayed memory instruction queue is integrated into the execution stage arbitration, in which the CPU core decides which instructions should be executed in that cycle. The queue will issue all entries that are no longer speculative, thereby sending them to memory again. As these loads are checked in the order they were issued to memory, but only issued again if they are no longer speculative, they are not necessarily issued in program order, but will issue close to program order. A reissued load will not be delayed again, as it is only issued when no longer speculative, and loads cannot later become speculative. It can still be blocked for other reasons related to memory, but these will be handled by the already existing design of the O3 CPU.

With these changes, DoM is now fully implemented and functioning on the O3 CPU on gem5. This enables us to execute programs and monitor performance, but with a much slower real-time performance than on the BOOM. In order to get detailed information from the gem5 implementation, we endeavor to add in statistics for all relevant aspects of the DoM implementation. This includes entries entered into, cleared from, and squashed in the queues, as well as information pertaining to the delayed loads. Since we already have functioning implementations for all of this, we add in appropriate statistics to the gem5 statistics system, and ensure that the appropriate parameter is increased when a relevant action occurs.

Chapter 4

Gathering and Presenting Results

4.1 Methodology

Evaluating a qualitative examination such as this work is difficult as performance results are not able to give us a complete picture of the research value of these tools. Instead, the insight into design complexity, hardware realism, and research precision is the core contribution of the work. In this chapter, we present the various configurations used when collecting data, presenting some limitations of our data collection, and discuss how we will use the collected data to investigate the research value of simulators and hardware prototypes.

We wish to evaluate how prototyping DoM on the BOOM has affected its performance. The BOOM has four supported design sizes: Small, Medium, Large, and Mega, but the Mega does not function on the FPGA cards we used for our experimental setup. Each size has a progressively larger core width, and also a deeper branch depth, more execution units and a larger ROB. This generally means that performance improves notably when going up a size, but increasing size can also make designing implementations more complex, as discussed in section 3.1. In order to be most comparable to the gem5 O3 CPU, which is balanced around an eight wide core, we use the LargeBoom, as despite its three wide core, is the largest core we have available.

In order to evaluate the differences in accuracy between simulators and hardware prototypes, a key metric of interest is the slowdown delta between the BOOM and gem5 versions. Slowdown delta is the difference in performance slowdown, and gives a good indication of whether the simulator and hardware implementation are similar. A small slowdown delta would indicate that the simulator gives an accurate approximation of the prototype, while a high slowdown delta would indicate that the simulator is inaccurate.

4.1.1 Notable Caveats

This research has some limitations, due to the complexity of the material we have worked with. Getting into and understanding computer architecture takes a considerable amount of effort in both reading previous work and developing the necessary experience with the tools used. In order to complete this work within time constraints, we have had to employ some shortcuts and some parts remain incomplete. We discuss these here, and discuss how to fix these for future research in section 5.2.

Firstly, the ISAs employed within this work are not identical. The BOOM uses the RISC-V ISA, while the work performed on gem5 uses the x86 ISA. This is for the sake of simplicity. The BOOM is naturally fully designed and realized for RISC-V and could not easily be changed, while gem5 does have some support for RISC-V as an ISA. However, as there is less support for this newer ISA compared to the long-maintained and more supported x86, it was deemed that the likelihood of complications would be too high and would consume too much time on tooling. However, these

differences are not judged to be notable for this work, due to the system ambiguous way gem5 is designed.

Secondly, as discussed later, the benchmarks are not identically applied across the BOOM and the gem5 implementation, due to gem5's considerably lower speeds. This might impact the results, but as the benchmarks are generally speaking consistent in performance behavior across most of their runtime, we expect this impact to be negligible.

4.1.2 SPEC2006 Suite

To evaluate our implementations, we chose to use the benchmarks from the SPEC2006 Suite. SPEC is a series of benchmarks that aim to give a detailed and representative collection of programs to represent current trends in the type of work that computers perform. SPEC2017 is the newest set of this suite, introduced in 2017 to replace the old SPEC2006 benchmark. However, the work of the original DoM was tested using many of the benchmarks in the SPEC2006 suite, and the 2006 suite remains very popular within the computer architecture community.

The reason for this is that SPEC is not primarily developed as a tool for architecture research, but rather as a benchmarking suite for hardware. The benchmarks are therefore large and can take a considerable amount of time to execute even natively on hardware. Most of the simulators employed in hardware architecture research are many orders of magnitude slower than realized hardware. This results in these simulators taking a prohibitively long time to complete benchmark execution. For this reason, researchers have adopted various methods by which to either select representative sections of the benchmarks or otherwise speed up execution. This is partly why the SPEC2017 suite is not ubiquitous in computer architecture research, as many of the resources developed for SPEC2006 are not euqally developed for the newer suite.

In order to get around the lower speeds that prevents executing whole benchmarks, a few different strategies have been developed. Most prominent among these are the process of profiling and checkpointing a benchmark. Profiling a program involves analyzing its runtime behavior, namely what the program spends its runtime doing. This can typically be achieved by monitoring how often a certain function call is invoked and how many cycles it takes for a program to return from such a function call. By these metrics, it is feasible to calculate a very accurate approximation of what parts of the code constitute how much of the program execution. Then, by modelling a smaller selection of these runs and then weighing them accordingly to frequency, it is possible to get highly accurate measurements of performance change without having to run an entire program. This is necessary in cases such as running a large program on gem5, in which full execution would take weeks or even months to run.

There are several complications with generating these representative runs. If one merely executes the program from a set point of execution, program execution would naturally be incorrect, as previous calculations would not have been completed. However, even if memory was updated with the correct values, starting execution in this manner would leave all caches cold, meaning they would operate at significantly lower performance than one would expect in a properly warmed up system. Therefore, it is necessary to both load the correct state into memory and to warm up the caches before execution.

Unfortunately, generating checkpoints and using them for experiments is challenging in several ways. Profiling takes a long amount of time and is not consistently reusable across different ISAs and configurations. Generating checkpoints also requires an extensive amount of execution time that might again be prohibitive unless they are executed at the start of research. As such, using checkpoints is not a viable option for this work. Instead, we adopt the same approach as the original DoM paper [1], in which the system is warmed up, and then we gather data. This creates some inaccuracy in the acquired data, although this inaccuracy should be very small.

We apply the technique of warming up the system for a considerable amount of time (1 billion instructions) and then gather statistics for a significant runtime after that (3 billion instructions). This will not be wholly representative, but as long as the simulated region is sufficiently large and is warmed up to prevent inaccurate performance from a cold system, the error margins are small.

Executed SPEC2006 CPUInt Benchmarks		
Benchmark Name	Simulated Work	Successful Workloads
400.perlbench	Based on PERL. Includes anti-spam checker, an email indexer and a comparison tool	3 / 3
403.gcc	Compiles program code into executable binaries	8 / 9
429.mcf	Schedules vehicles using simplex	1 / 1
462.libquantum	Simulates a quantum computer running Shor's algorithm	1 / 1
464.h264ref	Reference implementation of the video encoding standard	3 / 3
471.omnetpp	Uses the OMNet++ event simulator to simulate Ethernet for a large campus	1 / 1
473.astar	Pathfinding, including the A* algorithm	1 / 2
483.xalancbmk	XML processing to transform XML files into other documents	1 / 1

Table 4.1: The benchmarks that were used and how many of their reference size workloads that succeeded.

When running the benchmarks on BOOM realized on an FPGA, the performance is still slower than taped-out hardware, but considerably faster than when running a simulated system. For more details on this, see subsection 2.3.5, or the original FireSim paper [15]. The core benefit of this much faster system is the ability to realistically finish full runs of the SPEC2006 suite. This is a large benefit, as the most accurate representation of a benchmark is naturally completing the entire benchmark. This is also a topic of discussion in section 5.1. We therefore simply run all the benchmarks natively through the BOOM on an FPGA.

4.1.3 Data Results Collection

We planned to collect full performance and runtime data for the entire SPEC2006 suite, but this proved unfeasible due to development constraints in both time and hardware. During data collection, there is minimal information available about the state of execution, i.e. whether it has hung or is executing normally. As such, we utilized the SynthAsserts to abort execution upon a failed assertion, in order to prevent benchmarks from stalling indefinitely. Unfortunately, these assertions were not well-defined and they erroneously triggered for all floating point benchmarks. As such, only the integer benchmarks, SPECINT, which make up 12/29 of the benchmarks in SPEC2006 were possible benchmarks for results gathering.

In addition to the errors introduced by the DoM implementation, the baseline BOOM is not stable. The reason for this is two-fold: Firstly, the BOOM is developed as a research tool and primarily for projects as decided by the Berkeley BOOM-team. Therefore, there might be less motivation to maintain its stability than one would expect for a commercial processor, and a greater interest in doing sweeping changes. Secondly, stability when dealing with BOOM releases is considerably less rigorous than a commercial release. The BOOM team have not prioritized making all benchmarking suites function, as this is mostly not what the BOOM is used for. Rather, there is a selection of tests that the BOOM team deem suitable for their purposes, which determines whether a release is stable. As the team is relatively small, this means that some features will become unstable over time, as they no longer get adequate support.

Due to these two limitations, in addition to the remaining errors in our DoM implementation, only a minority of benchmarks from the SPEC2006 Suite ended up being usable on the BOOM. These are shown in Table 4.1. We still deem them to be satisfactory for the purposes of this work, and we highlight what limitations our limited selection of benchmarks introduce in subsection 5.1.2, as well as how to remedy the remaining errors in subsection 5.2.1.

4.1.4 Data Collection on the BOOM

For the BOOM, we use Linux distro images, generated by FireMarshal (see subsection 2.3.5 for more details), to run the benchmarks natively on the FPGA cards. We synthesized the BOOM implementation on the FPGA with SynthAsserts, using FireSim. We repurposed the build process for the SPEC2017 images maintained by EECS-NTNU [37] and used a similar setup to configure SPEC2006 benchmarks. We executed the image using FireSim and a custom harness developed for the FPGA cards available at the NTNU cluster. With this setup, we used a runscript that enabled us to run pre-defined suites, as well as individual workloads.

We originally attempted to run all the benchmarks on a single FPGA card, but several of the benchmarks failed during execution and forced us to manage the execution more directly: We split the workloads onto 4 separate cards and then attempted to run one benchmark at a time, monitoring which ones ran to completion and which ones failed. At the end, we mounted the images and retrieved all the performance data.

We then wanted to generate a baseline and attempted to run all the images on the unmodified BOOM image. However, as discussed in section 2.3, the SynthAsserts on the BOOM are not entirely stable, and we ran into some stability errors in the unmodified BOOM, even for the remaining SPECINT benchmarks. We had remedied many of these errors during development of the modified BOOM by removing the erronous assertions or changeing their parameters, but these changes were not sufficiently transferable. Therefore, we had to switch into running the unmodified BOOM without synth assertions and seeing that they ran to completion, which they did. However, this highlights the potential that several other benchmarks that previously failed due to SynthAsserts when running on the modified BOOM would have completed correctly. Without access to better tooling, it was not feasible to explore this further at the time.

4.1.5 Data Collection on gem5

In order to collect data on gem5, we executed the benchmarks which had successfully completed on the BOOM. We used a runscript and a parallelizing script, which dispatched each of the benchmarks independently. As gem5 is fully simulating the system internally in its execution, it is not affected by resource contention on its hardware platform. Therefore, the hardware and its system resource strain should not affect the results gathered by gem5. As discussed, we chose to warm up the system and then execute the benchmark, as the benchmarks were too large to run until completion with the lower speeds of gem5 compared to the BOOM.

Each benchmark executes independently, with a warm-up period of 1 billion instructions using the fast forwarding CPU, which runs the program on KVM. We then switch CPUs to the O3 CPU and run 3 billion instructions. Results are discarded during processor switch, meaning that only the 3 billion instructions executed under the O3 CPU affects them. The slice of 3 billion instructions should be a large enough size that any quirks of the benchmark will not overly affect the results. However, we have no statistical guarantee for this.

4.1.6 BOOM configuration

Outlined in Table 4.2 is the experimental configuration used. This is the standard configuration used for the Large BOOM. Note that to ensure stability, and to avoid the need to stall the frontend, the sizes of the shadow queue and release queue are both 50% larger than their untracked counterparts, the max branch depth and the number of LDQ entries respectively.

4.1.7 The gem5 simulator configuration

Outlined in Table 4.3 is the gem5 simulator configuration for the experiments. This configuration is not the same as that for the BOOM, as neither it should be, since the O3 CPU is configured

Parameter Name	Parameter Value
Fetch Width	8
Decode Width	3
ROB Entries	96
Integer Execution Units	3
Floating Point Execution Units	1
Max Branch Depth	16
Number LDQ Entries	24
Number STQ Entries	24
Number Shadow Queue Entries	24
Number Release Queue Entries	36
L1 Data Cache Size	32 KB
L1 Data Cache Associativity	8
L1 Instruction Cache Size	32 KB
L1 Instruction Cache Associativity	8
L2 Shared Cache Size	512 KB
L2 Shared Cache Associativity	8

Table 4.2: The configuration of the BOOM for the experiment run.

Parameter Name	Parameter Value
CPU Frequency	2.0 GHz
Fetch Width	8
Decode Width	8
ROB Entries	192
Integer Execution Units	8
Floating Point Execution Units	6
Max Branch Depth	Not Configured
Number LDQ Entries	32
Number STQ Entries	32
Number Shadow Queue Entries	64
Number Release Queue Entries	64
L1 Data Cache Size	64 KB
L1 Data Cache Associativity	2
L1 Instruction Cache Size	16 KB
L1 Instruction Cache Associativity	2
L2 Shared Cache Size	2 MB
L2 Shared Cache Associativity	8
L3 Shared Cache Size	16 MB
L3 Shared Cache Associativity	16

Table 4.3: The configuration of gem5 for the experiment run.

to support a much larger width than the BOOM. The O3 CPU is designed to be comparable to modern OoO processors, in which the corewidth has now reached 8 with some of the industry leaders, such as the Apple Cortex M1[9]. The O3 CPU is balanced around such a core width and we do not modify this parameter to avoid making unintended bottlenecks. Its fetch width is the same as the decode width, as it has less need to rely on a filled fetch buffer to forward instructions rapidly. Here the release queue has twice the max number of entries compared to the number of load entries, but experimentally we observe that it is never filled.

4.2 Results

This chapter presents the results of the simulator implementation and hardware prototype and discuss what insights this gives into the usefulness of both as research tools. First, we demonstrate that our implementation of DoM on both the BOOM and gem5 mitigate Spectre. We then present the summarized performance results for the two platforms, before diving into hardware specific results and gem5 specific results. Though the results themselves are presented here, the broader implications of these findings are discussed in section 5.1. We display each of the results to highlight the change in performance when enabling DoM in a system and the difference between performance changes when comparing the gem5 to the BOOM. Importantly, we also highlight what different forms of insight the different platforms provide.

4.2.1 Showing a successful mitigation

In order for the DoM technique to have been successfully implemented, we have to demonstrate that our modified version prevents Spectre attacks and that the unmodified version does not. A comprehensive study of the security implications of the modifications of DoM, including whether it introduces other weaknesses, is beyond the scope of this work. We demonstrate one attack against the BOOM, and one against gem5, and show that after mitigation, these attacks are no longer effective. The complete results of the runs are available in Appendix A. For this section we instead visualize the results to simplify the interpretation. We utilize different attacks for the two systems, as Spectre requires tuning in order to function on a given system, and these two attacks come pre-configured for the BOOM and gem5 respectively.

In figures Figure 4.1 and Figure 4.2, we see the results of a Spectre attack executed on the BOOM. The attack was developed by the BOOM team [38] and was later updated for newer versions of BOOM for a later workshop paper by another team [39]. In the first figure, we see that the attack on the unmodified BOOM successfully extracts the entire passphrase from the system. However, the modified version results in seemingly random guesses with a wide range of values and no consistent pattern. In the second figure, we see that the attack has a consistent high confidence in its guesses on the unmodified BOOM (except for the second to last guess), while the confidence for guesses against the modified BOOM is consistently low.

In figures Figure 4.3 and Figure 4.4, we see the results of a Spectre attack executed on gem5, the source code for which was developed by the Invisispec team [31]. In the first figure, it is clear that the attack on the unmodified gem5 implementation has successfully managed to exfiltrate the secret passphrase. On the modified, it has timed out and been unable to reveal any information. This is also reflected in the second figure, which shows zero confidence for the entire passphrase on the modified system, while showing a confidence between one and two for the unmodified system.

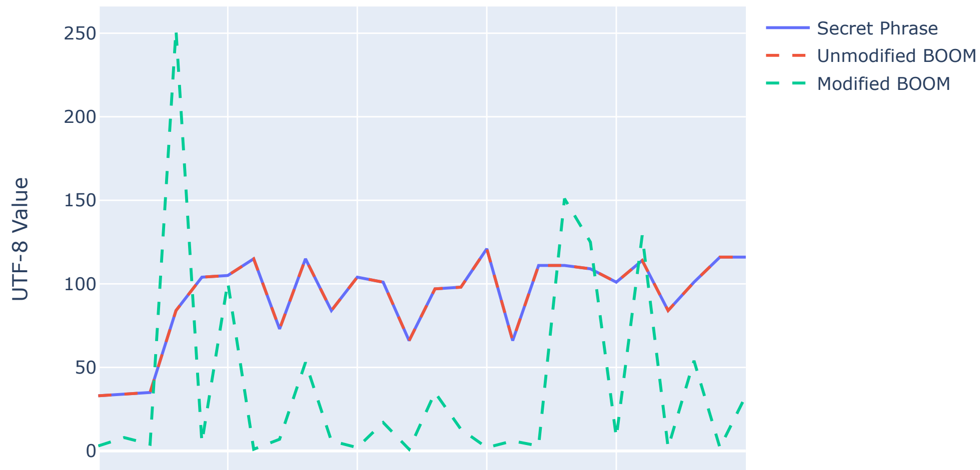

Figure 4.1: The results for the attack on the unmodified and modified BOOM. The secret is a passphrase, and the guessed values are their UTF-8 encoding.

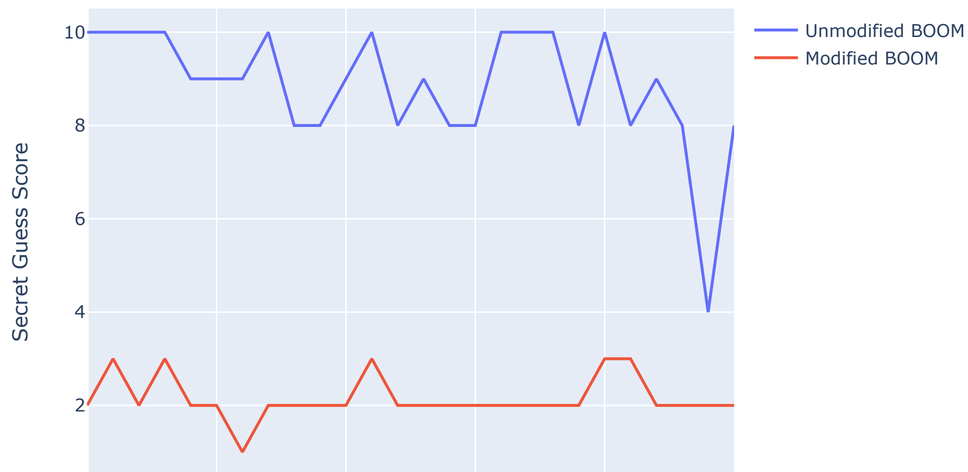

Figure 4.2: Confidence of guesses in system. Attack runs several iterations and increases confidence by one each time it gets a given value as the result. Low confidence indicates more random results.

Figure 4.3: The results for the Spectre attack on the unmodified and modified gem5. The secret is a passphrase, and the guessed values are their UTF-8 encoding.

Figure 4.4: The confidence for the Spectre attack on the unmodifed and modified gem5. 0 confidence indicates no information, while 1 indicates uncertainty. 2 indicates a highly likely guess.

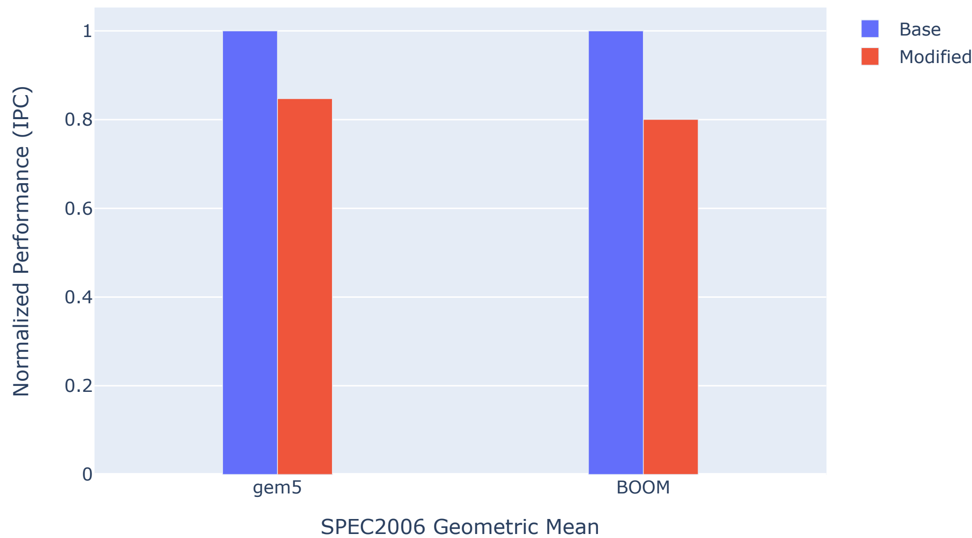

Figure 4.5: The performance results for both the BOOM and the gem5 implementation. A geometric mean is provided due to the variable results of the benchmarks.

4.2.2 Comparing Performance

In this section, we present the geometric mean performance slowdown for the BOOM and gem5. This is the geometric mean for the all the benchmarks from Table 4.1. It is clear that the performance slowdown is nearly identical for the two platforms: The BOOM implementation suffers slightly more slowdown, an even 20% in total, while gem5 reports 18% slowdown. We compare this to the original paper, in which there was a 19% mean performance loss for the DoM implementation on gem5. However, that version tracked all the types of shadows, while this work only tracks C-shadows.

For our results, the small performance delta indicates that to a large extent, the two platforms are experiencing the same sort of performance changes. The differences in performance across the two could be attributed to several causes. As the architecture and configurations for the two platforms are not the same, we expect there to be some difference among the performance impact of the two implementations. Significantly, the gem5 configuration is considerably wider than the configuration that was used for the BOOM. This, as well as the different cache properties, might result in some inherent performance variance, although in which direction is hard to estimate. To the best of our knowledge, no research investigating how DoM performance varies across core widths has been performed.

