Erling Rennemo Jellum
David Christoph Metz

Evaluating FIFO-based Instruction Scheduling Techniques using FPGAs

Master's thesis in MTTK
Supervisor: Magnus Själander, Sverre Hendseth
June 2020

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Erling Rennemo Jellum
David Christoph Metz

Evaluating FIFO-based Instruction Scheduling Techniques using FPGAs

Master's thesis in MTTK
Supervisor: Magnus Själander, Sverre Hendseth
June 2020

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

Contents

List of Figures

List of Tables

Acknowledgments

We would like to thank the following persons for their help during the work with our master thesis. First and foremost we want to thank our main supervisor, Magnus Själander, for his excellent feedback and for guiding us in the right direction. We are also thankful for the opportunity to pursue a topic of our choosing. We also would like to thank Rakesh Kumar, for sharing his work and giving us valuable feedback. We would also like to thank Erling's supervisor at ITK, Sverre Hendseth, for supporting us in doing a cross-departmental thesis.

Finally, we would like to extend our thanks to Christopher Celio, Jerry Zhao and the rest of the BOOM team at UC Berkeley. Their work with developing BOOM is fantastic, and they were always very helpful and responsive to inquiries.

D.M. & E.J.

Abstract

The performance advantage of out-of-order processors stems from their ability to extract more instruction-level parallelism (ILP) and memory-level parallelism (MLP) than in-order cores. This is largely the benefit of the dynamic out-of-order schedules they create. The downside of out-of-order scheduling is that it comes at high energy and die-area cost. We evaluate three recently proposed FIFO-based scheduling techniques found in Load Slice Core (LSC), Delay and Bypass (DnB), and CASINO. They all promise a large part of the performance gain of out-of-order scheduling, but at a much lower cost.

DnB and LSC focus on extracting MLP by iteratively building load slices and giving them precedence in the execution order. The dependency analysis technique they employ is called Iterative Backward Dependency Analysis (IBDA). We evaluate implementability, performance, and area of the proposed IBDA as well as proposing three improved implementations of IBDA that require less area and power while providing essentially the same performance. DnB, LSC, and CASINO, the third technique, are all based around the idea of replacing the expensive, content addressable issue queue with cheaper FIFOs.

We implement all these techniques based on BOOM, an open-source, RTL implementation of an out-of-order RISC-V core. We synthesize our designs and evaluate them on a Xilinx ZC707 FPGA. By instantiating our cores as part of a system on chip, we are also able to boot Linux on them.

Our evaluation, using parts of the SPEC CPU2006 benchmark suite, confirms the claims that these techniques come close to the performance of a fully-fledged out-of-order core. LSC and CASINO do this while consuming noticeably fewer resources. DnB comes closest to the performance of out-of-order cores, but it fails to show area-benefits in our implementation. Additionally, we provide insights into the overheads that the BOOM core has over its smaller sibling, the in-order processor Rocket.

As this form of implementation is much closer to real silicon tapeouts than simulators, it forces us to consider and analyze implementation specifics that can be ignored in high-level simulation. This provides insights into the implementability of the different techniques.

Our work provides a big step towards providing accurate measurements, instead of estimates, of performance, power, and area usage for LSC, DnB, and CASINO.

1 Introduction

In 1974 Robert Dennard, an IBM researcher, saw that by reducing the size of a transistor, it was possible to reduce its switching voltage, increase the switching frequency and still keep the power consumption per square mm constant [3]. This was later referred to as Dennard Scaling. It meant that, as transistors kept getting smaller, the operating voltage could be decreased, and the operating frequency could be increased, without using more power. For each technology node, a bigger processor that still wouldn't consume more power, or take up more valuable die area, could be designed. Computer architects could optimize solely for instruction-level-parallelism without being too concerned about area or power consumption [4]. This has led to complex and power-inefficient super-scalar out-of-order designs, some of which have instruction windows of over 200 instructions [5]. What Dennard scaling ignores is the constant leakage current and threshold voltage which contributes to the power density in the transistor and doesn't scale down with the size. As the leakage current starts to dominate, power consumption per transistor no longer scales down with smaller transistors. This means that while more transistors are available, their power consumption per square millimeter rises [6].

This has created the so-called Power Wall [7] which not only limits the clock frequency of desktop processors to around 4 GHz, but also makes reducing the power consumption of mobile processors more challenging. Today energy-efficiency is the key evaluation metric for computer architects [4].

The rise of the Internet of Things creates a need for very low-power processors that still offer enough performance to perform meaningful tasks. These systems are connected to the internet and often run full operating systems like Linux. In order to keep up with the rising demand for compute of more and more advanced cryptographic standards, these systems should offer a high amount of performance for their power envelope. This is important as it keeps these systems from becoming obsolete or insecure quickly. Furthermore, some applications only become possible if processors with sufficient performance in a small power envelope are available.