The performance of the individual benchmarks for the BOOM and gem5 are shown in subsection 4.2.3 and subsection 4.2.4 respectively. They are explored in more detail there, but we find that the performance change of each individual benchmark is highly similar across the two platforms, indicating that the performance impact of the BOOM seems relatively system-independent. This further strengthens the value of simulators as they are capable of not just providing highly similar total performance results, but also use their superior runtime insight to highlight why. This is shown clearly in later results, such as Figure 4.14.

The difference in performance between the BOOM and gem5 is relatively small, and it is hard to attribute this variance to one specific design difference. Functionally, the biggest difference between the two designs is the use of the MSHRs. The gem5 implementation checks the MSHRs to see

if there is already an outstanding request for the cache line, and in such cases allows the request to register itself on that MSHR. This means that potentially if both a shadowed and unshadowed load are requesting the same cache line, both will be served, opposed to the BOOM in which the shadowed load will only be served once it opportunistically retries. This is estimated to give around a 1-2% performance improvement for our benchmarks, and might explain the performance delta. Such optimizations are in use in DoM, and later works have explored this further. We go into more detail on optimizations in subsection 5.1.2.

4.2.3 BOOM Results

For the BOOM, the only available runtime results were those that were collectable through performance counters. We therefore look at the total amount of executed cycles and completed instructions in a benchmark, and use this to calculate instructions per cycle (IPC), which is the standard performance metric utilized for benchmarking. We split the performance results over two figures, in order to provide visual clarity.

Benchmark Performance

Figure 4.6 for `perlbench` and `gcc` display a fairly consistent trend. `perlbench` has only a small amount of slowdown, with `perlbench_1` even experiencing a very slight speedup. The occurrence of speedup is somewhat surprising, as one might assume that DoM could not have a positive impact on performance. However, if a branch prediction is erroneous and a load request that would normally have hoisted a mispredicted cache line from a lower level to the L1 cache gets blocked, there might be more relevant data in the L1 cache. For example, a loop that is incorrectly predicted to loop once more would continue bringing in cache lines into the cache hierarchy, thereby polluting the cache. DoM would prevent these incorrectly predicted loads from affecting the cache state.

`gcc` shows some variance, but lies on average between a 20% to 30% slowdown. The higher penalty in `gcc` would indicate two things. Firstly, `gcc` has a higher amount of speculation that ends up delaying loads. Some of these loads are critical, and as the original speculation was correctly predicted, but not resolved, there is a slowdown compared to the baseline.

Looking at Figure 4.7, we see greater variance on these other benchmarks. Both `mcf` and `libquantum` experience huge slowdowns, with `libquantum` being the worst performer, experiencing around 60% slowdown. `mcf` is a very memory dependent benchmark [40] that performs a lot of pointer chasing. It makes sense that many of these accesses will be speculative and be delayed by DoM, and we therefore observe a drastic slowdown. From previous profiling of the SPEC2006 benchmarks [40], we know there is a high amount of L1 cache misses in `libquantum`, which means that a high amount of loads that would normally access the L2 cache might now be delayed. We verify this by examining the information presented in Figure 4.13, later in this chapter.

`h264ref` has some variance between its three input runs, with `h264ref_1` seemingly experiencing a minor speedup. This speedup is larger than the one for `perlbench` and indicates that it most likely is not simply normal variance, but rather genuinely a performance improvement. The reasons for the improvement here, and indeed the general trend we see across all of `h264ref`, has to do with its highly regular nature, in which very long periods of work are executed. As `h264ref` is highly performance dependent on having the working set within the cache, DoM preventing the pollution of the cache will result in a speedup. This makes sense as video encoding is highly regular in nature.

`omnetpp, astar` and `xalancbmk` all experience slowdowns in the 30% range. This would indicate a considerable amount of delayed loads as a result of DoM, but not to the extreme levels seen in `mcf` and `libquantum`. These benchmarks are a midpoint between the two ranges of performance, as they are all somewhat reliant on speculative load requests. Interestingly, as we explore later, astar has a very low amount of delayed loads, but those delayed loads are very important for its performance.

Figure 4.6: The instructions per cycle normalized to the baseline of the unmodified BOOM. `perlbench` and `gcc` are shown together.

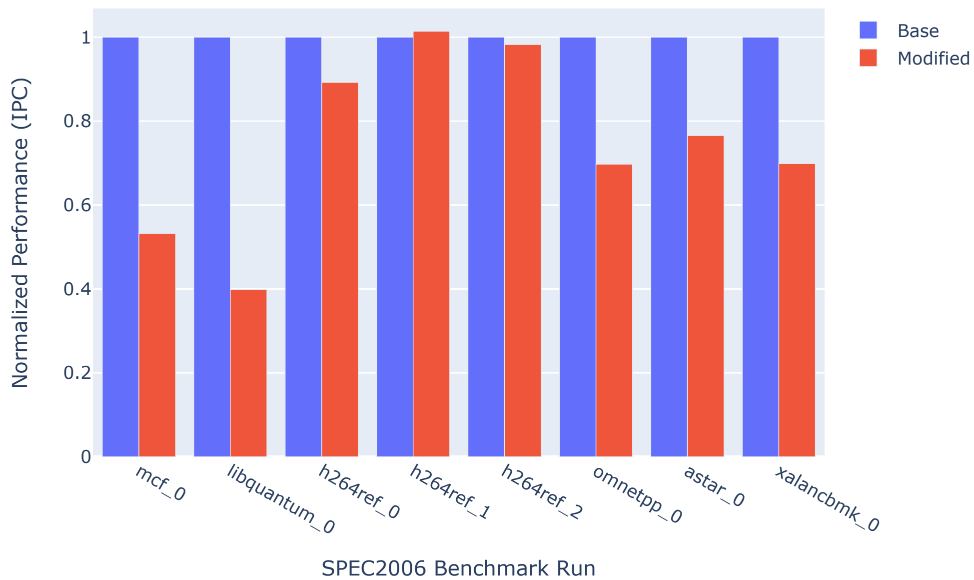

Figure 4.7: The instructions per cycle normalized to the baseline of the unmodified BOOM. `mcf`, `libquantum`, `h264ref`, `omnetpp`, `astar` and `xalancbmk` are shown together.

Component Name	Baseline BOOM LUT Usage	Modified BOOM LUT Usage
Boom Tile	229436	232350
Boom Core	159751	165568
LSU	19014	14433
Shadow Queue	N/A	1811
Release Queue	N/A	594

Table 4.4: Excepts from resource utilization analysis generated by Vivado.

As we see from the reuslts from the BOOM, there is large variance between the runtime profiles of the benchmarks executed. It is difficult to estimate whether the impact of DoM would be noticeable for workloads typical in personal computing, or whether only large-scale machines would suffer notable slowdowns. For the BOOM, we see in total a 20% geometric mean slowdown across the benchmarks, with the worst performer being `libquantum` with a 60% slowdown and the best performer being `h264ref_1` with a 2% speedup.

Size and Timing Results

We now introduce some unique results we generated that are only accessible on a hardware prototype. One of the advantages of a hardware implementation is that it has a physical resource utilization and timing summary, that gives us a more accurate representation of what the actual size overhead and timing restrictions of our implementation are. The full results from the size analysis run by Vivado is available in Appendix B, while the worst timing result, i.e., the worst path of our affected tile, is available in Appendix C. Note that as the place and route algorithm used by Vivado is not deterministic, these results should not be viewed as conclusive. Please see the associated discussion to properly understand what these numbers indicate.

In Table 4.4, we see resource usage of the baseline and modified BOOM. There are a few things to note from this information. As Vivado performs route and place according to a non-deterministic algorithm, without a guarantee for optimal placement, the two results are not directly comparable. As observed, the LSU takes a considerably larger amount of resources on the baseline BOOM, despite our modifications to the LSU only making it more complex. We outline some possible reasons for this.

It is possible that our design modifications to the LSU resulted in parts of its functionality no longer working as intended. If we introduced a trait that is a superset of another trait, the old trait will be removed by synthesis, and this can reduce resource usage. As we still have errors remaining in the design, we cannot guarantee that this is not the case, although we consider it unlikely, as our modifications are mostly independent of the previous design, outside of changes to the LSU issue arbitration.

We consider it more plausible that Vivado encountered a low-fitness generation when attempting to place and route the baseline BOOM, which the timing results also highlight. This means that Vivado ended up placing the hardware components in an inefficient manner and ended up using more LUTs than necessary.

With this in mind, we try to highlight roughly how many LUTs our changes to the BOOM utilize. The two queues, shadow queue and release queue, are naturally not present at all in the baseline BOOM, and therefore the 2400 LUTs they utilize are wholly resulting from DoM. These are organized under the total size of the BOOM core, which is roughly using 6000 more LUTs for the modified than for the baseline BOOM. As there are notable changes to the connectors and logic of both branch handling and the ROB, we expect that most of the changes in resource usage can be attributed to the implementation of DoM.

If we assume that our changes to the LSU ended up using no more LUTs than the baseline BOOM, the DoM implementation used 6000 more LUTs in total, based on the size changes to the BOOM core. Realistically, it is not feasible to know directly how many more LUTs the DoM implementation would use in an optimal placement, but the range of LUTs is likely to be around

Property Name	Baseline BOOM	Modified BOOM
Slack	6.299 ns	7.701 ns
Source	frontend/bpd/banked_predictors_1/ components_2/REG_9_reg/C	frontend/bpd/banked_predictors_0/ components_2/REG_15_reg/C
Destination	frontend/icache/ tag_array_4_reg/ENARDEN	frontend/icache/ tag_array_7_reg/ENARDEN
Requirement	33.333 ns	33.333 ns
Data Path Delay	26.718 ns	24.693 ns
Logic Levels	31	24

Table 4.5: Excerpts from the timing analysis generated by Vivado.

6000, and at least more than 2400 LUTs, due to the size of the release queue and shadow queue. As the BOOM tile, which is the full BOOM processor, uses around 230 000 LUTs, this gives us an estimate for the relative size overhead of DoM: At the minimum, based on 2400 LUTs, the size overhead is 1.04% for BOOM, but is more likely around 2.5%, based on the estimate of 6000 LUTs.

These size estimates give us a good estimate of the actual size overhead of DoM, and is more accurate than estimations based on simulator models, as they are unable to control for the extra resources that handling edge cases requires. As such, we consider the 2.5% estimate for the size overhead of DoM to be accurate for the BOOM, as it is a functioning prototype, although further verification of these results through more thorough placing and routing algorithms would be valuable.

Looking at Table 4.5, we also see some of the timing information for the Vivado timing analysis. This shows the worst timing result for paths going through the BOOM, which appears to be the connection between the branch predictor and the instruction cache. Although the baseline BOOM and the modified BOOM are reporting slightly different sources and destinations, this is due to the non-deterministic routing of the algorithm used by Vivado. It is clear that none of the changes introduced by DoM affected timing results, as they are not in the full path, as seen in Appendix C.

The timing results indicate a worse critical path for the baseline BOOM than the modified BOOM. As the full path does not contain any of the components that DoM has affected, it is likely that the difference in timing is the result of a low-fitness generation for the place and route algorithm used by Vivado. This strengthens our previous claim that the reduced LSU size was also a consequence of poor placement, as these are linked. Further research into the effects of DoM on timing results, primarily by giving the algorithm more time and using a more deterministic configuration, would be beneficial. However, it is very likely that the implementation of DoM does not affect timing requirements for the BOOM.

These hardware unique results indicate an advantage of hardware prototyping that is not present on gem5. Although there are ways to estimate size overheads for a design in gem5, these would not be able to predict nor adjust for extra hardware resources necessary for edge case handling, as those would not be present on the simulator implementation. As such, the resource utilization given here is more accurate and gives unique insights into the numerical performance qualities of DoM, as we have implemented it. These qualities will not be wholly transferable to other processor designs, but experienced hardware engineers will be able to more accurately estimate for other designs based on the details and results given here.

Figure 4.8: The instructions per cycle normalized to the baseline of the unmodified gem5 implementation using the O3 CPU. `perlbench` and `gcc` are shown together.

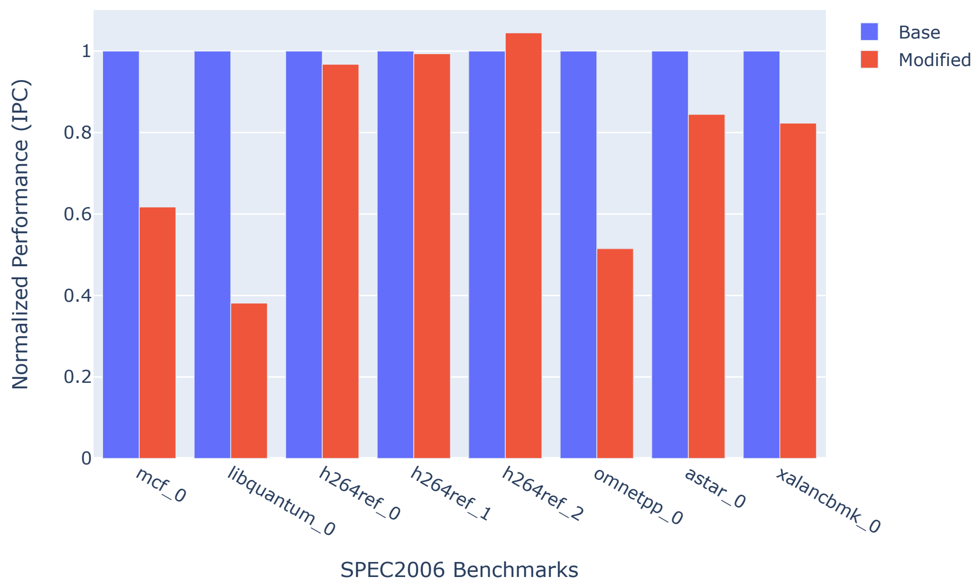

Figure 4.9: The instructions per cycle normalized to the baseline of the unmodified gem5 implementation using the O3 CPU. `mcf`, `libquantum`, `h264ref`, `omnetpp`, `astar` and `xalancbmk` are shown together.

4.2.4 gem5 Results

We now look at the gem5 results for the benchmarks. Unlike the BOOM, we are not reliant on performance counters in order to extract information about the runtime performance, and there is already a considerable amount of performance information available from gem5 statistics, in addition to what we add ourselves. Therefore, we present considerably more runtime information in this section, and delve more into the nature of the benchmarks and why they get the performance we observed. For a comparison of total performance results between gem5 and BOOM, see subsection 4.2.2.

Performance Results

Looking at the performance in Figure 4.8, we see a similar or better trend than we did with the BOOM. While `perlbench` remains relatively unaffected from the changes, some `gcc` benchmarks experience slowdown, although less than what we saw at the BOOM.

Similarly, in Figure 4.9 we see slowdowns from both `mcf` and `libquantum`, although the slowdown is slightly less for `mcf` and slightly more for `libquantum`. Both of the `h264ref` runs experience very similar performance to what we saw for the BOOM, in which they appear to be minimally affected, presumably due to the high level of regularity. However, here we do not observe a performance improvement for `h264ref_1`, indicating that the O3 CPU experiences a somewhat different runtime profile. Potentially, it is less prone to mispredicting branches and therefore manages to avoid the slowdown from inaccurate cache updates that the BOOM experienced.

Branch and Load Statistics

We now look at some of the more detailed statistics available thanks to the gem5 statistics system. We see in Figure 4.10 detailed information about the nature of control instructions, which are relevant for the shadow queue. Across `perlbench` the amount of branches, as well as the ratio between cleared and inserted branches, remains relatively stable. We also see that on average about 2-3 entries in the shadow queue are squashed per mispredict.

For `gcc` there is somewhat higher variance in the amount of branches as well as the ratio of mispredicts. We see a high amount of squashed entries compared to mispredicts, indicating that mispredicted branches often have a long resolution time or are at least closely followed by other branches. Interestingly, we see that `gcc_3`, which had the worst performance drop of `gcc` runs, has the lowest rate of misprediction. This makes sense, as it indicates that DoM will more often delay loads that would normally be completed without needing to wait for speculation to resolve.

The branch statistics for the other benchmarks in Figure 4.11 display a much higher variance, both in terms of amount of branches inserted, rate of misprediction and amount of entries squashed. For both `mcf` and `astar_1`, there is a higher amount of entries squashed than cleared in the shadow queue. This indicates a very low confidence branch predictor, yet also a notable amount of branches are squashed with every mispredict, which means that there are either tightly packed branches or long-latency on the mispredicting branch.

Interestingly, it is harder to see a direct correlation between the branch squash rate and the performance drop. `mcf` shows close to a 40% performance drop, while both of the `astar` runs experienced a little less than a 20% performance drop. However, the two `astar` runs have drastically different performance in terms of branch behavior, which means that the correlation is not consistent.

The other benchmarks display a very low amount of mispredicts and branch squashing. `libquantum`, which suffered a massive performance drop, seems completely regular with practically no branch mispredictions. `omnetpp`, which also had a massive performance drop, experiences a low amount of mispredictions, but is very comparable to the `h264ref_1` run, which did not experience performance drop. Overall, the branch information does not give full insight into why the performance drops we observe occur.

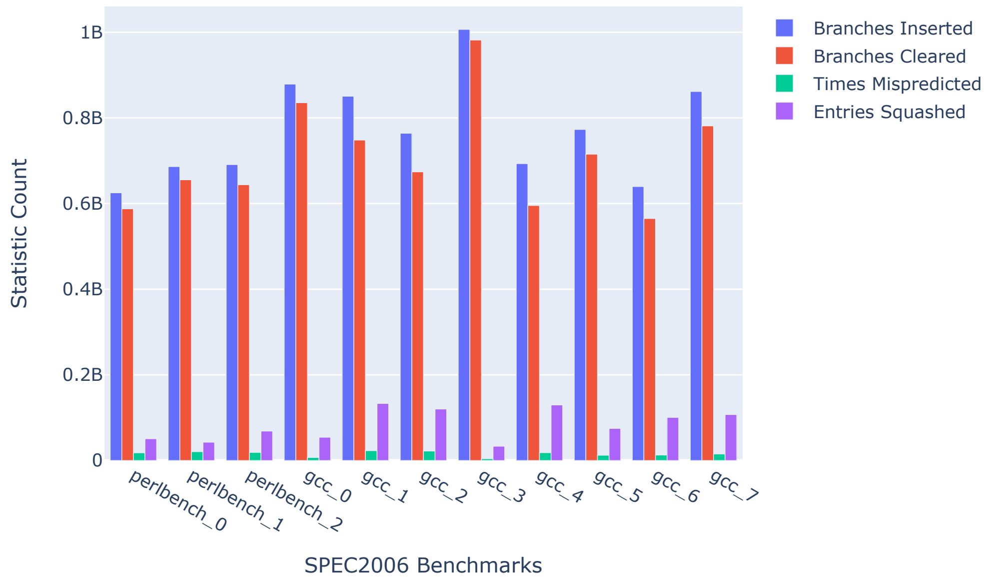

Figure 4.10: A myriad of branch statistics for the modified gem5 implementation using the O3 CPU. Branches Inserted are the number of control instructions inserted into the shadow queue, Branches Cleared are the number of branches declared safe in the shadow queue, Times Mispredicted are the amount of times a branch has been found to have been mispredicted, and Entries Squashed are the number of entries removed from the shadow queue as a result. `perlbench` and `gcc` are shown together

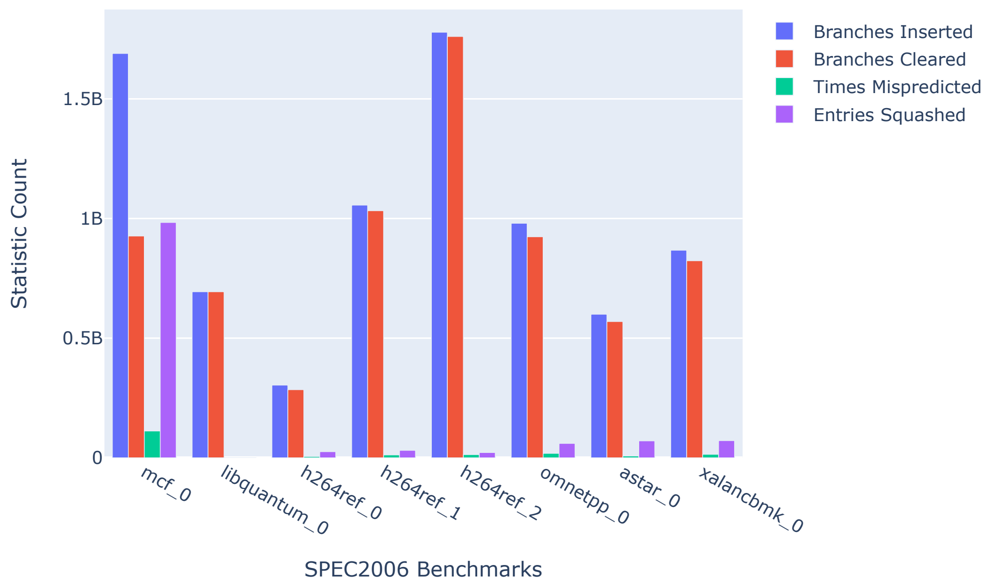

Figure 4.11: A myriad of branch statistics for the modified gem5 implementation using the O3 CPU. See Figure 4.10 for a full description. `mcf`, `libquantum`, `h264ref`, `omnetpp`, `astar` and `xalancbmk` are shown together.

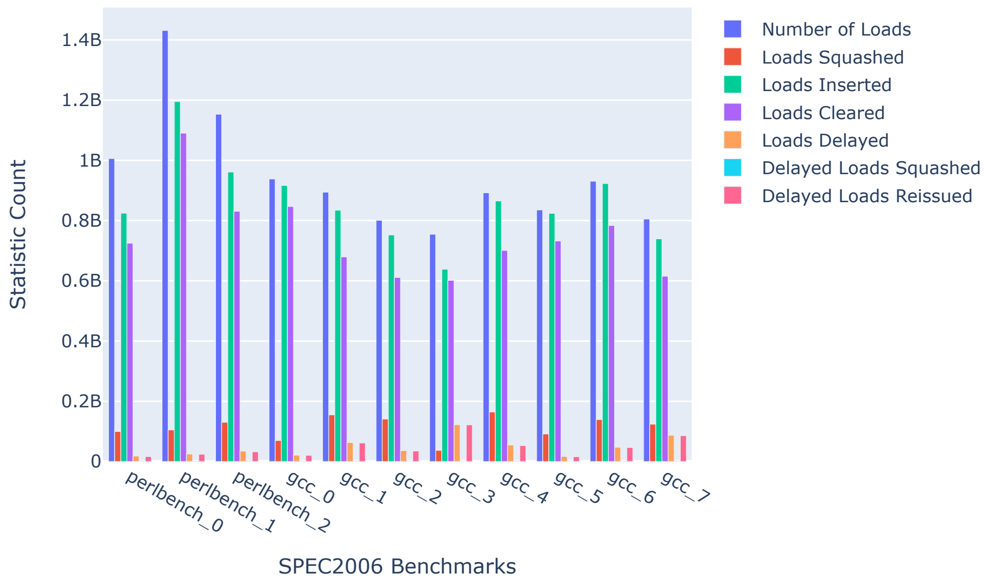

Figure 4.12: A selection of load statistics for the modified gem5 implementation using the O3 CPU. Number of Loads are the total number of loads that are entered into the processor, Loads Squashed the number of loads that were squashed for any reason, Loads Inserted the number of loads entered into the release queue, Loads Cleared the number of loads that were declared safe from release queue, Loads Delayed the number of loads that attempted to access memory and were delayed due to missing in the L1 cache, and squashed and reissued the number of these loads that were squashed and reissued respectively. `perlbench` and `gcc` are shown together.

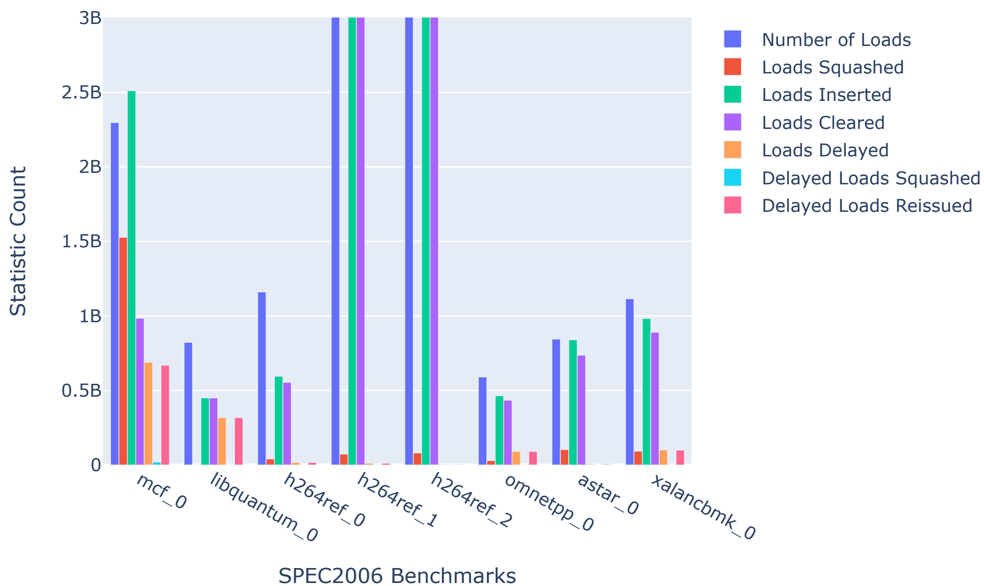

Figure 4.13: A selection of load statistics for the modified gem5 implementation using the O3 CPU. See Figure 4.12 for a full description. `mcf`, `libquantum`, `h264ref`, `omnetpp`, `astar` and `xalancbmk` are shown together.

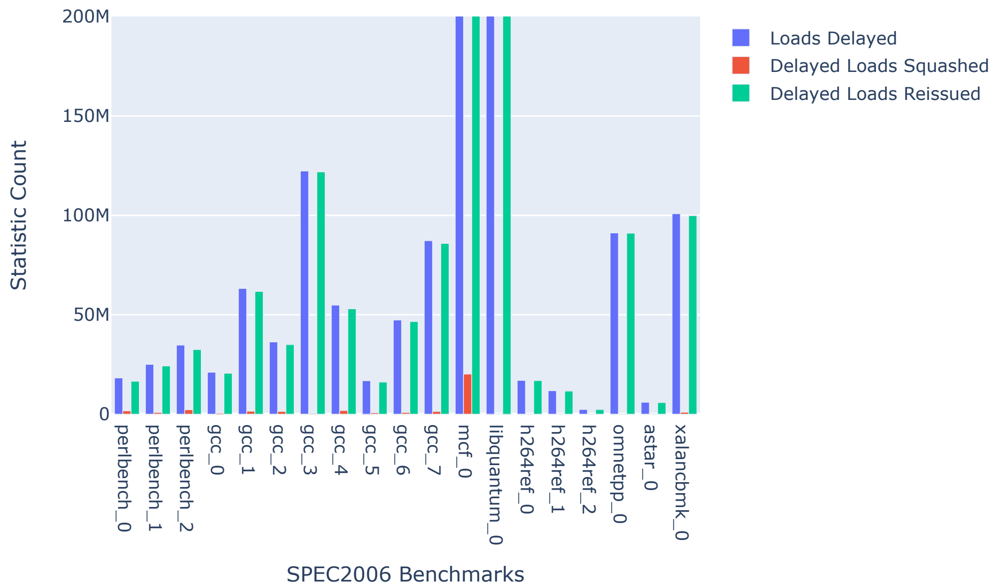

Figure 4.14: More detailed statistics about the delayed loads specifically, as their details were buried in their relatively low prevalence. The statistics are the same last 3 as for the previous two figures. All benchmarks are shown together.

Moving on to the detailed load statistics, it now becomes considerably clearer what is causing the slowdown. In Figure 4.12, we can see that the amount of delayed loads strongly correlates with the amount of slowdown that a given benchmark experiences. `gcc_3` has the highest amount of delayed loads and also the highest slowdown. Similarly, all the `perlbench` runs experience minimal slowdown and also have a very low amount of delayed loads.

We will delve more into the delayed loads statistics further down, but together with the branch statistics, we can see a clear pattern tying behavior to performance. Namely, when correctly predicted loads are delayed and have to be reissued later, this creates slowdown. This is to be expected.

Looking at the load statistics for the other benchmarks in Figure 4.13, we see a somewhat similar trend. Both `omnetpp, libquantum` and `mcf` have a relatively high ratio of delayed loads, with `libquantum` having the highest ratio of the three. This helps to explain their relative performance slowdowns, as they were the three experiencing the most by a large margin.

The unintuitive member of this group is `astar_0`. `astar_0` experienced approximately the same slowdown as `astar_1`, yet has nearly no delayed loads, especially not as a ratio of the total amounts of loads in the group. Similarly, we saw that the branch statistics for `astar_0` were very regular as well. Although more investigation would be required to fully understand why the performance drop is consistent across the two `astar` runs despite different statistics, it is likely that the load delays happen on a critical element that takes a long time to resolve. If the loads are consistently delayed for gathering a new tile for the `astar` algorithm, and the resolution for whether to move on to a new tile is long, it could have cascading effects.

Continuing, we see a very low amount of delayed loads for `h264ref`, which would correlate well with its low performance slowdown. Especially `h264ref_1`, which has an order of magnitude more loads, would logically suffer nearly no slowdown with so few delayed loads.

In Figure 4.14, we show the delayed load statistics. As the delayed loads typically make up a very small proportion of the total amount of loads, we choose to highlight only the delayed loads and

what happens to them. Both `mcf` and `libquantum` are cut short, not showing the total amount of delayed loads, as they are both several times larger than the rest of the benchmarks. Both of their ratio of delayed loads squashed remains minimal compared to delayed loads reissued, with `mcf` experiencing a few more squashed delayed loads than `lubquantum`.

The detailed statistics of the delayed loads make it clear that the vast majority of delayed loads will eventually be reissued. Only `mcf` and `astar_1` possess a large absolute amount of loads that are squashed, while in proportion we see a notable amount of squashes for `perlbench_0` and `astar_1`.

We see that for `h264, omnetpp`, and `astar_0` there are seemingly no delayed loads that are squashed, instead all delayed loads are later reissued. The vast difference in delayed loads for `astar_0` and `xalancbmk_0` while having similar performance changes requires a more thorough investigation, but the theory of critical loads being delayed for a longer period of time remains the most likely.

Chapter 5

Discussion and Evaluation

5.1 Discussion

This section will evaluate what insights the previous results reveal and will elaborate further on the core premise of this work, the differences and usefulness of between simulators and hardware prototyping as research tools. We will also discuss what our results tell us about DoM, and what development techniques that can be applied to minimize the inaccuracy of simulator research.

5.1.1 Result Trends from Hardware Prototype and gem5

We will now look at what the results tell us about the usefulness of simulators as research tools, particularly when considering the numerical accuracy of its data. Examining the results, it is clear that the gem5 results and the BOOM results are similar, even down to how much slowdown each individual benchmark experiences. There is a certain amount of variation in the specific numbers for a given benchmark, but this is to be expected, both because the underlying designs of the two systems are different, and because the configurations are different.

The general trend of benchmark performance holding across the platforms is encouraging for two reasons: Firstly, it indicates that it is possible to get highly similar results across different platforms when designing with hardware realism in mind. The large amount of similarity between the results from the benchmarks on the two platforms can be attributed at least partially to the similar design of the implementations on gem5 and the BOOM. Secondly, these results indicate that abstractions can maintain a level of realism that enables them to provide valuable quantitative value.

The accuracy of simulator results, combined with two other key factors, highlight why simulators can be a powerful research tool: Firstly, the considerably shorter development time allows for more research and more iterations to be performed within a reasonable time frame, therefore allowing for overall more and better research. Secondly, the more detailed data that were available during the gem5 simulation are valuable to researchers as a method by which to understand how their design changes impact a system.

The value of these two properties is important to highlight. It is not feasible to conduct preliminary and iterative research by using hardware prototypes unless the researchers involved have considerable practical experience with hardware engineering. Additionally, the greater amount of insight that was gained, especially concerning the nature of delayed loads, in the results for gem5 enable better reasoning about the impact of design changes. This shows why gem5 remains such a popular research tool for computer architecture research. This does not mean that it should be used without due care, but currently there are no feasible hardware alternatives that can provide the same level of ease-of-use and insight when performing architecture research. The results clearly indicate that although there might be some inaccuracy in the data, this inaccuracy is not disqualifying.

5.1.2 Evaluating Delay-on-Miss

This work has reproduced DoM in order to investigate the simulators and hardware prototypes as research tools. The reported performance of DoM has not been consistent when reproduced, such as for InvarSpec [35], and other techniques such as InvisiSpec [31] has had very different performance results when reproduced by other groups [32]. To better determine the approximate slowdown these techniques incur, more effort should be put into investigating why the reported results vary so much and what the likely performance range is. In this section, we discuss what our results mean for Dom, potential optimizations for DoM, and present what limitations this work has for evaluating DoM.

We first discuss the differences between this implementation of DoM and the other work. The original paper that introduced DoM [1] tracked more types of shadows than this work did, tracking the shadows for potential exceptions, data dependencies, and memory, referred to as E-Shadows, D-Shadows, and M-Shadows respectively. In the later work by Sakalis et al. [41], they find that the oldest active shadow over a speculative load was in most cases an M-shadow. This would indicate an expected performance degradation if our work covered more shadows. How much slowdown we can expect with more shadows is complicated to estimate.

Another work by Tran et al. [42], looked at how to improve the performance of DoM by lifting shadows earlier. They find that by reordering loads to reduce the duration for which M-shadows exist, performance improved by 7%. In addition, other work by Sakalis et al. [43] looked at the performance impact of coalescing MSHRs and found that it provides an average speedup of 1%. Coaclescing MSHRs means that a delayed load that is requesting a cache line that is already requested by an MSHR is woken up when that MSHR resolves. This is not possible to implement equivalently on the BOOM, which does not employ wakeup for loads under any circumstances.

By looking at these two optimizations together, it is clear that the performance on the BOOM should be at least 6% better than that of the original work, based on 1% slowdown from no MSHRs coalescing and 7% speedup by not tracking M-shadows. However, the updated work by the original authors [41] find an 18% slowdown, which should indicate at most a 10% slowdown for our BOOM and gem5 implementations, compared to the 18% and 20% we actually observed. This is a notable performance gap, as our results are nearly twice as bad as we expected to observe.

There are several factors that play into explaining why our performance is considerably lower than expected. Firstly, our design originated in a hardware context in which our development choices were more limited. As such, we might have implemented DoM in an overly restrained manner, and missed key performance enhancers. Alternatively, it might be that the original DoM implementation, which was developed on gem5, might be overly liberal in its implementation. We note that our performance results were consistent across gem5 and BOOM, which indicates that gem5 results might reflect hardware results. This is not certain, however, due to the difference in the configurations that the BOOM and gem5 implementations had.

With the BOOM implementation, the number of shadows and loads that can be released each cycle is equal to the width of the core. For the gem5 implementation, this number is equal to half the width of the core. We argue that this should only have a minor impact on performance, as the number of loads released in a cycle is enough to fully feed modern LSUs. For example, the Apple Cortex M1 has a maximum of three loads that can be issued in a single cycle, while we release up to four loads. For due diligence, investigating performance changes resulting from altering these parameters should be considered.

We expected the slowdown from DoM to be greater on the BOOM, as delayed loads on the BOOM are not woken up, but rather opportunistically rescheduled after they are no longer speculative. Our results indicate that the actual difference in performance impact from this is minor, as both the gem5 and the BOOM implementation experience the same amount of slowdown. This might also be because there is a relatively low amount of load contention by the time when the speculative loads are freed from the release queue, and they therefore get dispatched at a rate comparable to that of load wakeup.

As this work has not had adequate time to pursue optimizations, there are also likely to be

performance differences compared to earlier works where the DoM implementation was optimized over a longer period of time over the course of multiple publications [1, 41, 43, 42]. For example, it is possible to use the shadow tag of the release queue entry at its head, combined with whether the shadow queue is freeing any entries, to release and dispatch loads one cycle earlier than we currently are in certain circumstances. This is not an individual optimization that is likely to create a notable performance impact, but it is likely that many such optimizations exist and by not pursuing these, we are getting worse performance results.

In addition, we have to acknowledge that our results are limited in scope compared to the baseline we are comparing to. For our work, we could only use eight of the benchmarks in the SPEC2006 suite, while the most definitive DoM work [41] used 23 benchmarks. Of those 23 benchmarks, they reported only one other benchmark with more than 50% slowdown for DoM, namely `GemsFDTD`. The other two worst performing benchmarks, `mcf` and `omnetpp`, were both present in our results, giving a somewhat pessimistic benchmark selection. In addition, the previously discussed technique of MSHR coalescing, had a 12% performance difference specifically for `libquantum` [43], which also disproportionately negatively affects our performance results. We would expect to see an average performance improvement if more of the SPEC2006 benchmarks were included, but it is not possible to say by how much without running the actual experiments.

Finally, we discuss our contributions to the ongoing research around DoM. Most importantly, this work has verified that DoM is implementable on a CPU core. Simulator research is a valuable area to explore new designs, but we would expect complications when moving this work to an actual CPU core, as we discovered and outlined in for example section 3.1.2 and section 3.1.5. This work has now proven that it is possible to implement DoM on a CPU core, which has not been done for other Spectre mitigation techniques to the best of our knowledge.

In addition, this work has given further insight into some of the other properties of DoM. We have been able to evaluate the actual size changes a specific implementation of DoM has given, as shown in section 4.2.3. From the same tools, we also demonstrate that our implementation of DoM has no noticeable effects on critical paths within the design. This property is not necessarily transferable to higher frequency processors with stricter timing requirements, such as those at the forefront of the personal computing market.

We also note that this work is a partially independent reproduction of DoM. The original hardware prototype, on which much of the design of the gem5 implementation was based on, was done without any research artifacts provided by the authors of DoM. The gem5 implementation had some input from Christos Sakalis when considering how to check for the presence of a cache line in the L1 cache, but was otherwise based on the design from the hardware prototype. Despite this, we were able to get performance results that were much closer to the original results presented by the DoM authors than other reproductions such as InvarSpec [35]. Although this work is not conclusive as to what the performance range of DoM truly is, as outlined earlier in this section, our results indicate that the slowdown is more in the range of 20% than 50%.

5.1.3 The Validity of this Research

Near the beginning of this work, in subsection 1.2.1, we introduced several key properties we considered to be important to create good research. We will now discuss how this work satisfies the properties of *reproducible, comparable, well-presented*, and *transferable*.

This work goes into detail on both the method by which the work was performed and the vast majority of its intricacies. This would greatly aid anyone attempting to *reproduce* the work and enable them to stake out an efficient path for reproduction. In addition, the source code for the modifications made to the BOOM is available on GitHub [37], as is the source code for the implementation on gem5 [44]. This should enable anyone who wishes to investigate and compare the two DoM implementations.

In order to ensure that this work is *comparable*, we have utilized a standard benchmark suite for data collection. However, we note that we were unsuccessful in ensuring a large enough amount of the benchmarks succeeded to the point that we deem this work to be truly comparable for general

purposes. However, much of the qualitative insight presented throughout this work originates from current state-of-the-art platforms, such as the BOOM and gem5, and techniques, such as DoM, and we therefore deem it to be comparable to other work in computer architecture research.

We defined *well-presented* as meaning that dissemination of the research was comprehensible and unambiguous. Although these properties are inherently subjective to the person reading the dissemination, we believe that this work meets these criteria for the following reasons: The work provides all the necessary background and understanding of the utilized systems and techniques, as well as adhering to the terminology standard in the computer architecture field. In addition, the usage of figures and the processed results both aim to lower the cognitive load necessary to correctly understand and reason about this work. For the presentation of this work, effort has been taken to ensure that wording is as precise as possible. We have also highlighted all the limitations of this work, in an effort to make understanding our findings accurately as easy as possible.

Finally, we make the argument that this work both provides direct *transferability*, and is beneficial to other projects desiring transferability. Modern commercial processors are not available for academic research, and it is highly unlikely that our hardware prototype implementing DoM would be directly transferable to such a system. However, as the BOOM implementation is developed in hardware, it is likely to be much more representative of what an eventual hardware implementation would look like for these processors. We therefore consider that this work is useful for hardware engineers looking to implement DoM.

More importantly, this work provides several general considerations that aid the property of transferability for other research. By highlighting the design approaches that are utilized when working with hardware, we also demonstrate which techniques transfer well and how one can work to minimize the inaccuracies occuring when using simulators as a research tool. We discuss this further in subsection 5.1.5. For this reason, we deem this work to meet the property of *transferability.*

5.1.4 The Implementation Problem

The largest and most prevalent challenge of using simulators as a research tool, is the challenge of accurately modelling underlying hardware structures in sufficient detail that the premises of a work fo research is accurate. We have previously discussed the changes in size overhead when implementing DoM on a hardware prototype and why these changes were necessary, but we will now dive into more detail describing why these differences matter for research.

The difficulty of producing accurate results for both slowdown and size overhead when using simulators are one of the largest challenges computer architecture research must overcome to build up respectable knowledge bases. Most researchers in the community will choose to instead highlight the novelty of solutions instead of performance results for that exact reason, but quantitative results are important when examining research that aims to be *transferable.* A key motivation for computer architecture research is solving a problem efficiently, not just solving the problem. For that reason, it becomes necessary to generate data that are accurate and representative, both for future pursuits into deciding which forms of research are promising, but also for the commercial benefit of the research, which is one of the most direct positive impacts that computer architecture provides.

Therefore, with the importance of accurate and representative numbers, many of the abstractions provided by gem5 can prove detrimental. Although software methods can achieve highly precise results when working within a limited scope, once a considerable part of a system is dependent on abstractions, it becomes inevitable that emergent phenomenons occur. These complications are a result of the manner in which the individual abstractions work together. We provide two examples of these sorts of phenomenons.

First, the prevalence of software methods may make certain implementations considerably faster and smaller than is realistic in hardware. Functions such as search and roll back are both prohibitively expensive in hardware, such that rollback is only used in a minority of cases for the ROB on the BOOM. Despite this, they are commonplace in the software implementations in gem5. When these shortcuts are utilized without justifying how an eventual hardware implementation would

function, it makes the research less accurate. The size and time efficiency of these software methods might not be possible to match in a hardware implementation.

Secondly, the usage of global pointers and data objects translates poorly to a hardware model and contributes to obfuscating the impact of any given changes. While the BOOM implementation relied heavily on the use of synthesis to remove unused wiring, ultimately the synthesis would also reveal the actual changes to size in the case previously removed wires became utilized. This means that although there can be imprecision concerning size estimations, as the utilized hardware seems to already be available, a synthesis with size analysis would give the actual differences in resource usage.

Global pointers and data objects lower the accuracy of size overhead estimations. Despite being globally available, they provide no real sense of size changes as they are allocated globally and referenced through the runtime memory of the simulator, not through wiring in the simulated design. This prevents modifications from accurately portraying size overhead and wiring requirements. This creates a noticeable issue, as any research might easily make a wrong assumption, or indeed make no consideration, of how their changes would impact the actual size or timing of a functioning implementation.

To illustrate a potential problem this could cause, consider the following example of making changes to a cache policy based on micro-op properties. With a typical LSU, load requests going to the L1 cache would not issue the full micro-op, as this would be a considerable amount of information. However, it is fully possible that the micro-op is defined as being communicated in the hardware language, and then all but the accessed properties of the micro-op are synthesized away. Any modifications that use properties of the micro-op that were not previously accessed would result in synthesis not being able to discard as much of the micro-op. This would create an observable size difference in synthesis results. However, a simulator-based implementation might not have the same amount of clarity, as there would be less indication that this information would have been synthesized away on a hardware implementation, even if that was originally intended when designing the cache. This could lead to the research reporting that the modification created no size changes, despite this not being true in hardware.

These two problems together illustrate some serious limitations that exist uniquely within a simulator setting when performing research. Either of them can end up causing serious inaccuracies when measuring performance and size overhead, respectively, and contribute to misleading research results. That is not to say that simulators are not valuable, but rather that the quantitative insight into suggested modifications might be misleading and that research using simulators needs to discuss and reflect on these limitations.

In addition to the aforementioned challenges around the internal design methods of modifications, there are also limitations with data collection. We refer back to the discussion under subsection 4.1.2, in which we highlight the need for alternative approaches such as checkpointing when collecting data on a simulator due to its slower execution speed. This limitation can be adequately mitigated by proper methodology, but is not consistently addressed in research. Highlighting these limitations is key in order to adequately compare data, especially to hardware domains, such as in this work.

When working with the BOOM, simulator limitations are brought to light and are better understood. The inability to use global pointers and the requirements of more thoroughly understanding the underlying hardware, such as with rollback, allows for new understanding that can be ported back into the gem5 design process. Limitations such as the need to check for potential overwrites of the shadow queue and the release queue both provide interesting and unforeseen challenges of the given design, a process that is less likely to be discovered in simulators due to their level of abstraction.

Though the BOOM still has serious shortcomings in that it does not adequately replicate modern commercial processors, it remains highly valuable as it can force computer architects to discuss limitations and necessary modifications of their design. It is necessary to seriously dive into the challenging work to make an implementation hardware-ready, in order to properly be able to argue for the validity of a given design.

5.1.5 Minimizing inaccuracies

Many techniques can be applied to minimize inaccuracy when working with simulators such as gem5. Some of these relate specifically to the insights learned directly from the differences between BOOM and gem5, and these can be considered a core part of our contributions in this work.

The key strategy to minimizing inaccuracies when developing within a software context is being aware of what options are actually available in hardware. This does not mean that it is not wise to use the software abstractions that are available when working on a simulator, as those techniques can be invaluable for design speed. However, these techniques need to have a comparable implementation on hardware, and in cases where there is no equivalent implementation, the designer must carefully consider what ramifications this would have for the hardware implementation.