This has led to a series of innovative designs trying to maximize the performance per watt and performance per square millimeter of processors. Central to some of these designs are novel approaches to instruction scheduling. To reduce power and area they try to harness as much of the performance gains of out-of-order cores as possible, while keeping the cheap and simple structures of in-order cores.

This report is split into two parts. In the first part we look at Iterative Backwards Dependency Analysis (IBDA), a promising hardware technique that builds program slices. IBDA is a key enabling technique in several promising architectures, including the Load Slice Core [1], Delay and Bypass [8] and Freeway [9]. We introduce Single Write IBDA, Fuzzy IBDA and Bloom IBDA, three optimizations that greatly reduce the original IBDAs area and power consumption.

In the second part we present our implementation of, and evaluate, three promising, recently proposed instruction-scheduling techniques that aim to extract *some* instruction-level parallelism, but without needing all the expensive structures of conventional out-of-order designs. The techniques share the core idea of replacing expensive issue queues with cheap FIFOs. We, therefore, explore different designs for these FIFOs. The *Load Slice Core* [1] adds a bypass instruction queue to a stall-on-use in-order processor, to let memory slices bypass the rest of the instruction stream and executed early. *Delay and Bypass* [8], keeps the expensive out-of-order structures but tries to reduce their sizes at minimal performance loss. *CASINO* [10] adds a simple instruction queue to a stall-on-use in-order processor and lets instructions execute early if they are ready.

We implemented these techniques based on the most recent version of the BOOM core, BOOM v3 [11], an open-source core written in the hardware description language Chisel [12]. Furthermore, we evaluate them on an FPGA, integrated in the Chipyard [13] system on chip platform. Our evaluation includes size estimates based on FPGA synthesis, application performance under both Linux and a lightweight micro-kernel. To gauge the overhead an implementation based on BOOM incurs, we compare a simplified in-order version of BOOM to Rocket [14]. We also report on difficulties and caveats encountered during the implementation of these microarchitectures.

Part I

Iterative Backwards Dependency Analysis

2 Background

This chapter will introduce the main concepts underlying IBDA, Single Write IBDA, Fuzzy IBDA and Bloom IBDA. The sections Program Slices and Dependency Analysis lay the theoretical groundwork for dependency analysis in general. Sections Cashes and Memories give a foundation for understanding how different design choices for the IST affect how it is synthesized. Hashing introduces concepts used by Fuzzy IBDA which sacrifices precision for reduced area. Lastly we will introduce IBDA, as it was described by Carlson et al. [1].

2.1 Program Slices

IBDA builds on the concept of program slices [15]. A program slice is a series of dependent instructions which leads up to, or away from, a *criterion*. A criterion is an event, typically performance-degrading, which can occur at a point in the program if certain prerequisite are fulfilled. This can for instance be a branch misprediction, or a cache miss.

A slice can be broken up into four components. The *value sub-slice* consists of instructions that manipulate the input operands of the criterion. The *address sub-slice* consists of instructions that calculate the memory addresses, either for the criterion itself, when it's a store or a load, or for the value sub-slice. The *existence sub-slice* consists of branches that determine whether the criterion instruction will be executed at all. Lastly, the *control flow sub-slice* consists of branches, that decide which of multiple paths to the criterion the program will take.

Instructions are recognized as part of a sub-slice, if there is a chain of dependencies from it to the criterion. Dependencies come in two flavors. *Data dependencies* occur when the input operand of an instruction is the output of another. *Control dependencies* exist between an instruction and a branch, if the branch decides whether the instruction will execute or not. Furthermore, a data dependency is an *address dependency*, if the data contributes to the calculation of a memory address.

A slice architecture is one that is centered around extracting and diverting slices leading up to certain criteria. Examples of slice architectures are the Decoupled Access/Execute Architecture [16], Load Slice Core [1], Delay and Bypass [8], and Branch Slice Core [17].

2.2 Dependency Analysis

There are several techniques for detecting slices. The simplest option, from a hardware perspective, is to let the compiler do the analysis [18]. The disadvantage of this approach is that it is not backwards compatible. A proposed hardware solution is to store the N latest committed instructions with their dependencies in a FIFO buffer, called the *Slicer*, and, upon detecting a criterion, traversing the Slicer and extracting the PCs of the slice [19]. The slices are stored in a cache indexed by the PC of the lead instruction. This sets the maximum number of instructions to be stored within a single slice cache entry as the limit to the maximum slice length. The depth of the Slicer, N, sets the limit for how far into the past a slice can be computed.

2.3 Hashing

This section describes the basics of hash functions and entropy. Hashing is an operation $h(x) = y$, mapping an arbitrary sized input x, called a key, to a fixed length output y, the hash, typically of a smaller length [20] Since y is of a fixed length and x can be of arbitrary length, there will be multiple inputs x_1, x_2 which map to the same output. $h(x_1) = h(x_2)$. Such a scenario is called a collision. An ideal implementation of $h(x)$ is cheap to compute and distributes the keys uniformly among the output values, thus minimizing collisions. This means, an ideal implementation is very dependent on the set of input keys it will receive.