For example, if a software method utilizes a search method for its functional components, this could have several implications for the hardware design. If the searched structure is fairly small and does not grow in size when the processor design gets larger, it may be feasible to implement comparators on every entry. If the searched structure is not timing-sensitive and is handled relatively infrequently, it may be possible to compare one value each cycle, and eventually find the correct entry, while buffering incoming requests. If neither of these options are feasible, it might be necessary to reconsider how to implement the search function, considering other options such as keeping a reference to the desired entry.

All of these considerations require the designer to be aware of the limitations of hardware and how software abstractions they use translate to hardware design. If certain properties are absolutely necessary to make a certain design work, considerations for other, more specialized hardware components can be made and their ramifications examined. Many options can be considered and weighed, but the most important aspect is that they are actually considered, instead of passed by unexamined.

For these purposes, we introduce some general considerations to apply when conducting research using simulators. These considerations are not exhaustive, nor is it possible to provide a comprehensive list of considerations when designing for hardware realism. Rather, the best way to get an intuition for which hardware design considerations might be relevant for a given design is to implement an example design. The considerations offered here are focused more on flushing and cache interactions, as those were the key components that proved to be challenging for this work. Research more focused on the frontend of processors would want to pursue similar advice from practical work that involved those components.

First, it is important to consider the execution state of all changes introduced when making modifications to a system. If design changes create new structures, it is extra important to be aware of how to make these comply with the rest of the system, including mechanisms such as rollback and misprediction. If the work extends or alters another existing structure, consider whether this extra information is subject to the same mispredict and rollback handling as the rest of the existing structure, and if so whether that is already supported. These considerations can help reveal whether modifications need extra support in order to maintain valid states through misprediction. Importantly, this needs to be considered for all parts of the processor under which branches and rollback are relevant, as they are all subject to misprediction handling, but might not be necessary for parts such as the instruction fetch.

Second, consider whether the information used in a design changes is already available. Examples of this would be that information about the L1 cache, especially cache hits, would not be available in the L2 cache, as this would take a large amount of bandwidth. Therefore, whenever changes are made that relies on information not explicitly communicated through buffers, for example global instruction pointers, consider whether this information would logically be available. If it would not be present, consider whether it would be feasible to wire that information. Smaller signals, such as a single bit or a small bit bundle, might have a small design impact, while full micro-ops might require a large amount of resources. This is also relevant for higher level hardware languages, in which synthesis is used to remove superfluous signals. In these cases, the best practice is to see whether the unmodified design removed that information during synthesis.

Third, consider whether the timing requirements for both the rest of the affected system, and created structures, are likely to be met with a given design. This requires having some understanding of what the frequency of the designed for system is, but generally, large cascades of information should be pipelined to prevent missing timing requirements. For example, if information from one structure is used to access another structure, this should occur over two cycles, as information access through multiple registers typically is too slow to be done in one cycle. If there is a large amount of cascading information, the design should either be divided into more isolated components, or careful pipelining has to be applied across the entire design. This is easier to verify on hardware, where timing tools exist, but a general intuition should be enough to make reasonable assertions when working on simulators.

Fourth, consider whether the operations performed are realistic. Finding patterns within hardware is fully possible, but often relies on specialized hardware structures and approximations with potential conflicts, such as hashing conflicts. Finding complicated patterns within structures is typically not feasible without a form of staggering structure, such as those utilized by data prefetching algorithms [45]. Similarly, series of arithmetic operations need to be evaluated to assess whether they can be grouped together and collapsed to a single bitwise operation, or whether they need to be pipelined to meet timing requirements. There are a lot of good resources for clever bitwise operations, but this does not mean that complex functions can be resolved in a low amount of cycles, or without considerable size overhead. For example, floating point calculations take multiple cycles to resolve due to their complexity. More complicated operations need to either have a useful and direct translation to a hardware equivalent implementation, or be split into logical steps and pipelined.

Finally, consider whether any design changes are reliant on assumptions that might be inaccurate for a processor. If a design uses branch-related resources chronologically, but branches are resolved in any order, this can cause a difference in the internal state of the new structure and require additional support in order to function. Such differences between how a design operates compared to how other components operate can create the need for design modifications or extra resources. As a more general rule, it is therefore wise to check whether the assumptions and design of your methods conform with the assumptions and design of the processor in total.

5.1.6 The Value of Simulators

This work has been critical of the simplification that simulators employ, and some of the design decisions in gem5 and the O3 CPU. Nonetheless, we assess that there is great value in simulators. This section aims to highlight the usefulness of simulators as research tools for computer architecture research based on our experience with implementing DoM.

The biggest and most notable advantage of using a simulator is the drastic difference in design and iteration speed. For this work, although spread over a longer period with interruptions in between, it is estimated that developing the hardware prototype to its final state as discussed in section 3.1.5, took around 4 months. For comparison, the gem5 implementation took around 4 weeks. Naturally, this time would have been considerably shorter for more experienced developers. At the start of this project, we possessed no previous experience with either system, and the lower development time for gem5 would indicate that it is easier to learn. However, by the time we started the gem5 implementation, the BOOM implementation had been completed, giving valuable insights into the nuances of the DoM technique. This might have contributed to completing the gem5 implementation faster, making it hard to say if gem5 is definitively easier to learn.

There are many reasons for our experience of a considerably shorter development time for gem5. The gem5 simulator has a considerable suite of previously developed tools and paradigms to rely on, much more so than equivalent hardware platforms for research. Of note here is gem5's ability to harness the power of its large community and propel development forward through community efforts, which has resulted in a considerably higher level of quality than other research tools are able to maintain with their smaller teams.

Additionally, gem5's abstractions prove to be a useful aid in making systems more intuitive to

understand. Although they can still be unnecessarily convoluted, as with the load execution pipeline outlined in subsection 2.4.5, there are still considerable simplifications that provide a much lower barrier of entry when attempting to understand a given design. Specifically, it is considerably easier to follow both the flow of execution, and figure out what each part of the code is doing. As gem5 attempts to emulate the parallel execution in hardware through serialized execution, there is considerably less confusion around what is happening at any given point. By following the various function calls during a single cycle, it is possible to figure out what is happening and in what order. In comparison, on BOOM, all of this happens concurrently, and it is necessary to have a mental or visual model that displays and interprets what interactions are happening and why.

When working to improve correctness, the advantages of certain software attributes, such as function names, become clearer. When looking at a series of events in gem5, it is possible to piece together intended behavior from the series of function calls, as well as system information that is regularly communicated to the user. When working with the BOOM, one has to use the state values of hardware components to piece together the state of the processor and then attempt to figure out what the processor is currently doing. Logically, understanding and debugging the BOOM is as such considerably harder. In addition, a lot of this complexity of understanding the BOOM exists due to the complicated hardware design necessary to support concurrent execution. Checks such as dependencies between instructions in one cycle add a lot of complexity that simply does not have to exist in gem5, in which tasks are handled atomically, even within a single cycle.

The ability to quickly and accurately debug in gem5 saves a large amount of time over multiple iterations. While rebuilding gem5 took around 20 minutes for our work, rebuilding the RTL for Verilator took over an hour. Running gem5 is considerably faster than running RTL, and provides a stack trace as well as allowing the designer to insert print statements. In comparison, running Verilator is slower and produces only waveforms, and only when running in debug mode. Waveforms are considerably harder to debug as it is not obvious what specific method failed, rather just the state the hardware was in when it stopped executing. From this, it is necessary to investigate which component did not function as expected, and also how this functional error originated. This can often be a multistep process, in which it is necessary to track down the origin of the cascading error that eventually crashed the system through following several sets of invalid states.

While both gem5 and the BOOM could stall in an unfortunate manner that requires extensive investigation, carrying out this investigation is considerably easier on gem5. As the simulation speed of gem5 is considerably faster than that of Verilator, it is possible to explore benchmarks much more thoroughly, and the stack trace provided by the core dump proves invaluable in debugging. Meanwhile, if an error originates as a result of instructions deep within a program, Verilator is too slow to be used as a tool for debugging. Rather, it becomes necessary to utilize an FPGA run and collecting its limited information. This makes iteration even slower, as synthesizing on FPGAs took around four to six hours for our work, depending on the size of the BOOM configuration used. As there is limited information available by default, it becomes necessary to use techniques such as ILAs, but if those are not available, it might be difficult to efficiently find and fix errors.

Finally, the amount of support for gem5 and its toolchains is much comprehensive. C++ is a mature programming language with good resources, and a large selection of tools such as Valgrind and GDB exist to make debugging potential weaknesses easier. In contrast, hardware programming languages such as Chisel are still juvenile in terms of userbase and support. Similarly, there are many resources on how to use gem5, as well as an extensive amount of people currently using it for projects. Many of these people will answer user questions through community communication platforms such as forums and mailing lists. The corresponding communication platforms for Chisel, FireSim, and BOOM are much less used.

In addition to the faster development speed, there is a large benefit in the more detailed insights available during runtime in gem5. The large amount of available statistics, and the ability to easily add more, allow for much more detailed insights than the performance counter dependent hardware prototype. More specifically, the ability to easily add statistics to uncover the reasons behind unexpected differences in performance can be vital to developing quality research. These sorts of statistics are not easily available when working with hardware, and when they are, they are much harder to collect.

When considering the benefits and limitations of both hardware prototyping and simulators, it is clear that simulators serve a vital role in computer architecture research. Although their level of abstraction can creature inaccurate premises for research if not properly understood, they are invaluable for investigating new designs. As developing new techniques is a challenging endeavor in and of itself, it is important to not be hampered by miniscule hardware interactions, as the hardware prototype we developed was. Additionally, even though the performance precision of gem5 can be too inaccurate to be valuable to hardware engineers, it can still provide a preliminary range for what the eventual performance might be. Low-performing solutions can therefore be pruned early, and modifications and extensions that improve performance can be explored before committing fully to adopting them for new processors.

In addition, even after an eventual hardware implementation, or alternatively in response to developments in hardware, gem5 can be used to investigate why the performance of certain implementations arises. Although it would not be possible to investigate implementation specific phenomena with this approach, the core characteristics of memory level parallelism, execution unit utilization, cache hit ratios, and similar should generally give a good indication as to why certain performance changes occur.

As such, we posit the two general recommendations concerning the usage of simulators as a research tool within computer architecture: First, it is critical to be aware of the limitations of simulators, even more detailed ones such as gem5, as they can never be sufficiently accurate to give absolute confidence in the numerical results of research. The imprecise nature of the numerical results in computer architecture is well known within the field, but as it is not explicitly addressed by research dissemination, it can lead to ambiguity in understanding research. Second, it is of great importance to have done practical work with actual hardware prototyping or have studied in detail the inner workings of a hardware-realized processor. This should improve the design choices made by researchers to keep the models as fair and realistic as possible when considering eventual hardware implementation. Similarly, it is necessary to understand when shortcuts are employed, in order to reason about whether there really would be a viable implementation alternative in hardware and how those would function.

5.2 Future Work

This work is both an attempt at reproducing previous work and exploring the strengths and weaknesses of the various research tools available within the computer architecture community. However, although the work here is presented in detail and is valid within its constraints, there are notable limitations. In this section, we outline both work that was not completed due to hardware and time constraints, as well as other research that seems promising based on our findings.

5.2.1 Finalizing the FPGA implementation

As discussed at the end of section 3.1.5, the BOOM implementation of DoM is not finalized. Some errors remain in the design which prevent the completion of some benchmarks. Any future work aiming to develop this project further should firstly aim to locate and correct these errors. We present how to accomplish this.

Tools that would aid debugging, such as integrated logic analyzers (ILA), would be a necessity to finalize the BOOM implementation. ILAs allow for the selection of certain signals to be traced out of a system while it is running on an FPGA. By selecting the control signals for the shadow queue, one could figure out whether an invalid state is reached, such as SQ-Head and SQ-Tail becoming disjointed. Equivalent checks could be used for the release queue. In addition, this tool can allow developers to produce more general information as well through minor design modifications, giving insight into valuable properties such as active entries.

By enabling SynthAsserts and then using the ILA, one could get a fairly good overview of the state of DoM at the point at which the processor stalls. This would supply precise hints to figure out

what is going wrong, but might need to be supplemented with signals detailing information about the state of the LSU, the branch mask and the ROB to be truly effective. For reference, figuring out the need to stall branches to prevent overwriting shadow buffer entries took an extensive investigation into branch masks, SQ-Head, SQ-Tail and shadow queue indexes. Some amount of familiarity with hardware design would be necessary to be able to investigate these errors, as the information provided is not comprehensive.

Verifying that all errors have been rectified could be accomplishedby ensuring that all SPEC2006 benchmarks are successfully executed on the BOOM, running on an FPGA. This is not a guarantee for correctness, but would indicate a high level of robustness, which would be enough for the vast majority of use cases.

5.2.2 Improving the gem5 implementation

The gem5 implementation as presented provides the key functionality of the DoM implementation and allows for a large amount of flexibility for managing squashes and loads. However, it is a heavily abstracted model, relying on shortcuts present both within C++, the O3 CPU, and the gem5 simulator to function. Even within the scope of gem5 and the O3 CPU, there are many changes that could be made to make the gem5 implementation considerably closer to a realistic hardware implementation.

For ease of implementation and for gathering statistics, the shadow queue and the release queue are both implemented as vectors. Vectors are not a feasible structure within hardware, where the sizes of arrays would need to be statically set or otherwise allocated within a limited scope. Therefore, to more accurately represent a hardware design, it would be necessary to use cyclic arrays, as done on the BOOM. This would require statically initiating an array, instead of dynamically allocating entries to a vector, which is only possible in hardware through the use of expensive components such as SRAM. As these arrays should mimic hardware arrays, it would not be possible to store the full instructions either, but instead simplified references to the relevant data would be used. In addition, this would also require several other changes to remove the software shortcuts previously employed. We explore these, particularly the changes to inserts, mispredicts and squashes.

For the insertions to function when the arrays are both cyclic, it becomes necessary to implement RQ-Head and RQ-Tail, as on the BOOM. To insert a control instruction, we would now only store the active/inactive status in the shadow queue, and leave the corresponding instruction in the ROB. Similarly, to insert a load instruction, we would need to store the relevant load queue index into the release queue instead of a global pointer to the instruction.

This complicates the implementation in the LDST, as it now needs explicit connectors from the release queue to detect whether a load is speculative. The easiest would be to extend the load queue entries in the LSU, adding the speculative bit, that can be updated by the input from the release queue. This would be almost identical to how it is done on the BOOM.

The changes required to support mispredicts and squashes would be similar to those implemented on the BOOM. ROB entries would have to be extended with a shadow queue index, or otherwise there would need to be a mapping between an ROB entry and a shadow queue index in a separate buffer. Either approach would work, and they should be mostly identical in implementation size, although the separate buffer would require a tag comparison. Whenever a mispredict occurs, the index would have to be extracted and issued to the shadow queue, which would have to reset its indexes accordingly. This would then have to cascade to the release queue. The shadow queue would have to store a release queue reset index, as searching the release queue would not be transferable to hardware.

Squashing cannot be fully replicated in gem5 in the same manner as it works on the BOOM. This is because occasionally gem5 will squash individual instructions in a manner that is not feasible for a hardware system (again because it requires searching). These will have to be handled by removing the entries from the queues and then closing the hole and resetting the index according to the youngest valid entry. It would be an expensive operation on hardware, but it occurs relatively rarely and as it is a gem5 unique phenomena, it should not matter for implementation accuracy.

Ultimately, the largest noticeable changes would be within the memory system, in which the snooping would no longer be used. Instead, the caches would have to be redesigned to support the speculative parameter, that would prevent downwards propagation in the event of a miss and instead signal back a special nack. This would be comparable to how it is currently done in the BOOM, but is likely to impact performance less, as gem5 employs a wakeup system for memory calls. BOOM's opportunistic rescheduling has a generally lower performance than a wakeup based retry system.

As the gem5 ecosystem is fairly architecture ambiguous, it should not take much effort to move the system from an x86 architecture to a RISC-V architecture. However, some complications may occur when running benchmarks, as the RISC-V architecture currently has less legacy and support behind it. The lack of legacy means that there are less available resources such as guides, test sets, and similar. This can make it harder to develop for and debug issues, and was the primary reason why we did not use RISC-V as the ISA for our gem5 implementation. Even so, the transition should be performed or at least supported as en equivalent option, so that results between the BOOM and gem5 versions would be more comparable.

With these changes, the gem5 implementation would look very similar to the BOOM implementation. The benefit of this would be a highly comparable model between the two systems, which would allow for greater insight into what is causing the slowdown on the BOOM side and potentially allow for future research. In addition, it might reveal previous inaccuracies in the gem5 implementation. If there are no noticeable performance change between the current model and this future model, it would be fair to say that the abstractions were appropriate and accurate. However, if there is a noticeable performance difference, it would be worthwhile to investigate which assumptions that the abstractions relied on are inaccurate, and why they impact performance in the way they do.

5.2.3 Other Simulators

Implementing similar work in other simulators and comparing the performance differences would be highly beneficial in terms of understanding simulator accuracy and challenges. The more simulators are utilized to compare the results and attempt to reproduce this work, the better our collective understanding of the benefits and limitations each simulator provides would be. Preferably, there would be variety in which implementations are replicated across the simulators, but that is a large undertaking, and replicating only DoM would still be wortwhile.

Of special interest would be the simulators discussed in Butko et al. [46], namely Simics [47], PTLSim [48], SimpleScalar [49], and OVPSim [50]. These simulators represented a majority of utilized simulators for architecture research around 2011. We also recommend replicating this work in Sniper [51], which rose to prominence after 2011. Since that time, gem5 has most prominently risen in use, while most other simulators have received less attention as the community has become more homogeneous. However, there would be great benefits to see whether any of them provide similar or better precision for performance than gem5, and to see whether gem5 can learn from these other simulators.

Considerably more difficult to accomplish, but very interesting, would be to use the simulators employed by leading industry actors. Many industry actors have their own simulators or simulator-extensions, which they utilize for early-design profiling and development. Comparing their results to that of gem5, as well as an HDL implementation, would have great value.

5.2.4 Small Delay-on-Miss with only C-Shadows

As shown in the hardware implementation, section 3.1, one of the big advantages of attempting to create a hardware prototype are the key insights that would not otherwise be available. The most important among these insights is using existing systems to make a solution that remains minimal in its changes to a system. It is understandable that systems which are highly conforming to existing hardware would be beneficial both in terms of size and energy efficiency, as well as

development time. Therefore, we highlight a size-efficient implementation of DoM on the BOOM, when only tracking C-shadows.

The first insight is recognizing the large amount of support already provided to branch handling under the BOOM. This is most likely a fairly standard approach of doing things, as mispredictions need to be killed as soon as possible and the pipeline refilled to maintain high throughput. We therefore see it as very reasonable that most modern processors would have some mechanism to kill all mispredicted instructions in a small amount of cycles. We expect similar methods for communicating about which branches are now safe. Therefore, by extending on such existing support, we could implement DoM when only using C-shadows with lower overhead in the following manner.

Currently, DoM tracks all shadows in the shadow queue, with the release queue tracking all loads. In most systems, it would be possible to instead track the speculative status of loads directly in the load queue. In order to kill mispredicted loads, there has to be information stored detailing which control instructions a load is dependent on, or at least the youngest control instruction one. Using this information, it becomes possible to discern which loads are dependent on at least one unresolved C-shadow, or it should be possible with only small modifications to the LSU. These loads are speculative and should be blocked from accessing a lower level of cache if they miss in the L1 cache and should always be blocked from altering cache state, as with all speculative loads under DoM.

This approach has several advantages compared to the current implementation of DoM. It remains isolated nearly entirely within the LSU and the memory system, instead of relying on both the frontend, execution stage and ROB. This reduction in scope requires considerably less design modifications across the processor, as well as less complex logic to handle complicated dependencies. In addition, the implementation uses only mechanisms already present, potentially with small modifications, and should as such have a much lower implementation overhead.

This is an important consideration for hardware engineers. As observed for the Samsung Exynos [27], the hardening of the BTB by using hashing was implemented, despite being a non-comprehensive solution, simply because it could be implemented with minor modifications and provided adequate security. A similar argument could be made for DoM, which is non-comprehensive, but provide adequate security and requires relatively few design modifications.

Chapter 6

Conclusion

This work has evaluated the usefulness of simulators and hardware prototypes as research tools, by reproducing a state-of-the-art security technique on both the BOOM and in the popular gem5 simulator. Through a comprehensive investigation into the key differences in design between simulators and hardware prototyping, we have critically examined the key challenges that can make simulator research inaccurate. We have also evaluated the limitations of hardware prototyping as a research tool for computer architecture research.

A comprehensive background has been provided, which introduced elements of computer architecture research, hardware design, the BOOM, gem5, and other common tooling and conventions. This work covers a rudimentary understanding and some key nuances of each of those topics. We discussed some aspects of computer architecture research, and why researchers utilize simulators. We have explained abstract models for various processor designs, including traditional in-order designs, superscalar designs and Out-of-Order designs. We examined what the differences between these designs are, why they are necessary and what implications they have.

We introduce the Berkeley Out-of-Order machine, and explain all its relevant components, including its inspirations, its abstract design, and the details of its execution stage, reorder-buffer, load store unit and its cache. We presented the gem5 simulator and its Out-of-Order CPU and looked at its core design philosophy, as well as the complicated nature of its load execution schedule. Finally, we introduced the core tooling and conventions utilized for research in these fields, including the necessary hardware tools and ecosystems for the BOOM, such as Chipyard and FireSim.

We explained transient attacks and introduced the core elements of how transient attacks work. We introduced and discussed a variety of the techniques that seem promising within the field, and then took a deeper look at the state-of-the-art technique Delay-on-Miss. DoM was chosen as a suitable example for both comparison and reproduction in order to examine the accuracy of simulators.

We provided an extensive deep-dive into the details of prototyping the Delay-on-Miss technique on the BOOM, including design challenges and the limitations of hardware prototyping, across three different core widths. We explored how the abstract model of DoM gets changed in order to meet essential requirements in hardware, such as misprediction. The value of more hands-on experience is examined through these examples and showing how unique challenges with DoM, such as the need to stall branch dispatches, are discovered only in this more detailed work.

A similar implementation into gem5 was performed and discussed. We highlighted its considerably shorter development time and faster debugging, demonstrating both the increased ease by which designs can be implemented in gem5, and the lower barrier of access. The statistics sampling of gem5 is discussed and proposed as a key advantage of simulator research. This implementation, as well as the BOOM, are then both tested on a selection of the benchmarks from the SPEC2006 Suite and their results are compiled and examined, revealing highly similar total performance changes, as well as highly similar trends for specific benchmarks. The detailed statistics provided by gem5 proves key into explaining most of these performance changes, although not conclusive. The unique

hardware analysis available on the hardware prototype demonstrate more accurate size overhead estimations, and furthers research into DoM.

Using these results, several important insights are discussed. The similarities of the results from the two research tools indicate a high level of accuracy in simulator-based research when hardware limitations are kept in mind. The many challenges of successfully implementing a modification on hardware are discussed, as well as how researchers can minimize inaccuracies in research by being conscious of equivalent hardware designs. We affirm simulators as comprehensive research tools that provide detailed insights not available elsewhere, and that their faster design process is vital for research.

We conclude that the value of hands-on work remains high for any computer architecture researcher as a valuable way to build intuition for hardware realism. However, we note that not every research project needs to pursue a hardware prototype to be viewed as accurate, even when considering numerical performance. This is because the accuracy of simulators is found to be high, even for numerical performance, when hardware limitations are considered during the design process.

Ultimately, we highlight that the core weaknesses of simulators, such as gem5, are not an intrinsic property of the simulator, but rather stems from its ability to allow for shortcuts, which might not be transferable to a hardware implementation. A strong understanding of processor design and hardware is therefore necessary to effectively avoid these pitfalls, and understand and acknowledge the limitations of research tools needs to be a core consideration for researchers going forward.

Bibliography

[1] Christos Sakalis et al. 'Efficient invisible speculative execution through selective delay and value prediction'. In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2019, pp. 723–735.

[2] Jerry Zhao et al. 'SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine'. In: (May 2020).

[3] *Artifact Review and Badging - Current*. Aug. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. 6th. Elsevier, 2011.

[5] Björn Gottschall, Lieven Eeckhout and Magnus Jahre. 'TIP: Time-Proportional Instruction Profiling'. In: *To Appear In, Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021.

[6] Hadi Esmaeilzadeh et al. 'Dark silicon and the end of multicore scaling'. In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[7] Alon Amid et al. 'Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs'. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[8] RISCV-BOOM. *RISCV-BOOM Documentation*. URL: https://docs.boom-core.org/en/latest/.

[9] Andrei Frumusanu. 'Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14'. In: (Nov. 2020). URL: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive.

[10] *SiFive TileLink Specification*. SiFive, 2020.

[11] Henry Cook, Wesley Terpstra and Yunsup Lee. 'Diplomatic design patterns: A TileLink case study'. In: *1st Workshop on Computer Architecture Research with RISC-V*. 2017.

[12] Krste Asanović and David A Patterson. 'Instruction sets should be free: The case for risc-v'. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[13] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 20191214-draft. RISC-V Foundation, Dec. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/draft-20210811-a37624e/riscv-spec.pdf.

[14] *GRLIB IP Core User's Manual*. July 2021. URL: https://gaisler.com/products/grlib/grip.pdf.

[15] Sagar Karandikar et al. 'FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud'. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[16] *Energy Efficient Computing Systems (EECS)*. URL: https://www.ntnu.edu/ie/eecs.

[17] Nathan Binkert et al. 'The gem5 simulator'. In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[18] Milo MK Martin et al. 'Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset'. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[19] Jason Lowe-Power et al. 'The gem5 simulator: Version 20.0+'. In: *arXiv preprint arXiv:2007.03152* (2020).

[20] Yasir Mahmood Qureshi et al. 'Gem5-X: A Gem5-based system level simulation framework to optimize many-core platforms'. In: *2019 Spring Simulation Conference (SpringSim)*. IEEE. 2019, pp. 1–12.

[21] Richard E Kessler. 'The alpha 21264 microprocessor'. In: *IEEE micro* 19.2 (1999), pp. 24–36.

[22] Aamer Jaleel et al. 'High performance cache replacement using re-reference interval prediction (RRIP)'. In: *ACM SIGARCH Computer Architecture News* 38.3 (2010), pp. 60–71.

[23] Paul Kocher et al. 'Spectre attacks: Exploiting speculative execution'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[24] Claudio Canella et al. 'A systematic evaluation of transient execution attacks and defenses'. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 249–266.

[25] Esmaeil Mohammadian Koruyeh et al. 'Spectre returns! speculation attacks using the return stack buffer'. In: *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*. 2018.

[26] *Intel Analysis of Speculative Execution Side Channels*. Jan. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[27] Brian Grayson et al. 'Evolution of the samsung exynos cpu microarchitecture'. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 40–51.

[28] Gururaj Saileshwar and Moinuddin K Qureshi. 'Cleanupspec: An" undo" approach to safe speculation'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 73–86.

[29] Vladimir Kiriansky et al. 'DAWG: A defense against cache timing attacks in speculative execution processors'. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 974–987.

[30] Daniel Gruss et al. 'Kaslr is dead: long live kaslr'. In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 161–176.

[31] Mengjia Yan et al. 'Invisispec: Making speculative execution invisible in the cache hierarchy'. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 428–441.

[32] Jiyong Yu et al. 'Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 954–968.

[33] C. Sakalis et al. 'Ghost loads: what is the cost of invisible speculation?' In: *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. May 2019, pp. 153–163.

[34] Kevin Loughlin et al. '{DOLMA}: Securing Speculation with the Principle of Transient Non-Observability'. In: *30th {USENIX} Security Symposium ({USENIX} Security 21)*. 2021.

[35] Zirui Neil Zhao et al. 'Speculation invariance (invarspec): Faster safe execution through program analysis'. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 1138–1152.

[36] Xida Ren et al. 'I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches'. In: ().

[37] *EECS NTNU — RISCV-BOOM — lobo-bump*. URL: https://github.com/EECS-NTNU/riscv-boom/tree/lobo-bump.

[38] RISCV-BOOM. *BOOM Speculative Attacks*. URL: https://github.com/riscv-boom/boom-attacks.

[39] Majid Sabbagh and Yunsi Fei. 'Secure Speculative Execution via RISC-V Open Hardware Design'. In: *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*. 2021.

[40] Sarah Bird et al. ‘Performance characterization of SPEC CPU benchmarks on Intel’s Core microarchitecture based processor’. In: *SPEC Benchmark Workshop*. 2007, pp. 1–7.

[41] C. Sakalis et al. ‘Understanding Selective Delay as a Method for Efficient Secure Speculative Execution’. In: *IEEE Transactions on Computers* 69.11 (Aug. 2020), pp. 1584–1595.

[42] K.-A. Tran et al. ‘Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design’. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Oct. 2020, pp. 241–254.

[43] C. Sakalis, M. Själander and S.Kaxiras. ‘Seeds of SEED: Preventing Priority Inversion in Instruction Scheduling to Disrupt Speculative Interference’. In: *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. Sept. 2021.

[44] *EECS NTNU — gem5 — Delay-on-Miss*. URL: https://github.com/EECS-NTNU/gem5-dom.

[45] Marius Grannaes, Magnus Jahre and Lasse Natvig. ‘Storage efficient hardware prefetching using delta-correlating prediction tables’. In: *Journal of Instruction-Level Parallelism* 13 (2011), pp. 1–16.

[46] Anastasiia Butko et al. ‘Accuracy evaluation of gem5 simulator system’. In: *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*. IEEE. 2012, pp. 1–7.

[47] Peter S Magnusson et al. ‘Simics: A full system simulation platform’. In: *Computer* 35.2 (2002), pp. 50–58.

[48] Matt T Yourst. ‘PTLsim: A cycle accurate full system x86-64 microarchitectural simulator’. In: *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. 2007, pp. 23–34.

[49] Todd Austin, Eric Larson and Dan Ernst. ‘SimpleScalar: An infrastructure for computer system modeling’. In: *Computer* 35.2 (2002), pp. 59–67.

[50] *Technology OVPsim*. URL: https://www.ovpworld.org/technology_ovpsim.

[51] Trevor E. Carlson, Wim Heirman and Lieven Eeckhout. ‘Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations’. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2011, 52:1–52:12.

Appendix A

Results of Running Spectre Attacks

This appendix shows the results of running a Spectre attack on both the unmodified and modified BOOM and gem5 to show that the mitigation we implemented successfully mitigated the attack. The attacks for the BOOM were developed by the BOOM team [38] and updated for newer BOOM versions for later academic work by a separate team [39]. The attacks for gem5 were developed by the Invisispec team [31].

```
tracers init
Commencing simulation.
m[0x0x80002770] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(2, 4, )
m[0x0x80002771] = want(") =?= guess(hits,dec,char) 1.(10, 34, ") 2.(2, 4, )
m[0x0x80002772] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(2, 179, )
m[0x0x80002773] = want(T) =?= guess(hits,dec,char) 1.(10, 84, T) 2.(3, 160, )
m[0x0x80002774] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 0, )
m[0x0x80002775] = want(i) =?= guess(hits,dec,char) 1.(9, 105, i) 2.(2, 7, )
m[0x0x80002776] = want(s) =?= guess(hits,dec,char) 1.(9, 115, s) 2.(2, 8,)
m[0x0x80002777] = want(I) =?= guess(hits,dec,char) 1.(10, 73, I) 2.(3, 7, )
m[0x0x80002778] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(2, 2, )
m[0x0x80002779] = want(T) =?= guess(hits,dec,char) 1.(8, 84,  T) 2.(2, 5, )
m[0x0x8000277a] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 62, >)
m[0x0x8000277b] = want(e) =?= guess(hits,dec,char) 1.(10, 101,e) 2.(2, 3, )
m[0x0x8000277c] = want(B) =?= guess(hits,dec,char) 1.(8, 66,  B) 2.(3, 181, )
m[0x0x8000277d] = want(a) =?= guess(hits,dec,char) 1.(9, 97,  a) 2.(2, 2, )
m[0x0x8000277e] = want(b) =?= guess(hits,dec,char) 1.(8, 98,  b) 2.(2, 1, )
m[0x0x8000277f] = want(y) =?= guess(hits,dec,char) 1.(8, 121, y) 2.(3, 5, )
m[0x0x80002780] = want(B) =?= guess(hits,dec,char) 1.(10, 66, B) 2.(3, 18, )
m[0x0x80002781] = want(o) =?= guess(hits,dec,char) 1.(10, 111,o) 2.(2, 6, )
m[0x0x80002782] = want(o) =?= guess(hits,dec,char) 1.(10, 111,o) 2.(2, 20, )
m[0x0x80002783] = want(m) =?= guess(hits,dec,char) 1.(8, 109, m) 2.(2, 4, )
m[0x0x80002784] = want(e) =?= guess(hits,dec,char) 1.(10, 101,e) 2.(2, 1, )
m[0x0x80002785] = want(r) =?= guess(hits,dec,char) 1.(8, 114, r) 2.(2, 2, )
m[0x0x80002786] = want(T) =?= guess(hits,dec,char) 1.(9, 84,  T) 2.(2, 3, )
m[0x0x80002787] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(2, 40, ()
m[0x0x80002788] = want(s) =?= guess(hits,dec,char) 1.(4, 115, s) 2.(2, 4, )
m[0x0x80002789] = want(t) =?= guess(hits,dec,char) 1.(8, 116, t) 2.(2, 62, >)

*** PASSED *** after 27288002 cycles
time elapsed: 1.3 s, simulation speed = 21.08 MHz
FPGA-Cycles-to-Model-Cycles Ratio (FMR): 1.42
Ran 27288002 cycles (fastest target clock)
[PASS] FireSim Test
SEED: 1625660998
```

Figure A.1: Spectre attack on the unmodified BOOM. Extra spaces have been inserted to improve readability.

```
tracers init
Commencing simulation.
m[0x0x80002770] = want(!) =?= guess(hits,dec,char) 1.(2, 3,    ) 2.(1, 1, )
m[0x0x80002771] = want(") =?= guess(hits,dec,char) 1.(3, 8,    ) 2.(2, 3, )
m[0x0x80002772] = want(#) =?= guess(hits,dec,char) 1.(2, 4,    ) 2.(2, 5, )
m[0x0x80002773] = want(T) =?= guess(hits,dec,char) 1.(3, 252,  ) 2.(2, 7, )
m[0x0x80002774] = want(h) =?= guess(hits,dec,char) 1.(2, 5,    ) 2.(2, 122, z)
m[0x0x80002775] = want(i) =?= guess(hits,dec,char) 1.(2, 101, e) 2.(2, 115, s)
m[0x0x80002776] = want(s) =?= guess(hits,dec,char) 1.(1, 1,    ) 2.(1, 2, )
m[0x0x80002777] = want(I) =?= guess(hits,dec,char) 1.(2, 7,    ) 2.(2, 8,)
m[0x0x80002778] = want(s) =?= guess(hits,dec,char) 1.(2, 53, 5 ) 2.(2, 111, o)
m[0x0x80002779] = want(T) =?= guess(hits,dec,char) 1.(2, 6,    ) 2.(2, 69, E)
m[0x0x8000277a] = want(h) =?= guess(hits,dec,char) 1.(2, 2,    ) 2.(2, 10, )
m[0x0x8000277b] = want(e) =?= guess(hits,dec,char) 1.(3, 17,   ) 2.(3, 72, H)
m[0x0x8000277c] = want(B) =?= guess(hits,dec,char) 1.(2, 1,    ) 2.(2, 186, )
m[0x0x8000277d] = want(a) =?= guess(hits,dec,char) 1.(2, 35, # ) 2.(2, 45, -)
m[0x0x8000277e] = want(b) =?= guess(hits,dec,char) 1.(2, 13,   ) 2.(2, 69, E)
m[0x0x8000277f] = want(y) =?= guess(hits,dec,char) 1.(2, 2,    ) 2.(2, 9,  )
m[0x0x80002780] = want(B) =?= guess(hits,dec,char) 1.(2, 6,    ) 2.(2, 13,  )
m[0x0x80002781] = want(o) =?= guess(hits,dec,char) 1.(2, 3,    ) 2.(2, 16, )
m[0x0x80002782] = want(o) =?= guess(hits,dec,char) 1.(2, 151,  ) 2.(1, 1, )
m[0x0x80002783] = want(m) =?= guess(hits,dec,char) 1.(2, 125, }) 2.(1, 1, )
m[0x0x80002784] = want(e) =?= guess(hits,dec,char) 1.(3, 8,    ) 2.(3, 202, )
m[0x0x80002785] = want(r) =?= guess(hits,dec,char) 1.(3, 129,  ) 2.(2, 9,  )
m[0x0x80002786] = want(T) =?= guess(hits,dec,char) 1.(2, 2,    ) 2.(2, 177, )
m[0x0x80002787] = want(e) =?= guess(hits,dec,char) 1.(2, 54, 6 ) 2.(2, 254, )
m[0x0x80002788] = want(s) =?= guess(hits,dec,char) 1.(2, 2,    ) 2.(2, 34, ")
m[0x0x80002789] = want(t) =?= guess(hits,dec,char) 1.(2, 33, ! ) 2.(2, 58, :)

*** PASSED *** after 30780002 cycles
time elapsed: 1.5 s, simulation speed = 21.13 MHz
FPGA-Cycles-to-Model-Cycles Ratio (FMR): 1.42
Ran 30780002 cycles (fastest target clock)
[PASS] FireSim Test
SEED: 1625645658
```

Figure A.2: Spectre attack on the modified BOOM. Extra spaces have been inserted to improve readability. Special characters, such as header indicators, have been transformed into spaces. For more information, see the UTF-8 encoding standard.

```
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdfc8... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffffffffffdfc9... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffffdfca... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffffdfcb... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdfcc... Success: 0x4D='M' score=2
Reading at malicious_x = 0xffffffffffffdfcd... Success: 0x61='a' score=1
Reading at malicious_x = 0xffffffffffffdfce... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffffdfcf... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffffdfd0... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffffffffffdfd1... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdfd2... Success: 0x57='W' score=2
Reading at malicious_x = 0xffffffffffffdfd3... Success: 0x6F='o' score=2
Reading at malicious_x = 0xffffffffffffdfd4... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffffdfd5... Success: 0x64='d' score=2
Reading at malicious_x = 0xffffffffffffdfd6... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffffdfd7... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdfd8... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffffdfd9... Success: 0x72='r' score=1
Reading at malicious_x = 0xffffffffffffdfda... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffffdfdb... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdfdc... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffffffffffdfdd... Success: 0x71='q' score=2
Reading at malicious_x = 0xffffffffffffdfde... Success: 0x75='u' score=2
Reading at malicious_x = 0xffffffffffffdfdf... Success: 0x65='e' score=1
Reading at malicious_x = 0xffffffffffffdfe0... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffffdfe1... Success: 0x6D='m' score=2
Reading at malicious_x = 0xffffffffffffdfe2... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffffdfe3... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffffdfe4... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffffdfe5... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdfe6... Success: 0x4F='O' score=2
Reading at malicious_x = 0xffffffffffffdfe7... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffffdfe8... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffffdfe9... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffffdfea... Success: 0x66='f' score=2
Reading at malicious_x = 0xffffffffffffdfeb... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffffdfec... Success: 0x61='a' score=1
Reading at malicious_x = 0xffffffffffffdfed... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffffdfee... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffffdfef... Success: 0x2E='.' score=2
Exiting @ tick 575572251618 because exiting with last active thread context
```

Figure A.3: Spectre attack on the unmodified gem5 implementation.

```
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdfc8... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfc9... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfca... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfcb... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfcc... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfcd... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfce... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfcf... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd0... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd1... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd2... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd3... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd4... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd5... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd6... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd7... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd8... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfd9... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfda... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfdb... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfdc... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfdd... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfde... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfdf... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe0... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe1... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe2... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe3... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe4... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe5... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe6... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe7... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe8... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfe9... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfea... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfeb... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfec... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfed... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfee... Success: 0xFF='' score=0
Reading at malicious_x = 0xffffffffffffdfef... Success: 0xFF='' score=0
Exiting @ tick 1492493289672 because exiting with last active thread context
```

Figure A.4: Spectre attack on the modified gem5 implementation. As there is no UTF-8 character corresponding to 0xFF, all slots are left blank.

Appendix B

Hardware Size Analysis

In this appendix, we introduce the full results from the size analysis performed by Vivado for both the unmodified and modified BOOM, in that order. We highlight here, as elsewhere, that placing and routing algorithms used for this work are not deterministic, nor are they optimal. Results will vary somewhat between runs and this can have notable impact on both of properties such as size. Further investigation into size would be beneficial, especially using deterministic algorithms.

The results presented here are the results from invoking a cell utilization analysis from the cell starting with the boom and going down. The data is generated by Vivado, and includes multiple different properties of resource utilization. For our purposes, we look at the LUTs used, and how these vary between designs. Data is presented by instance name, which is what the component is named as a variable during synthesis, as well as a module name, which is the name given to the class defining the component.

Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

```
-------------------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2014.4 (lin64) Build 1071353 Tue Nov 18 16:47:07 MST 2014
| Date         : Tue Jul  6 11:14:50 2021
| Host         : WX2000Server running 64-bit CentOS release 6.10 (Final)
| Command      : report_utilization -cells [get_cells ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain/tile_reset_domain_boom_tile] -hierarchical -file base
| Design       : aemc_fpga
| Device       : xc7v2000t
| Design State : Routed
-------------------------------------------------------------------------------------------------------------------------
```

Utilization Design Information

Table of Contents

----

1. Utilization by Hierarchy

1. Utilization by Hierarchy

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
tile_reset_domain_boom_tile	BoomTile	229436	228266	1138	32	94695	40	109	36
tile_reset_domain_boom_tile	BoomTile	229436	228266	1138	32	94695	40	109	36
core	BoomCore	159751	159577	150	24	51420	0	3	36
(core)	BoomCore	1502	1502	0	0	847	0	0	0
alu_exe_unit	ALUExeUnit_3	8325	8322	0	3	469	0	0	16
alu	ALUUnit_2	4058	4058	0	0	247	0	0	0
imul	PipelinedMulUnit	4267	4264	0	3	222	0	0	16
(imul)	PipelinedMulUnit	4027	4024	0	3	63	0	0	0
imul	PipelinedMultiplier	240	240	0	0	159	0	0	16
csr	CSRFile	1761	1761	0	0	1017	0	0	0
csr_exe_unit	ALUExeUnit_2	12011	11953	58	0	511	0	0	0
alu	ALUUnit_1	5851	5851	0	0	164	0	0	0
ifpu	IntToFPUnit	1626	1626	0	0	175	0	0	0
(ifpu)	IntToFPUnit	39	39	0	0	37	0	0	0
ifpu	IntToFP	1594	1594	0	0	138	0	0	0
queue	BranchKillableQueue_5	4534	4476	58	0	172	0	0	0
dec_brmask_logic	BranchMaskGenerationLogic	158	158	0	0	16	0	0	0
fp_pipeline	FpPipeline	36722	36609	92	21	12089	0	0	20
(fp_pipeline)	FpPipeline	0	0	0	0	177	0	0	0
fp_issue_unit	IssueUnitCollapsing	4731	4731	0	0	2043	0	0	0
(fp_issue_unit)	IssueUnitCollapsing	0	0	0	0	3	0	0	0
slots_0	IssueSlot	108	108	0	0	85	0	0	0
slots_1	IssueSlot_15380	491	491	0	0	85	0	0	0
slots_10	IssueSlot_15381	12	12	0	0	85	0	0	0
slots_11	IssueSlot_15382	83	83	0	0	85	0	0	0
slots_12	IssueSlot_15383	14	14	0	0	85	0	0	0
slots_13	IssueSlot_15384	12	12	0	0	85	0	0	0
slots_14	IssueSlot_15385	11	11	0	0	85	0	0	0
slots_15	IssueSlot_15386	11	11	0	0	85	0	0	0
slots_16	IssueSlot_15387	29	29	0	0	85	0	0	0
slots_17	IssueSlot_15388	11	11	0	0	85	0	0	0
slots_18	IssueSlot_15389	11	11	0	0	85	0	0	0
slots_19	IssueSlot_15390	10	10	0	0	85	0	0	0
slots_2	IssueSlot_15391	197	197	0	0	85	0	0	0
slots_20	IssueSlot_15392	11	11	0	0	85	0	0	0
slots_21	IssueSlot_15393	11	11	0	0	85	0	0	0
slots_22	IssueSlot_15394	28	28	0	0	85	0	0	0
slots_23	IssueSlot_15395	52	52	0	0	85	0	0	0
slots_3	IssueSlot_15396	206	206	0	0	85	0	0	0
slots_4	IssueSlot_15397	3170	3170	0	0	85	0	0	0
slots_5	IssueSlot_15398	21	21	0	0	85	0	0	0
slots_6	IssueSlot_15399	11	11	0	0	85	0	0	0
slots_7	IssueSlot_15400	196	196	0	0	85	0	0	0
slots_8	IssueSlot_15401	15	15	0	0	85	0	0	0
slots_9	IssueSlot_15402	11	11	0	0	85	0	0	0
fpiu_unit	FPUExeUnit	18120	18007	92	21	3218	0	0	20
fdivsqrt	FDivSqrtUnit	5101	5101	0	0	1100	0	0	9
(fdivsqrt)	FDivSqrtUnit	3397	3397	0	0	310	0	0	0
divsqrt	DivSqrtRecF64	1677	1677	0	0	790	0	0	9
ds	DivSqrtRecF64_mulAddZ31	1446	1446	0	0	637	0	0	0
divSqrtRecF64ToRaw	DivSqrtRecF64ToRaw_mulAddZ31	1445	1445	0	0	637	0	0	0
roundRawFNToRecFN	RoundRawFNToRecFN_2	1	1	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_7	1	1	0	0	0	0	0	0
mul	Mul54	231	231	0	0	153	0	0	9
downvert_d2s	RecFNToRecFN_15378	27	27	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_4_15379	27	27	0	0	0	0	0	0
fp_sdq	BranchKillableQueue_7	239	193	46	0	146	0	0	0
fpu	FPUUnit	12207	12186	0	21	1636	0	0	11
(fpu)	FPUUnit	4092	4071	0	21	99	0	0	0
fpu	FPU	8115	8115	0	0	1537	0	0	11
(fpu)	FPU	0	0	0	0	205	0	0	0
dfma	FPUFMAPipe	5558	5558	0	0	547	0	0	9
(dfma)	FPUFMAPipe	3517	3517	0	0	271	0	0	0
fma	MulAddRecFNPipe	2041	2041	0	0	276	0	0	9
fpiu	FPToInt	1030	1030	0	0	142	0	0	0
(fpiu)	FPToInt	945	945	0	0	142	0	0	0
conv	RecFNToIN	85	85	0	0	0	0	0	0
fpmu	FPToFP	382	382	0	0	347	0	0	0
(fpmu)	FPToFP	342	342	0	0	347	0	0	0
narrower	RecFNToRecFN	40	40	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_4	40	40	0	0	0	0	0	0
sfma	FPUFMAPipe_1	1145	1145	0	0	296	0	0	2
(sfma)	FPUFMAPipe_1	262	262	0	0	143	0	0	0
fma	MulAddRecFNPipe_1	883	883	0	0	153	0	0	2
(fma)	MulAddRecFNPipe_1	616	616	0	0	153	0	0	2
mulAddRecFNToRaw_postMul	MulAddRecFNToRaw_postMul_1	263	263	0	0	0	0	0	0
roundRawFNToRecFN	RoundRawFNToRecFN_1	4	4	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_3	4	4	0	0	0	0	0	0
queue	BranchKillableQueue_6	573	527	46	0	336	0	0	0
fregfile	RegisterFileSynthesizable	13309	13309	0	0	6343	0	0	0
fregister_read	RegisterRead	562	562	0	0	308	0	0	0
fp_rename_stage	RenameStage_1	8112	8112	0	0	5691	0	0	0
(fp_rename_stage)	RenameStage_1	188	188	0	0	147	0	0	0
busytable	RenameBusyTable_1	317	317	0	0	96	0	0	0
freelist	RenameFreeList_1	5729	5729	0	0	1640	0	0	0
maptable	RenameMapTable_1	1878	1878	0	0	3808	0	0	0
ftq_arb	Arbiter_19	9	9	0	0	0	0	0	0
int_issue_unit	IssueUnitCollapsing_2	17145	17145	0	0	4515	0	0	0
(int_issue_unit)	IssueUnitCollapsing_2	1	1	0	0	3	0	0	0
slots_0	IssueSlot_24_15346	8107	8107	0	0	141	0	0	0
slots_1	IssueSlot_24_15347	565	565	0	0	141	0	0	0
slots_10	IssueSlot_24_15348	178	178	0	0	141	0	0	0

slots_11	IssueSlot_24_15349	395	395	0	0	141	0	0	0
slots_12	IssueSlot_24_15350	68	68	0	0	141	0	0	0
slots_13	IssueSlot_24_15351	627	627	0	0	141	0	0	0
slots_14	IssueSlot_24_15352	444	444	0	0	141	0	0	0
slots_15	IssueSlot_24_15353	67	67	0	0	141	0	0	0
slots_16	IssueSlot_24_15354	310	310	0	0	141	0	0	0
slots_17	IssueSlot_24_15355	265	265	0	0	141	0	0	0
slots_18	IssueSlot_24_15356	58	58	0	0	141	0	0	0
slots_19	IssueSlot_24_15357	153	153	0	0	141	0	0	0
slots_2	IssueSlot_24_15358	201	201	0	0	141	0	0	0
slots_20	IssueSlot_24_15359	394	394	0	0	141	0	0	0
slots_21	IssueSlot_24_15360	87	87	0	0	141	0	0	0
slots_22	IssueSlot_24_15361	485	485	0	0	141	0	0	0
slots_23	IssueSlot_24_15362	538	538	0	0	141	0	0	0
slots_24	IssueSlot_24_15363	77	77	0	0	141	0	0	0
slots_25	IssueSlot_24_15364	429	429	0	0	141	0	0	0
slots_26	IssueSlot_24_15365	295	295	0	0	141	0	0	0
slots_27	IssueSlot_24_15366	56	56	0	0	141	0	0	0
slots_28	IssueSlot_24_15367	355	355	0	0	141	0	0	0
slots_29	IssueSlot_24_15368	769	769	0	0	141	0	0	0
slots_3	IssueSlot_24_15369	74	74	0	0	141	0	0	0
slots_30	IssueSlot_24_15370	361	361	0	0	141	0	0	0
slots_31	IssueSlot_24_15371	245	245	0	0	141	0	0	0
slots_4	IssueSlot_24_15372	339	339	0	0	141	0	0	0
slots_5	IssueSlot_24_15373	366	366	0	0	141	0	0	0
slots_6	IssueSlot_24_15374	57	57	0	0	141	0	0	0
slots_7	IssueSlot_24_15375	463	463	0	0	141	0	0	0
slots_8	IssueSlot_24_15376	258	258	0	0	141	0	0	0
slots_9	IssueSlot_24_15377	67	67	0	0	141	0	0	0
iregfile	RegisterFileSynthesizable_1	18489	18489	0	0	6400	0	0	0
iregister_read	RegisterRead_1	6033	6033	0	0	1692	0	0	0
jmp_unit	ALUExeUnit_1	14896	14896	0	0	340	0	0	0
alu	ALUUnit	4834	4834	0	0	95	0	0	0
div	DivUnit	10062	10062	0	0	245	0	0	0
(div)	DivUnit	6	6	0	0	35	0	0	0
div	MulDiv	10056	10056	0	0	210	0	0	0
mem_issue_unit	IssueUnitCollapsing_1	4830	4830	0	0	1683	0	0	0
(mem_issue_unit)	IssueUnitCollapsing_1	0	0	0	0	3	0	0	0
slots_0	IssueSlot_24	240	240	0	0	105	0	0	0
slots_1	IssueSlot_24_15331	41	41	0	0	105	0	0	0
slots_10	IssueSlot_24_15332	135	135	0	0	105	0	0	0
slots_11	IssueSlot_24_15333	43	43	0	0	105	0	0	0
slots_12	IssueSlot_24_15334	51	51	0	0	105	0	0	0
slots_13	IssueSlot_24_15335	135	135	0	0	105	0	0	0
slots_14	IssueSlot_24_15336	49	49	0	0	105	0	0	0
slots_15	IssueSlot_24_15337	171	171	0	0	105	0	0	0
slots_2	IssueSlot_24_15338	337	337	0	0	105	0	0	0
slots_3	IssueSlot_24_15339	308	308	0	0	105	0	0	0
slots_4	IssueSlot_24_15340	343	343	0	0	105	0	0	0
slots_5	IssueSlot_24_15341	397	397	0	0	105	0	0	0
slots_6	IssueSlot_24_15342	2007	2007	0	0	105	0	0	0
slots_7	IssueSlot_24_15343	104	104	0	0	105	0	0	0
slots_8	IssueSlot_24_15344	107	107	0	0	105	0	0	0
slots_9	IssueSlot_24_15345	365	365	0	0	105	0	0	0
rename_stage	RenameStage	11798	11798	0	0	6023	0	0	0
(rename_stage)	RenameStage	392	392	0	0	526	0	0	0
busytable	RenameBusyTable	279	279	0	0	100	0	0	0
freelist	RenameFreeList	9471	9471	0	0	1708	0	0	0
maptable	RenameMapTable	1656	1656	0	0	3689	0	0	0
rob	Rob	17969	17969	0	0	10127	0	3	0
dcache	BoomNonBlockingDCache	6946	6727	218	1	4831	16	8	0
(dcache)	BoomNonBlockingDCache	312	312	0	0	905	0	0	0
data	BoomDuplicatedDataArray	926	926	0	0	0	16	0	0
lfsr_prng	MaxPeriodFibonacciLFSR_1_15324	4	3	0	1	17	0	0	0
meta_0	L1MetadataArray	86	86	0	0	7	0	8	0
mshrs	BoomMSHRFile	5168	4950	218	0	3275	0	0	0
(mshrs)	BoomMSHRFile	149	17	132	0	39	0	0	0
mmios_0	BoomIOMSHR	288	288	0	0	203	0	0	0
mshrs_0	BoomMSHR	1038	1028	10	0	727	0	0	0
(mshrs_0)	BoomMSHR	267	267	0	0	98	0	0	0
rpq	BranchKillableQueue_15330	771	761	10	0	629	0	0	0
mshrs_1	BoomMSHR_15325	1083	1073	10	0	727	0	0	0
(mshrs_1)	BoomMSHR_15325	320	320	0	0	98	0	0	0
rpq	BranchKillableQueue_15329	763	753	10	0	629	0	0	0
mshrs_2	BoomMSHR_15326	822	812	10	0	727	0	0	0
(mshrs_2)	BoomMSHR_15326	80	80	0	0	98	0	0	0
rpq	BranchKillableQueue_15328	742	732	10	0	629	0	0	0
mshrs_3	BoomMSHR_15327	1611	1601	10	0	727	0	0	0
(mshrs_3)	BoomMSHR_15327	383	383	0	0	98	0	0	0
rpq	BranchKillableQueue	1228	1218	10	0	629	0	0	0
respq	BranchKillableQueue_4	178	132	46	0	125	0	0	0
prober	BoomProbeUnit	70	70	0	0	56	0	0	0
wb	BoomWritebackUnit	381	381	0	0	571	0	0	0
frontend	BoomFrontend	41434	40657	770	7	27587	24	96	0
(frontend)	BoomFrontend	14	14	0	0	324	0	0	0
bpd	BranchPredictor	17512	16994	512	6	16306	2	88	0
(bpd)	BranchPredictor	2	2	0	0	66	0	0	0
banked_predictors_0	ComposedBranchPredictorBank	8905	8646	256	3	8098	1	44	0
components_0	LoopBranchPredictorBank_15308	3570	3567	0	3	2840	0	0	0
(components_0)	LoopBranchPredictorBank_15308	103	100	0	3	152	0	0	0
columns_0	LoopBranchPredictorColumn_15320	978	978	0	0	726	0	0	0
columns_1	LoopBranchPredictorColumn_15321	811	811	0	0	654	0	0	0
columns_2	LoopBranchPredictorColumn_15322	840	840	0	0	654	0	0	0
columns_3	LoopBranchPredictorColumn_15323	838	838	0	0	654	0	0	0
components_1	TageBranchPredictorBank_15309	1756	1500	256	0	1105	0	24	0
(components_1)	TageBranchPredictorBank_15309	156	156	0	0	365	0	0	0
alloc_lfsr_prng	MaxPeriodFibonacciLFSR_3_15313	31	31	0	0	6	0	0	0
tables_0	TageTable_15314	219	187	32	0	114	0	4	0
tables_1	TageTable_1_15315	204	172	32	0	118	0	4	0
tables_2	TageTable_2_15316	276	212	64	0	126	0	4	0
tables_3	TageTable_3_15317	317	253	64	0	126	0	4	0
tables_4	TageTable_4_15318	272	240	32	0	125	0	4	0
tables_5	TageTable_5_15319	282	250	32	0	125	0	4	0
components_2	BTBBranchPredictorBank_15310	1281	1281	0	0	365	1	16	0
components_3	FAMicroBTBBranchPredictorBank_15311	2270	2270	0	0	3717	0	0	0
components_4	BIMBranchPredictorBank_15312	57	57	0	0	71	0	4	0
banked_predictors_1	ComposedBranchPredictorBank_15304	8648	8389	256	3	8142	1	44	0
components_0	LoopBranchPredictorBank	3567	3564	0	3	2837	0	0	0
(components_0)	LoopBranchPredictorBank	100	97	0	3	149	0	0	0
columns_0	LoopBranchPredictorColumn	978	978	0	0	726	0	0	0
columns_1	LoopBranchPredictorColumn_15305	811	811	0	0	654	0	0	0
columns_2	LoopBranchPredictorColumn_15306	840	840	0	0	654	0	0	0
columns_3	LoopBranchPredictorColumn_15307	838	838	0	0	654	0	0	0
components_1	TageBranchPredictorBank	1757	1501	256	0	1104	0	24	0
(components_1)	TageBranchPredictorBank	154	154	0	0	364	0	0	0

```
|            alloc_lfsr_prng                |         MaxPeriodFibonacciLFSR_3 |         31 |         31 |       0 |    0 |     6 |      0 |      0 |            0 |
|            tables_0                       |                        TageTable |        219 |        187 |      32 |    0 |   114 |      0 |      4 |            0 |
|            tables_1                       |                      TageTable_1 |        204 |        172 |      32 |    0 |   118 |      0 |      4 |            0 |
|            tables_2                       |                      TageTable_2 |        277 |        213 |      64 |    0 |   126 |      0 |      4 |            0 |
|            tables_3                       |                      TageTable_3 |        318 |        254 |      64 |    0 |   126 |      0 |      4 |            0 |
|            tables_4                       |                      TageTable_4 |        272 |        240 |      32 |    0 |   125 |      0 |      4 |            0 |
|            tables_5                       |                      TageTable_5 |        284 |        252 |      32 |    0 |   125 |      0 |      4 |            0 |
|          components_2                     |          BTBBranchPredictorBank |       1160 |       1160 |       0 |    0 |   411 |      1 |     16 |            0 |
|          components_3                     |   FAMicroBTBBranchPredictorBank |       2178 |       2178 |       0 |    0 |  3717 |      0 |      0 |            0 |
|          components_4                     |          BIMBranchPredictorBank |         51 |         51 |       0 |    0 |    73 |      0 |      4 |            0 |
|      f3                                   |                         Queue_30 |       3064 |       3064 |       0 |    0 |   247 |      0 |      0 |            0 |
|      f3_bpd_resp                          |                         Queue_31 |        432 |        432 |       0 |    0 |   568 |      0 |      0 |            0 |
|      f4                                   |                         Queue_33 |       3855 |       3855 |       0 |    0 |   814 |      0 |      0 |            0 |
|      f4_btb_corrections                   |                         Queue_32 |        293 |         53 |     240 |    0 |     3 |      0 |      0 |            0 |
|      fb                                   |                      FetchBuffer |       5875 |       5875 |       0 |    0 |  1915 |      0 |      0 |            0 |
|      ftq                                  |                 FetchTargetQueue |       7106 |       7088 |      18 |    0 |  3998 |      6 |      0 |            0 |
|      icache                               |                           ICache |       1814 |       1813 |       0 |    1 |   589 |     16 |      8 |            0 |
|        (icache)                           |                           ICache |       1232 |       1232 |       0 |    0 |   572 |     16 |      8 |            0 |
|        repl_way_prng                      |         MaxPeriodFibonacciLFSR_1 |        582 |        581 |       0 |    1 |    17 |      0 |      0 |            0 |
|      ras                                  |                          BoomRAS |        383 |        383 |       0 |    0 |  1361 |      0 |      0 |            0 |
|      tlb                                  |                              TLB |       1090 |       1090 |       0 |    0 |  1462 |      0 |      0 |            0 |
|    lsu                                    |                              LSU |      19014 |      19014 |       0 |    0 |  9196 |      0 |      0 |            0 |
|      (lsu)                                |                              LSU |      17511 |      17511 |       0 |    0 |  8198 |      0 |      0 |            0 |
|      dtlb                                 |                           NBDTLB |       1504 |       1504 |       0 |    0 |   998 |      0 |      0 |            0 |
|    ptw                                    |                              PTW |       2422 |       2422 |       0 |    0 |  1649 |      0 |      2 |            0 |
|    tlMasterXbar                           |                        TLXbar_7 |         19 |         19 |       0 |    0 |    12 |      0 |      0 |            0 |
+-------------------------------------------+----------------------------------+------------+------------+---------+------+-------+--------+--------+--------------+
* Note: The sum of lower-level cells may be larger than their parent cells total, due to cross-hierarchy LUT combining
```

Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

```
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2014.4 (lin64) Build 1071353 Tue Nov 18 16:47:07 MST 2014
| Date         : Wed Jul  7 09:23:15 2021
| Host         : WX2000Server running 64-bit CentOS release 6.10 (Final)
| Command      : report_utilization -cells [get_cells ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain/tile_reset_domain_boom_tile] -hierarchical -file modified_size.rp
| Design       : aemc_fpga
| Device       : xc7v2000t
| Design State : Routed
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Utilization Design Information

Table of Contents
----
1. Utilization by Hierarchy

1. Utilization by Hierarchy

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
tile_reset_domain_boom_tile	BoomTile	232350	231180	1138	32	95891	40	109	36
tile_reset_domain_boom_tile	BoomTile	232350	231180	1138	32	95891	40	109	36
core	BoomCore	165568	165394	150	24	52621	0	3	36
(core)	BoomCore	1415	1415	0	0	855	0	0	0
alu_exe_unit_3	ALUExeUnit_3	7708	7705	0	3	471	0	0	16
alu	ALUUnit_2	3701	3701	0	0	215	0	0	0
imul	PipelinedMulUnit	4007	4004	0	3	256	0	0	16
(imul)	PipelinedMulUnit	3768	3765	0	3	97	0	0	0
imul	PipelinedMultiplier	239	239	0	0	159	0	0	16
csr	CSRFile	1846	1846	0	0	1017	0	0	0
csr_exe_unit	ALUExeUnit_2	12372	12314	58	0	516	0	0	0
alu	ALUUnit_1	6150	6150	0	0	169	0	0	0
ifpu	IntToFPUnit	1627	1627	0	0	175	0	0	0
(ifpu)	IntToFPUnit	38	38	0	0	37	0	0	0
ifpu	IntToFP	1593	1593	0	0	138	0	0	0
queue	BranchKillableQueue_5	4595	4537	58	0	172	0	0	0
dec_brmask_logic	BranchMaskGenerationLogic	190	190	0	0	16	0	0	0
dispatcher	BasicDispatcher	3	3	0	0	0	0	0	0
fp_pipeline	FpPipeline	37094	36981	92	21	12089	0	0	20
(fp_pipeline)	FpPipeline	0	0	0	0	177	0	0	0
fp_issue_unit	IssueUnitCollapsing	4723	4723	0	0	2043	0	0	0
(fp_issue_unit)	IssueUnitCollapsing	0	0	0	0	3	0	0	0
slots_0	IssueSlot	110	110	0	0	85	0	0	0
slots_1	IssueSlot_15380	491	491	0	0	85	0	0	0
slots_10	IssueSlot_15381	12	12	0	0	85	0	0	0
slots_11	IssueSlot_15382	83	83	0	0	85	0	0	0
slots_12	IssueSlot_15383	14	14	0	0	85	0	0	0
slots_13	IssueSlot_15384	12	12	0	0	85	0	0	0
slots_14	IssueSlot_15385	11	11	0	0	85	0	0	0
slots_15	IssueSlot_15386	11	11	0	0	85	0	0	0
slots_16	IssueSlot_15387	29	29	0	0	85	0	0	0
slots_17	IssueSlot_15388	11	11	0	0	85	0	0	0
slots_18	IssueSlot_15389	11	11	0	0	85	0	0	0
slots_19	IssueSlot_15390	10	10	0	0	85	0	0	0
slots_2	IssueSlot_15391	197	197	0	0	85	0	0	0
slots_20	IssueSlot_15392	11	11	0	0	85	0	0	0
slots_21	IssueSlot_15393	11	11	0	0	85	0	0	0
slots_22	IssueSlot_15394	28	28	0	0	85	0	0	0
slots_23	IssueSlot_15395	52	52	0	0	85	0	0	0
slots_3	IssueSlot_15396	206	206	0	0	85	0	0	0
slots_4	IssueSlot_15397	3159	3159	0	0	85	0	0	0
slots_5	IssueSlot_15398	21	21	0	0	85	0	0	0
slots_6	IssueSlot_15399	11	11	0	0	85	0	0	0
slots_7	IssueSlot_15400	196	196	0	0	85	0	0	0
slots_8	IssueSlot_15401	15	15	0	0	85	0	0	0
slots_9	IssueSlot_15402	11	11	0	0	85	0	0	0
fpiu_unit	FPUExeUnit	18501	18388	92	21	3218	0	0	20
fdivsqrt	FDivSqrtUnit	5095	5095	0	0	1100	0	0	9
(fdivsqrt)	FDivSqrtUnit	3398	3398	0	0	310	0	0	0
divsqrt	DivSqrtRecF64	1670	1670	0	0	790	0	0	9
ds	DivSqrtRecF64_mulAddZ31	1439	1439	0	0	637	0	0	0
divSqrtRecF64ToRaw	DivSqrtRecF64ToRaw_mulAddZ31	1438	1438	0	0	637	0	0	0
roundRawFNToRecFN	RoundRawFNToRecFN_2	1	1	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_7	1	1	0	0	0	0	0	0
mul	Mul54	231	231	0	0	153	0	0	9
downvert_d2s	RecFNToRecFN_15378	27	27	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_4_15379	27	27	0	0	0	0	0	0
fp_sdq	BranchKillableQueue_7	240	194	46	0	146	0	0	0
fpu	FPUUnit	12250	12229	0	21	1636	0	0	11
(fpu)	FPUUnit	4139	4118	0	21	99	0	0	0
fpu	FPU	8111	8111	0	0	1537	0	0	11
(fpu)	FPU	0	0	0	0	205	0	0	0
dfma	FPUFMAPipe	5556	5556	0	0	547	0	0	9
(dfma)	FPUFMAPipe	3517	3517	0	0	271	0	0	0
fma	MulAddRecFNPipe	2039	2039	0	0	276	0	0	9
fpiu	FPToInt	1030	1030	0	0	142	0	0	0
(fpiu)	FPToInt	945	945	0	0	142	0	0	0
conv	RecFNToIN	85	85	0	0	0	0	0	0
fpmu	FPToFP	381	381	0	0	347	0	0	0
(fpmu)	FPToFP	341	341	0	0	347	0	0	0
narrower	RecFNToRecFN	40	40	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_4	40	40	0	0	0	0	0	0
sfma	FPUFMAPipe_1	1144	1144	0	0	296	0	0	2
(sfma)	FPUFMAPipe_1	263	263	0	0	143	0	0	0
fma	MulAddRecFNPipe_1	883	883	0	0	153	0	0	2
(fma)	MulAddRecFNPipe_1	616	616	0	0	153	0	0	2
mulAddRecFNToRaw_postMul	MulAddRecFNToRaw_postMul_1	263	263	0	0	0	0	0	0
roundRawFNToRecFN	RoundRawFNToRecFN_1	4	4	0	0	0	0	0	0
roundAnyRawFNToRecFN	RoundAnyRawFNToRecFN_3	4	4	0	0	0	0	0	0
queue	BranchKillableQueue_6	916	870	46	0	336	0	0	0
fregfile	RegisterFileSynthesizable	13309	13309	0	0	6343	0	0	0
fregister_read	RegisterRead	562	562	0	0	308	0	0	0
fp_rename_stage	RenameStage_1	8086	8086	0	0	5689	0	0	0
(fp_rename_stage)	RenameStage_1	187	187	0	0	145	0	0	0
busytable	RenameBusyTable_1	316	316	0	0	96	0	0	0
freelist	RenameFreeList_1	5705	5705	0	0	1640	0	0	0
maptable	RenameMapTable_1	1878	1878	0	0	3808	0	0	0
int_issue_unit	IssueUnitCollapsing_2	17451	17451	0	0	4515	0	0	0
(int_issue_unit)	IssueUnitCollapsing_2	0	0	0	0	3	0	0	0
slots_0	IssueSlot_24_15346	188	188	0	0	141	0	0	0
slots_1	IssueSlot_24_15347	214	214	0	0	141	0	0	0
slots_10	IssueSlot_24_15348	215	215	0	0	141	0	0	0

```
|       slots_11                  |                  IssueSlot_24_15349 |   356 |   356 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_12                  |                  IssueSlot_24_15350 |   169 |   169 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_13                  |                  IssueSlot_24_15351 |   542 |   542 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_14                  |                  IssueSlot_24_15352 |   207 |   207 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_15                  |                  IssueSlot_24_15353 |   126 |   126 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_16                  |                  IssueSlot_24_15354 |   435 |   435 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_17                  |                  IssueSlot_24_15355 |   180 |   180 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_18                  |                  IssueSlot_24_15356 |    83 |    83 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_19                  |                  IssueSlot_24_15357 |   271 |   271 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_2                   |                  IssueSlot_24_15358 |  8774 |  8774 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_20                  |                  IssueSlot_24_15359 |   262 |   262 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_21                  |                  IssueSlot_24_15360 |   804 |   804 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_22                  |                  IssueSlot_24_15361 |   260 |   260 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_23                  |                  IssueSlot_24_15362 |   151 |   151 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_24                  |                  IssueSlot_24_15363 |   114 |   114 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_25                  |                  IssueSlot_24_15364 |   295 |   295 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_26                  |                  IssueSlot_24_15365 |   746 |   746 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_27                  |                  IssueSlot_24_15366 |    57 |    57 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_28                  |                  IssueSlot_24_15367 |   190 |   190 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_29                  |                  IssueSlot_24_15368 |  1076 |  1076 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_3                   |                  IssueSlot_24_15369 |   434 |   434 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_30                  |                  IssueSlot_24_15370 |    60 |    60 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_31                  |                  IssueSlot_24_15371 |    91 |    91 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_4                   |                  IssueSlot_24_15372 |   308 |   308 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_5                   |                  IssueSlot_24_15373 |    56 |    56 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_6                   |                  IssueSlot_24_15374 |    75 |    75 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_7                   |                  IssueSlot_24_15375 |   463 |   463 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_8                   |                  IssueSlot_24_15376 |   143 |   143 |   0 |  0 |   141 |  0 |  0 |   0 |
|       slots_9                   |                  IssueSlot_24_15377 |   123 |   123 |   0 |  0 |   141 |  0 |  0 |   0 |
|     iregfile                    |         RegisterFileSynthesizable_1 | 18813 | 18813 |   0 |  0 |  6400 |  0 |  0 |   0 |
|     iregister_read              |                      RegisterRead_1 |  6334 |  6334 |   0 |  0 |  1692 |  0 |  0 |   0 |
|     jmp_unit                    |                      ALUExeUnit_1 | 13959 | 13959 |   0 |  0 |   341 |  0 |  0 |   0 |
|       alu                       |                             ALUUnit |  4325 |  4325 |   0 |  0 |    95 |  0 |  0 |   0 |
|       div                       |                             DivUnit |  9634 |  9634 |   0 |  0 |   246 |  0 |  0 |   0 |
|         (div)                   |                             DivUnit |     0 |     0 |   0 |  0 |    35 |  0 |  0 |   0 |
|         div                     |                              MulDiv |  9634 |  9634 |   0 |  0 |   211 |  0 |  0 |   0 |
|     mem_issue_unit              |               IssueUnitCollapsing_1 |  4627 |  4627 |   0 |  0 |  1683 |  0 |  0 |   0 |
|       (mem_issue_unit)          |               IssueUnitCollapsing_1 |     0 |     0 |   0 |  0 |     3 |  0 |  0 |   0 |
|       slots_0                   |                        IssueSlot_24 |   111 |   111 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_1                   |                  IssueSlot_24_15331 |  2137 |  2137 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_10                  |                  IssueSlot_24_15332 |   150 |   150 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_11                  |                  IssueSlot_24_15333 |    46 |    46 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_12                  |                  IssueSlot_24_15334 |   300 |   300 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_13                  |                  IssueSlot_24_15335 |    58 |    58 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_14                  |                  IssueSlot_24_15336 |    43 |    43 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_15                  |                  IssueSlot_24_15337 |    62 |    62 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_2                   |                  IssueSlot_24_15338 |   339 |   339 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_3                   |                  IssueSlot_24_15339 |   479 |   479 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_4                   |                  IssueSlot_24_15340 |   285 |   285 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_5                   |                  IssueSlot_24_15341 |    63 |    63 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_6                   |                  IssueSlot_24_15342 |   294 |   294 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_7                   |                  IssueSlot_24_15343 |    98 |    98 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_8                   |                  IssueSlot_24_15344 |   108 |   108 |   0 |  0 |   105 |  0 |  0 |   0 |
|       slots_9                   |                  IssueSlot_24_15345 |    55 |    55 |   0 |  0 |   105 |  0 |  0 |   0 |
|     rename_stage                |                         RenameStage | 13614 | 13614 |   0 |  0 |  6025 |  0 |  0 |   0 |
|       (rename_stage)            |                         RenameStage |  5177 |  5177 |   0 |  0 |   528 |  0 |  0 |   0 |
|       busytable                 |                     RenameBusyTable |   282 |   282 |   0 |  0 |   100 |  0 |  0 |   0 |
|       freelist                  |                      RenameFreeList |  6414 |  6414 |   0 |  0 |  1708 |  0 |  0 |   0 |
|       maptable                  |                      RenameMapTable |  1741 |  1741 |   0 |  0 |  3689 |  0 |  0 |   0 |
|     rob                         |                                 Rob | 19672 | 19672 |   0 |  0 | 10705 |  0 |  3 |   0 |
|     rq                          |                        ReleaseQueue |  1811 |  1811 |   0 |  0 |   416 |  0 |  0 |   0 |
|     sb                          |                        ShadowBuffer |   594 |   594 |   0 |  0 |   191 |  0 |  0 |   0 |
|   dcache                        |                BoomNonBlockingDCache |  7757 |  7538 | 218 |  1 |  4833 | 16 |  8 |   0 |
|     (dcache)                    |                BoomNonBlockingDCache |   311 |   311 |   0 |  0 |   907 |  0 |  0 |   0 |
|     data                        |             BoomDuplicatedDataArray |   925 |   925 |   0 |  0 |     0 | 16 |  0 |   0 |
|     lfsr_prng                   |       MaxPeriodFibonacciLFSR_1_15324 |     4 |     3 |   0 |  1 |    17 |  0 |  0 |   0 |
|     meta_0                      |                    L1MetadataArray |    86 |    86 |   0 |  0 |     7 |  0 |  8 |   0 |
|     mshrs                       |                        BoomMSHRFile |  5977 |  5759 | 218 |  0 |  3275 |  0 |  0 |   0 |
|       (mshrs)                   |                        BoomMSHRFile |   149 |    17 | 132 |  0 |    39 |  0 |  0 |   0 |
|       mmios_0                   |                         BoomIOMSHR |   288 |   288 |   0 |  0 |   203 |  0 |  0 |   0 |
|       mshrs_0                   |                            BoomMSHR |  2106 |  2096 |  10 |  0 |   727 |  0 |  0 |   0 |
|         (mshrs_0)               |                            BoomMSHR |   292 |   292 |   0 |  0 |    98 |  0 |  0 |   0 |
|         rpq                     |         BranchKillableQueue_15330 |  1814 |  1804 |  10 |  0 |   629 |  0 |  0 |   0 |
|       mshrs_1                   |                      BoomMSHR_15325 |  1101 |  1091 |  10 |  0 |   727 |  0 |  0 |   0 |
|         (mshrs_1)               |                      BoomMSHR_15325 |   325 |   325 |   0 |  0 |    98 |  0 |  0 |   0 |
|         rpq                     |         BranchKillableQueue_15329 |   776 |   766 |  10 |  0 |   629 |  0 |  0 |   0 |
|       mshrs_2                   |                      BoomMSHR_15326 |   816 |   806 |  10 |  0 |   727 |  0 |  0 |   0 |
|         (mshrs_2)               |                      BoomMSHR_15326 |    86 |    86 |   0 |  0 |    98 |  0 |  0 |   0 |
|         rpq                     |         BranchKillableQueue_15328 |   730 |   720 |  10 |  0 |   629 |  0 |  0 |   0 |
|       mshrs_3                   |                      BoomMSHR_15327 |  1354 |  1344 |  10 |  0 |   727 |  0 |  0 |   0 |
|         (mshrs_3)               |                      BoomMSHR_15327 |   315 |   315 |   0 |  0 |    98 |  0 |  0 |   0 |
|         rpq                     |                 BranchKillableQueue |  1039 |  1029 |  10 |  0 |   629 |  0 |  0 |   0 |
|       respq                     |               BranchKillableQueue_4 |   167 |   121 |  46 |  0 |   125 |  0 |  0 |   0 |
|     prober                      |                       BoomProbeUnit |    73 |    73 |   0 |  0 |    56 |  0 |  0 |   0 |
|     wb                          |                   BoomWritebackUnit |   381 |   381 |   0 |  0 |   571 |  0 |  0 |   0 |
|   frontend                      |                        BoomFrontend | 40832 | 40055 | 770 |  7 | 27587 | 24 | 96 |   0 |
|     (frontend)                  |                        BoomFrontend |    14 |    14 |   0 |  0 |   324 |  0 |  0 |   0 |
|     bpd                         |                     BranchPredictor | 17494 | 16976 | 512 |  6 | 16306 |  2 | 88 |   0 |
|       (bpd)                     |                     BranchPredictor |     1 |     1 |   0 |  0 |    66 |  0 |  0 |   0 |
|       banked_predictors_0       |       ComposedBranchPredictorBank |  8909 |  8650 | 256 |  3 |  8098 |  1 | 44 |   0 |
|         components_0            |       LoopBranchPredictorBank_15308 |  3570 |  3567 |   0 |  3 |  2840 |  0 |  0 |   0 |
|           (components_0)        |       LoopBranchPredictorBank_15308 |   102 |    99 |   0 |  3 |   152 |  0 |  0 |   0 |
|           columns_0             |     LoopBranchPredictorColumn_15320 |   979 |   979 |   0 |  0 |   726 |  0 |  0 |   0 |
|           columns_1             |     LoopBranchPredictorColumn_15321 |   811 |   811 |   0 |  0 |   654 |  0 |  0 |   0 |
|           columns_2             |     LoopBranchPredictorColumn_15322 |   840 |   840 |   0 |  0 |   654 |  0 |  0 |   0 |
|           columns_3             |     LoopBranchPredictorColumn_15323 |   838 |   838 |   0 |  0 |   654 |  0 |  0 |   0 |
|         components_1            |       TageBranchPredictorBank_15309 |  1755 |  1499 | 256 |  0 |  1105 |  0 | 24 |   0 |
|           (components_1)        |       TageBranchPredictorBank_15309 |   156 |   156 |   0 |  0 |   365 |  0 |  0 |   0 |
|           alloc_lfsr_prng       |       MaxPeriodFibonacciLFSR_3_15313 |    31 |    31 |   0 |  0 |     6 |  0 |  0 |   0 |
|           tables_0              |                    TageTable_15314 |   218 |   186 |  32 |  0 |   114 |  0 |  4 |   0 |
|           tables_1              |                  TageTable_1_15315 |   204 |   172 |  32 |  0 |   118 |  0 |  4 |   0 |
|           tables_2              |                  TageTable_2_15316 |   277 |   213 |  64 |  0 |   126 |  0 |  4 |   0 |
|           tables_3              |                  TageTable_3_15317 |   318 |   254 |  64 |  0 |   126 |  0 |  4 |   0 |
|           tables_4              |                  TageTable_4_15318 |   272 |   240 |  32 |  0 |   125 |  0 |  4 |   0 |
|           tables_5              |                  TageTable_5_15319 |   281 |   249 |  32 |  0 |   125 |  0 |  4 |   0 |
|         components_2            |        BTBBranchPredictorBank_15310 |  1284 |  1284 |   0 |  0 |   365 |  1 | 16 |   0 |
|         components_3            | FAMicroBTBBranchPredictorBank_15311 |  2270 |  2270 |   0 |  0 |  3717 |  0 |  0 |   0 |
|         components_4            |        BIMBranchPredictorBank_15312 |    57 |    57 |   0 |  0 |    71 |  0 |  4 |   0 |
|       banked_predictors_1       |  ComposedBranchPredictorBank_15304 |  8630 |  8371 | 256 |  3 |  8142 |  1 | 44 |   0 |
|         components_0            |             LoopBranchPredictorBank |  3565 |  3562 |   0 |  3 |  2837 |  0 |  0 |   0 |
|           (components_0)        |             LoopBranchPredictorBank |    99 |    96 |   0 |  3 |   149 |  0 |  0 |   0 |
|           columns_0             |           LoopBranchPredictorColumn |   977 |   977 |   0 |  0 |   726 |  0 |  0 |   0 |
|           columns_1             |     LoopBranchPredictorColumn_15305 |   811 |   811 |   0 |  0 |   654 |  0 |  0 |   0 |
|           columns_2             |     LoopBranchPredictorColumn_15306 |   840 |   840 |   0 |  0 |   654 |  0 |  0 |   0 |
|           columns_3             |     LoopBranchPredictorColumn_15307 |   838 |   838 |   0 |  0 |   654 |  0 |  0 |   0 |
```

```
|          components_1                     |                  TageBranchPredictorBank |       1749 |       1493 |     256 |    0 |  1104 |      0 |     24 |            0 |
|            (components_1)                 |                  TageBranchPredictorBank |        154 |        154 |       0 |    0 |   364 |      0 |      0 |            0 |
|            alloc_lfsr_prng                |                MaxPeriodFibonacciLFSR_3 |         31 |         31 |       0 |    0 |     6 |      0 |      0 |            0 |
|            tables_0                       |                                TageTable |        217 |        185 |      32 |    0 |   114 |      0 |      4 |            0 |
|            tables_1                       |                              TageTable_1 |        202 |        170 |      32 |    0 |   118 |      0 |      4 |            0 |
|            tables_2                       |                              TageTable_2 |        277 |        213 |      64 |    0 |   126 |      0 |      4 |            0 |
|            tables_3                       |                              TageTable_3 |        318 |        254 |      64 |    0 |   126 |      0 |      4 |            0 |
|            tables_4                       |                              TageTable_4 |        271 |        239 |      32 |    0 |   125 |      0 |      4 |            0 |
|            tables_5                       |                              TageTable_5 |        281 |        249 |      32 |    0 |   125 |      0 |      4 |            0 |
|          components_2                     |                   BTBBranchPredictorBank |       1154 |       1154 |       0 |    0 |   411 |      1 |     16 |            0 |
|          components_3                     |            FAMicroBTBBranchPredictorBank |       2178 |       2178 |       0 |    0 |  3717 |      0 |      0 |            0 |
|          components_4                     |                   BIMBranchPredictorBank |         50 |         50 |       0 |    0 |    73 |      0 |      4 |            0 |
|      f3                                   |                                 Queue_30 |       3062 |       3062 |       0 |    0 |   247 |      0 |      0 |            0 |
|      f3_bpd_resp                          |                                 Queue_31 |        432 |        432 |       0 |    0 |   568 |      0 |      0 |            0 |
|      f4                                   |                                 Queue_33 |       4774 |       4774 |       0 |    0 |   814 |      0 |      0 |            0 |
|      f4_btb_corrections                   |                                 Queue_32 |        293 |         53 |     240 |    0 |     3 |      0 |      0 |            0 |
|      fb                                   |                              FetchBuffer |       4364 |       4364 |       0 |    0 |  1916 |      0 |      0 |            0 |
|      ftq                                  |                         FetchTargetQueue |       7116 |       7098 |      18 |    0 |  3997 |      6 |      0 |            0 |
|      icache                               |                                   ICache |       1814 |       1813 |       0 |    1 |   589 |     16 |      8 |            0 |
|        (icache)                           |                                   ICache |       1232 |       1232 |       0 |    0 |   572 |     16 |      8 |            0 |
|        repl_way_prng                      |                MaxPeriodFibonacciLFSR_1 |        582 |        581 |       0 |    1 |    17 |      0 |      0 |            0 |
|      ras                                  |                                  BoomRAS |        383 |        383 |       0 |    0 |  1361 |      0 |      0 |            0 |
|      tlb                                  |                                      TLB |       1088 |       1088 |       0 |    0 |  1462 |      0 |      0 |            0 |
|    lsu                                    |                                      LSU |      15870 |      15870 |       0 |    0 |  9189 |      0 |      0 |            0 |
|      (lsu)                                |                                      LSU |      14433 |      14433 |       0 |    0 |  8191 |      0 |      0 |            0 |
|      dtlb                                 |                                   NBDTLB |       1437 |       1437 |       0 |    0 |   998 |      0 |      0 |            0 |
|    ptw                                    |                                      PTW |       2457 |       2457 |       0 |    0 |  1649 |      0 |      2 |            0 |
|    tlMasterXbar                           |                                 TLXbar_7 |         19 |         19 |       0 |    0 |    12 |      0 |      0 |            0 |
+-------------------------------------------+------------------------------------------+------------+------------+---------+------+-------+--------+--------+--------------+
* Note: The sum of lower-level cells may be larger than their parent cells total, due to cross-hierarchy LUT combining
```

Appendix C

Timing Analysis

In this section, we present the first timing result from the Vivado timing analysis. For this analysis, we set the parameters so that Vivado analyzed the 100 worst paths timing wire, that passed through the BOOM. This gives the definitive worst results of timing for that particular place and route, but does not indicate what results an optimal place and route might give. The timing results for both the unmodified and the modified BOOM are presented, in that order.

The timing here only concerns itself with wires passing through the BOOM tile, as those were the only ones affected by our design modifications. It shows the entire path from the driver of the clock, all the way through to the clock arrival position. At the end of this, a summary of the timing results for that clock path is given, and it is confirmed whether timing requirements are met. The summary at the top also shows this, with "Slack (MET)". Other timing properties are also available.

As mentioned, the timing results are not conclusive. The place and route algorithm used by Vivado is not deterministic, and can therefore give different results depending on the randomness of an attempted place and route. Future research should aim to use a more deterministic approach, and also give the program more iterations to discover a more performant place and route solution.

Note that a linebreak has been inserted in the full path of the timing results, after "tile_prci_domain", in order to prevent the path from overfilling the page.

```
Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Tool Version      : Vivado v.2014.4 (lin64) Build 1071353 Tue Nov 18 16:47:07 MST 2014
| Date              : Tue Jul  6 11:13:43 2021
| Host              : WX2000Server running 64-bit CentOS release 6.10 (Final)
| Command           : report_timing -max_paths 100 -through [get_cells ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile] -name boom -file base_timing.rpt
| Design            : aemc_fpga
| Device            : 7v2000t-fhg1761
| Speed File        : -2  PRODUCTION 1.10 2014-09-11
| Temperature Grade : C
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Timing Report

Slack (MET) :              6.299ns  (required time - arrival time)
  Source:                  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/REG_9_reg/C
                             (rising edge-triggered cell FDRE clocked by clk_per_i  {rise@0.000ns fall@16.667ns period=33.333ns})
  Destination:             ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_4_reg/ENARDEN
                             (rising edge-triggered cell RAMB18E1 clocked by clk_per_i  {rise@0.000ns fall@16.667ns period=33.333ns})
  Path Group:              clk_per_i
  Path Type:               Setup (Max at Slow Process Corner)
  Requirement:             33.333ns  (clk_per_i rise@33.333ns - clk_per_i rise@0.000ns)
  Data Path Delay:         26.718ns  (logic 2.410ns (9.020%)  route 24.308ns (90.980%))
  Logic Levels:            31  (CARRY4=5 LUT2=1 LUT3=3 LUT5=5 LUT6=15 MUXF7=2)
  Clock Path Skew:         0.277ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    10.739ns = ( 44.072 - 33.333 )
    Source Clock Delay      (SCD):    11.333ns
    Clock Pessimism Removal (CPR):    0.871ns
  Clock Uncertainty:       0.266ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.526ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock clk_per_i rise edge)
                                                      0.000     0.000 r
    AB33                                              0.000     0.000 r  c0_sys_clk_p
                         net (fo=0)                   0.000     0.000    io/c0_sys_clk_p
    AB33                 IBUFDS (Prop_ibufds_I_O)     0.877     0.877 r  io/ckbuf/O
                         net (fo=1, routed)           1.712     2.589    io/sys_ref_clk_i
    BUFGCTRL_X0Y48       BUFG (Prop_bufg_I_O)         0.093     2.682 r  io/ckbufg/O
                         net (fo=6, routed)           1.552     4.234    io/n_2_ckbufg
    PLLE2_ADV_X0Y4       PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT1)
                                                      0.077     4.311 r  io/pll_idly/CLKOUT1
                         net (fo=1, routed)           1.541     5.852    io/clk_i
    BUFGCTRL_X0Y32       BUFG (Prop_bufg_I_O)         0.093     5.945 r  io/clkbufg/O
                         net (fo=85441, routed)       2.071     8.016    io/cae0_clock/I1
    SLR Crossing[1->0]
    MMCME2_ADV_X0Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                      0.077     8.093 r  io/cae0_clock/gasync.gbufg.gmmcm.pll/CLKOUT1
                         net (fo=1, routed)           1.543     9.636    io/cae0_clock/clk_per_i
    BUFGCTRL_X0Y3        BUFG (Prop_bufg_I_O)         0.093     9.729 r  io/cae0_clock/gasync.gbufg.per_bufg/O
                         net (fo=12751, routed)       0.132     9.861    ae0/cae0.cae_pers/top/top/sim/target/FireSim/out[0]
    BUFGCTRL_X0Y2        BUFGCTRL (Prop_bufgctrl_I0_O)
                                                      0.093     9.954 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/RationalClockBridge_clocks_0_buffer/O
                         net (fo=110524, routed)      1.379    11.333    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/lazyModule_clocks_subsystem_cbus_0
    SLICE_X124Y58                                                       r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/REG_9_reg/C
  -------------------------------------------------------------------    -------------------
    SLICE_X124Y58        FDRE (Prop_fdre_C_Q)         0.223    11.556 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/REG_9_reg/Q
                         net (fo=43, routed)          0.981    12.537    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/REG_9
    SLICE_X104Y57        LUT3 (Prop_lut3_I2_O)        0.043    12.580 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_2/ram_meta_1[54]_i_8/O
                         net (fo=4, routed)           1.302    13.882    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_0/components_2_io_resp_f3_1_taken
    SLICE_X40Y56         LUT5 (Prop_lut5_I3_O)        0.043    13.925 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_0/ram_meta_1[54]_i_26/O
                         net (fo=1, routed)           0.364    14.290    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_1/I30
    SLICE_X39Y56         LUT6 (Prop_lut6_I1_O)        0.043    14.333 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_1/ram_meta_1[54]_i_21/O
                         net (fo=1, routed)           0.316    14.649    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_2/I18
    SLICE_X39Y55         LUT6 (Prop_lut6_I3_O)        0.043    14.692 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_2/ram_meta_1[54]_i_11/O
                         net (fo=1, routed)           0.000    14.692    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_4/I50
    SLICE_X39Y55         MUXF7 (Prop_muxf7_I0_O)      0.107    14.799 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_4/ram_meta_1_reg[54]_i_5/O
                         net (fo=2, routed)           0.416    15.215    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_5/I44
    SLICE_X41Y53         LUT6 (Prop_lut6_I5_O)        0.124    15.339 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_1/tables_5/ram_preds_1_taken[0]_i_4/O
                         net (fo=1, routed)           1.808    17.147    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_1/components_1_io_resp_f3_1_taken
    SLICE_X97Y10         LUT6 (Prop_lut6_I0_O)        0.043    17.190 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_1/ram_preds_1_taken[0]_i_2/O
                         net (fo=4, routed)           1.607    18.797    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_1/io_resp_f3_1_taken
    SLICE_X105Y48        LUT5 (Prop_lut5_I0_O)        0.043    18.840 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_1/ram_edge_inst_1[0]_i_6/O
                         net (fo=2, routed)           1.206    20.046    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/f3_bpd_resp_io_deq_bits_preds_1_taken
    SLICE_X112Y85        LUT6 (Prop_lut6_I0_O)        0.043    20.089 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_edge_inst_1[0]_i_3/O
                         net (fo=2, routed)           0.639    20.729    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_edge_inst_1[0]_i_3
    SLICE_X110Y72        LUT6 (Prop_lut6_I0_O)        0.043    20.772 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_edge_inst_1[0]_i_2/O
                         net (fo=13, routed)          0.824    21.596    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/O50[2]
    SLICE_X103Y73        LUT6 (Prop_lut6_I2_O)        0.043    21.639 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_mask[6]_i_4/O
                         net (fo=5, routed)           0.433    22.072    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_mask[6]_i_4
    SLICE_X103Y69        LUT6 (Prop_lut6_I3_O)        0.043    22.115 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_mask[6]_i_1/O
                         net (fo=9, routed)           1.131    23.245    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
```

```
/tile_reset_domain_boom_tile/frontend/f3/O50[6]
    SLICE_X102Y64        LUT5 (Prop_lut5_I0_O)        0.051    23.296 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_btb_mispredicts_reg_0_1_6_7_i_3/O
                         net (fo=2, routed)           0.342    23.639    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_btb_mispredicts_reg_0_1_6_7_i_3
    SLICE_X102Y62        LUT6 (Prop_lut6_I1_O)        0.138    23.777 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_cfi_idx_valid[0]_i_2/O
                         net (fo=54, routed)          0.471    24.247    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/f4_io_enq_bits_cfi_idx_valid
    SLICE_X101Y61        LUT2 (Prop_lut2_I1_O)        0.051    24.298 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s2_ghist_new_saw_branch_not_taken_i_18/O
                         net (fo=7, routed)           0.455    24.753    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/_f3_predicted_ghist_not_taken_branches_T_161
    SLICE_X100Y60        LUT6 (Prop_lut6_I5_O)        0.136    24.889 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s2_ghist_new_saw_branch_not_taken_i_16/O
                         net (fo=1, routed)           0.351    25.240    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/p_0_in3_out[3]
    SLICE_X100Y59        LUT5 (Prop_lut5_I3_O)        0.043    25.283 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s2_ghist_new_saw_branch_not_taken_i_6/O
                         net (fo=2, routed)           0.371    25.654    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/predicted_ghist_first_bank_saw_not_taken
    SLICE_X98Y56         LUT5 (Prop_lut5_I0_O)        0.043    25.697 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s2_ghist_old_history[63]_i_6/O
                         net (fo=66, routed)          0.983    26.680    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_s2_ghist_old_history[63]_i_6
    SLICE_X72Y38         LUT6 (Prop_lut6_I1_O)        0.043    26.723 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s2_ghist_old_history[12]_i_2/O
                         net (fo=5, routed)           0.529    27.252    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_s2_ghist_old_history[12]_i_2
    SLICE_X71Y39         LUT6 (Prop_lut6_I5_O)        0.043    27.295 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc[39]_i_97/O
                         net (fo=1, routed)           0.000    27.295    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc[39]_i_97
    SLICE_X71Y39         CARRY4 (Prop_carry4_S[0]_CO[3])
                                                      0.259    27.554 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_83/CO[3]
                         net (fo=1, routed)           0.000    27.554    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_83
    SLICE_X71Y40         CARRY4 (Prop_carry4_CI_CO[3])
                                                      0.053    27.607 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_69/CO[3]
                         net (fo=1, routed)           0.000    27.607    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_69
    SLICE_X71Y41         CARRY4 (Prop_carry4_CI_CO[3])
                                                      0.053    27.660 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_50/CO[3]
                         net (fo=1, routed)           0.000    27.660    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_50
    SLICE_X71Y42         CARRY4 (Prop_carry4_CI_CO[3])
                                                      0.053    27.713 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_31/CO[3]
                         net (fo=1, routed)           0.000    27.713    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_31
    SLICE_X71Y43         CARRY4 (Prop_carry4_CI_CO[1])
                                                      0.077    27.790 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_19/CO[1]
                         net (fo=1, routed)           0.706    28.496    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/_f3_correct_f2_ghist_T_20
    SLICE_X92Y50         LUT3 (Prop_lut3_I2_O)        0.122    28.618 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc[39]_i_10/O
                         net (fo=5, routed)           0.985    29.603    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc[39]_i_10
    SLICE_X105Y58        LUT6 (Prop_lut6_I0_O)        0.043    29.646 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc[39]_i_14/O
                         net (fo=1, routed)           0.000    29.646    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc[39]_i_14
    SLICE_X105Y58        MUXF7 (Prop_muxf7_I1_O)      0.108    29.754 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_5/O
                         net (fo=3, routed)           0.613    30.367    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_5
    SLICE_X110Y69        LUT6 (Prop_lut6_I5_O)        0.124    30.491 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/meta_0_0_reg_i_49__0/O
                         net (fo=69, routed)          0.705    31.196    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/O33
    SLICE_X132Y69        LUT6 (Prop_lut6_I1_O)        0.043    31.239 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/meta_0_0_reg_i_1__0/O
                         net (fo=34, routed)          3.242    34.481    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/buffer_1/bundleIn_0_d_q/icache_io_req_valid
    SLICE_X333Y60        LUT3 (Prop_lut3_I0_O)        0.043    34.524 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/buffer_1/bundleIn_0_d_q/tag_array_1_reg_i_1/O
                         net (fo=8, routed)           3.527    38.051    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_0_rw_r_data_pipe_0O__6
    RAMB18_X9Y12         RAMB18E1                                      r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_4_reg/ENARDEN
  -------------------------------------------------------------------    -------------------

                         (clock clk_per_i rise edge)
                                                     33.333    33.333 r
    AB33                                              0.000    33.333 r  c0_sys_clk_p
                         net (fo=0)                   0.000    33.333    io/c0_sys_clk_p
    AB33                 IBUFDS (Prop_ibufds_I_O)     0.775    34.109 r  io/ckbuf/O
                         net (fo=1, routed)           1.597    35.706    io/sys_ref_clk_i
    BUFGCTRL_X0Y48       BUFG (Prop_bufg_I_O)         0.083    35.789 r  io/ckbufg/O
                         net (fo=6, routed)           1.390    37.179    io/n_2_ckbufg
    PLLE2_ADV_X0Y4       PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT1)
                                                      0.073    37.252 r  io/pll_idly/CLKOUT1
                         net (fo=1, routed)           1.427    38.679    io/clk_i
    BUFGCTRL_X0Y32       BUFG (Prop_bufg_I_O)         0.083    38.762 r  io/clkbufg/O
                         net (fo=85441, routed)       1.881    40.643    io/cae0_clock/I1
    SLR Crossing[1->0]
    MMCME2_ADV_X0Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                      0.073    40.716 r  io/cae0_clock/gasync.gbufg.gmmcm.pll/CLKOUT1
                         net (fo=1, routed)           1.429    42.145    io/cae0_clock/clk_per_i
    BUFGCTRL_X0Y3        BUFG (Prop_bufg_I_O)         0.083    42.228 r  io/cae0_clock/gasync.gbufg.per_bufg/O
                         net (fo=12751, routed)       0.119    42.347    ae0/cae0.cae_pers/top/top/sim/target/FireSim/out[0]
    BUFGCTRL_X0Y2        BUFGCTRL (Prop_bufgctrl_I0_O)
                                                      0.083    42.430 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/RationalClockBridge_clocks_0_buffer/O
                         net (fo=110524, routed)      1.643    44.072    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/lazyModule_clocks_subsystem_cbus_0
    RAMB18_X9Y12                                                       r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_4_reg/CLKARDCLK
                         clock pessimism              0.871    44.943
                         clock uncertainty           -0.266    44.678
    RAMB18_X9Y12         RAMB18E1 (Setup_ramb18e1_CLKARDCLK_ENARDEN)
                                                     -0.328    44.350    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
```

```
/tile_reset_domain_boom_tile/frontend/icache/tag_array_4_reg
  ---------------------------------------------------------------------
                         required time                           44.350
                         arrival time                           -38.051
  ---------------------------------------------------------------------
                         slack                                    6.299
```

```
Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Tool Version      : Vivado v.2014.4 (lin64) Build 1071353 Tue Nov 18 16:47:07 MST 2014
| Date              : Wed Jul  7 09:21:51 2021
| Host              : WX2000Server running 64-bit CentOS release 6.10 (Final)
| Command           : report_timing -max_paths 100 -through [get_cells ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile] -name boom -file modified_timing.rpt
| Design            : aemc_fpga
| Device            : 7v2000t-fhg1761
| Speed File        : -2  PRODUCTION 1.10 2014-09-11
| Temperature Grade : C
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Timing Report

Slack (MET) :             7.701ns  (required time - arrival time)
  Source:                 ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/REG_15_reg/C
                            (rising edge-triggered cell FDRE clocked by clk_per_i  {rise@0.000ns fall@16.667ns period=33.333ns})
  Destination:            ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_7_reg/ENARDEN
                            (rising edge-triggered cell RAMB18E1 clocked by clk_per_i  {rise@0.000ns fall@16.667ns period=33.333ns})
  Path Group:             clk_per_i
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            33.333ns  (clk_per_i rise@33.333ns - clk_per_i rise@0.000ns)
  Data Path Delay:        24.693ns  (logic 2.166ns (8.772%)  route 22.527ns (91.228%))
  Logic Levels:           24  (CARRY4=2 LUT3=2 LUT5=3 LUT6=14 MUXF7=3)
  Clock Path Skew:        -0.346ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    10.382ns = ( 43.715 - 33.333 )
    Source Clock Delay      (SCD):    11.601ns
    Clock Pessimism Removal (CPR):    0.873ns
  Clock Uncertainty:      0.266ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.526ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock clk_per_i rise edge)
                                                      0.000     0.000 r
    AB33                                              0.000     0.000 r  c0_sys_clk_p
                         net (fo=0)                   0.000     0.000    io/c0_sys_clk_p
    AB33                 IBUFDS (Prop_ibufds_I_O)     0.877     0.877 r  io/ckbuf/O
                         net (fo=1, routed)           1.712     2.589    io/sys_ref_clk_i
    BUFGCTRL_X0Y48       BUFG (Prop_bufg_I_O)         0.093     2.682 r  io/ckbufg/O
                         net (fo=6, routed)           1.552     4.234    io/n_2_ckbufg
    PLLE2_ADV_X0Y4       PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT1)
                                                      0.077     4.311 r  io/pll_idly/CLKOUT1
                         net (fo=1, routed)           1.541     5.852    io/clk_i
    BUFGCTRL_X0Y32       BUFG (Prop_bufg_I_O)         0.093     5.945 r  io/clkbufg/O
                         net (fo=85441, routed)       2.071     8.016    io/cae0_clock/I1
    SLR Crossing[1->0]
    MMCME2_ADV_X0Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                      0.077     8.093 r  io/cae0_clock/gasync.gbufg.gmmcm.pll/CLKOUT1
                         net (fo=1, routed)           1.543     9.636    io/cae0_clock/clk_per_i
    BUFGCTRL_X0Y3        BUFG (Prop_bufg_I_O)         0.093     9.729 r  io/cae0_clock/gasync.gbufg.per_bufg/O
                         net (fo=12751, routed)       0.132     9.861    ae0/cae0.cae_pers/top/top/sim/target/FireSim/out[0]
    BUFGCTRL_X0Y2        BUFGCTRL (Prop_bufgctrl_I0_O)
                                                      0.093     9.954 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/RationalClockBridge_clocks_0_buffer/O
                         net (fo=111720, routed)      1.647    11.601    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/lazyModule_clocks_subsystem_cbus_0
    SLICE_X423Y39                                                       r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/REG_15_reg/C
  -------------------------------------------------------------------    -------------------
    SLICE_X423Y39        FDRE (Prop_fdre_C_Q)         0.223    11.824 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/REG_15_reg/Q
                         net (fo=85, routed)          1.936    13.760    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/REG_15
    SLICE_X496Y14        LUT3 (Prop_lut3_I2_O)        0.043    13.803 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_2/ram_meta_0[55]_i_16/O
                         net (fo=2, routed)           0.532    14.336    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_0/components_2_io_resp_f3_2_taken
    SLICE_X504Y13        LUT5 (Prop_lut5_I4_O)        0.043    14.379 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_0/ram_meta_0[55]_i_27/O
                         net (fo=1, routed)           0.264    14.643    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_0/n_2_ram_meta_0[55]_i_27
    SLICE_X503Y14        LUT6 (Prop_lut6_I5_O)        0.043    14.686 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_0/ram_meta_0[55]_i_25/O
                         net (fo=1, routed)           0.000    14.686    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_2/I16
    SLICE_X503Y14        MUXF7 (Prop_muxf7_I1_O)      0.108    14.794 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_2/ram_meta_0_reg[55]_i_18/O
                         net (fo=1, routed)           0.239    15.033    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_2/n_2_ram_meta_0_reg[55]_i_18
    SLICE_X502Y14        LUT6 (Prop_lut6_I0_O)        0.124    15.157 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_2/ram_meta_0[55]_i_12/O
                         net (fo=1, routed)           0.000    15.157    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_4/I36
    SLICE_X502Y14        MUXF7 (Prop_muxf7_I1_O)      0.108    15.265 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_4/ram_meta_0_reg[55]_i_3/O
                         net (fo=2, routed)           0.330    15.595    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_5/I30
    SLICE_X499Y14        LUT6 (Prop_lut6_I5_O)        0.124    15.719 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_1/tables_5/ram_preds_2_taken[0]_i_7/O
                         net (fo=1, routed)           2.028    17.747    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_0/columns_2/components_1_io_resp_f3_2_taken
    SLICE_X387Y7         LUT6 (Prop_lut6_I0_O)        0.043    17.790 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_0/components_0/columns_2/ram_preds_2_taken[0]_i_3/O
                         net (fo=4, routed)           1.432    19.222    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_2/banked_predictors_0_io_resp_f3_2_taken
    SLICE_X409Y18        LUT5 (Prop_lut5_I2_O)        0.054    19.276 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/bpd/banked_predictors_1/components_0/columns_2/ram_sfbs_3[0]_i_9/O
                         net (fo=2, routed)           0.877    20.153    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/f3_bpd_resp_io_deq_bits_preds_2_taken
    SLICE_X422Y33        LUT6 (Prop_lut6_I0_O)        0.137    20.290 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_sfbs_3[0]_i_6/O
                         net (fo=2, routed)           0.325    20.615    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_sfbs_3[0]_i_6
    SLICE_X428Y30        LUT6 (Prop_lut6_I5_O)        0.043    20.658 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_sfbs_4[0]_i_8/O
                         net (fo=2, routed)           0.356    21.014    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_sfbs_4[0]_i_8
    SLICE_X429Y30        LUT6 (Prop_lut6_I3_O)        0.043    21.057 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_mask[6]_i_3/O
                         net (fo=4, routed)           0.296    21.354    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
```

```
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_mask[6]_i_3
    SLICE_X430Y30        LUT6 (Prop_lut6_I2_O)        0.043    21.397 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_mask[6]_i_1/O
                         net (fo=9, routed)           0.373    21.769    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/O52[6]
    SLICE_X430Y31        LUT6 (Prop_lut6_I0_O)        0.043    21.812 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_cfi_idx_bits[0]_i_2/O
                         net (fo=160, routed)         1.817    23.629    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_cfi_idx_bits[0]_i_2
    SLICE_X387Y45        MUXF7 (Prop_muxf7_S_O)       0.147    23.776 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s1_vpc_reg[24]_i_12/O
                         net (fo=1, routed)           1.533    25.309    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_s1_vpc_reg[24]_i_12
    SLICE_X412Y48        LUT6 (Prop_lut6_I0_O)        0.124    25.433 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s1_vpc[24]_i_9/O
                         net (fo=1, routed)           0.606    26.040    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_s1_vpc[24]_i_9
    SLICE_X438Y49        LUT5 (Prop_lut5_I4_O)        0.043    26.083 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/s1_vpc[24]_i_6/O
                         net (fo=5, routed)           0.745    26.828    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/predicted_target[24]
    SLICE_X439Y46        LUT6 (Prop_lut6_I3_O)        0.043    26.871 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc[39]_i_30/O
                         net (fo=1, routed)           0.000    26.871    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc[39]_i_30
    SLICE_X439Y46        CARRY4 (Prop_carry4_S[0]_CO[3])
                                                      0.259    27.130 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_16/CO[3]
                         net (fo=1, routed)           0.000    27.130    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc_reg[39]_i_16
    SLICE_X439Y47        CARRY4 (Prop_carry4_CI_CO[1])
                                                      0.077    27.207 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc_reg[39]_i_8/CO[1]
                         net (fo=2, routed)           0.820    28.026    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/_T_942
    SLICE_X456Y33        LUT6 (Prop_lut6_I0_O)        0.122    28.148 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/ram_pc[39]_i_2/O
                         net (fo=3, routed)           0.875    29.023    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/n_2_ram_pc[39]_i_2
    SLICE_X448Y43        LUT6 (Prop_lut6_I4_O)        0.043    29.066 f  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/f3/meta_0_0_reg_i_50__0/O
                         net (fo=2, routed)           0.869    29.935    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/core/rob/I787
    SLICE_X424Y53        LUT6 (Prop_lut6_I3_O)        0.043    29.978 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/core/rob/meta_0_0_reg_i_1__0/O
                         net (fo=34, routed)          3.611    33.589    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/buffer_1/bundleIn_0_d_q/icache_io_req_valid
    SLICE_X310Y22        LUT3 (Prop_lut3_I0_O)        0.043    33.632 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/buffer_1/bundleIn_0_d_q/tag_array_1_reg_i_1/O
                         net (fo=8, routed)           2.662    36.294    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_0_rw_r_data_pipe_00__6
    RAMB18_X4Y1          RAMB18E1                                      r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_7_reg/ENARDEN
  -------------------------------------------------------------------    -------------------

                         (clock clk_per_i rise edge)
                                                     33.333    33.333 r
    AB33                                              0.000    33.333 r  c0_sys_clk_p
                         net (fo=0)                   0.000    33.333    io/c0_sys_clk_p
    AB33                 IBUFDS (Prop_ibufds_I_O)     0.775    34.109 r  io/ckbuf/O
                         net (fo=1, routed)           1.597    35.706    io/sys_ref_clk_i
    BUFGCTRL_X0Y48       BUFG (Prop_bufg_I_O)         0.083    35.789 r  io/ckbufg/O
                         net (fo=6, routed)           1.390    37.179    io/n_2_ckbufg
    PLLE2_ADV_X0Y4       PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT1)
                                                      0.073    37.252 r  io/pll_idly/CLKOUT1
                         net (fo=1, routed)           1.427    38.679    io/clk_i
    BUFGCTRL_X0Y32       BUFG (Prop_bufg_I_O)         0.083    38.762 r  io/clkbufg/O
                         net (fo=85441, routed)       1.881    40.643    io/cae0_clock/I1
    SLR Crossing[1->0]
    MMCME2_ADV_X0Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                      0.073    40.716 r  io/cae0_clock/gasync.gbufg.gmmcm.pll/CLKOUT1
                         net (fo=1, routed)           1.429    42.145    io/cae0_clock/clk_per_i
    BUFGCTRL_X0Y3        BUFG (Prop_bufg_I_O)         0.083    42.228 r  io/cae0_clock/gasync.gbufg.per_bufg/O
                         net (fo=12751, routed)       0.119    42.347    ae0/cae0.cae_pers/top/top/sim/target/FireSim/out[0]
    BUFGCTRL_X0Y2        BUFGCTRL (Prop_bufgctrl_I0_O)
                                                      0.083    42.430 r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/RationalClockBridge_clocks_0_buffer/O
                         net (fo=111720, routed)      1.286    43.715    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/lazyModule_clocks_subsystem_cbus_0
    RAMB18_X4Y1                                                        r  ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_7_reg/CLKARDCLK
                         clock pessimism              0.873    44.588
                         clock uncertainty           -0.266    44.323
    RAMB18_X4Y1          RAMB18E1 (Setup_ramb18e1_CLKARDCLK_ENARDEN)
                                                     -0.328    43.995    ae0/cae0.cae_pers/top/top/sim/target/FireSim/lazyModule/system/tile_prci_domain
/tile_reset_domain_boom_tile/frontend/icache/tag_array_7_reg
  -------------------------------------------------------------------
                         required time                         43.995
                         arrival time                         -36.294
  -------------------------------------------------------------------
                         slack                                  7.701
```

Amund Bergland Kvalsvik

Investigating the Usefulness of Simulators and Prototypes as Research Tools

NTNU
Norwegian University of
Science and Technology