2.3.1 Entropy

Entropy is a measure of randomness or unpredictability. Given a set X, consisting of n-bit symbols, the entropy $H(X)$ of the set is the average number of bits per symbol needed to encode all symbols in X [21] Mathematically entropy can be expressed as

$$H(X) = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i) \tag{2.1}$$

Where $P(X)$ is the probability distribution for the symbols in X.

If the symbols in X are uniformly distributed, i.e. it is equally likely to draw any symbol from that set, the set would have $log_2 n$ bits of entropy, where n is the number of unique symbols. However, if the distribution is not uniform, but rather skewed towards some value, there would be less entropy because there is less randomness. Consider for instance the set $X_1 = \{a, b, c, d\}$. With a uniform probability distribution, i.e. $P(X_1) = \{0.25, 0.25, 0.25, 0.25\}$. The entropy for X_1 is $H(X_1) = 2$. However, consider the set X_2 which also contains only four different types of symbols, but has a non-uniform probability distribution $P(X_2) = \{0.7, 0.1, 0.1, 0.1\}$. The entropy is in this case $H(X_2) = 1.86$. It is lower because it is a set with a lower degree of randomness. An efficient hash function $h(x)$ will retain as much as possibly of the entropy of the input keys and spread it to the output bits.

2.4 Memories

Regular SRAMs use shared bit lines for reading and writing. This means they can perform one read or write during each cycle. Adding the ability to read and write in the same cycle generally doubles the area of a single SRAM cell. Since the size of an SRAM is largely dominated by the cells, the total size effectively doubles. Additional ports can be added by using additional bit lines. As this increases the fan-out of the cells, the transistors need to drive a higher load and their sizes might need to be increased. [22]

While it is possible to customize the SRAMs and add additional ports when targeting ASICs, the limits of resources within FPGAs can not be altered. The Xilinx Zynq FPGAs we use for evaluation provide SRAM blocks with synchronous reads called BRAMs. These 18 Kbits large blocks can be

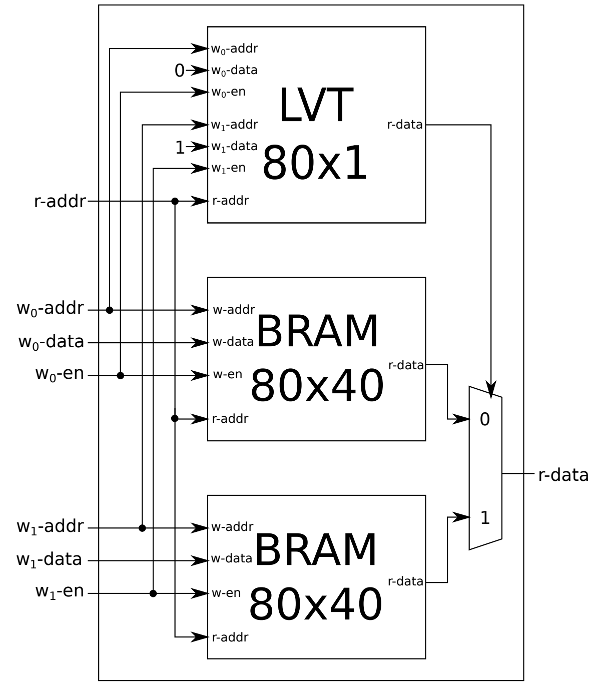


Figure 1: Memory with two write ports and one read ports using LVT

configured to different widths and sizes. Each BRAM provides two ports that can both either read or write (true dual port). [23]

The other type of SRAM available in Zynq FPGAs is called Distributed RAM. Up to a quarter of the SRAM-based look-up tables (LUTs) that are used to implement combinational logic can be re-purposed as Distributed RAMs. Distributed RAMs support both synchronous and asynchronous reads, and consist of up to 4 LUTs. They provide one primary port that can both read and write, as well as up to 3 additional read-only ports. Depending on the port configuration their size ranges from 32x1 to 256x1 with options for 32x6 and 64x3. [24]

Multiple BRAMs can be combined to emulate an SRAM with more ports. To emulate a single-write-multiple-read SRAM, multiple BRAMs can be used without requiring external logic, by writing the same data to one port of all BRAMs and using the other ports as independent reads. Adding write ports is not possible without external logic. Different schemes that can be used to emulate such a memory on an FPGA were described by LaForest et al. [25]. The simplest but most expensive option is emulating the SRAM using registers and LUTs.

A more advanced scheme uses a Live Value Table (LVT). An example of a memory using the LVT is shown in Figure 1. Here BRAMs are used to store the data itself and a LVT, a small multi-ported structure implemented using LUTs and registers, to select the BRAM with the right data.

A general factor that has to be decided for SRAMs with multiple write ports is precedence. If several ports write to the same address, the precedence decides which write will be stored. In our implementation higher ports have precedence over lower ones. When multiple instructions write data to an SRAM, the data from the latest instruction is used.

2.5 Caches

Building hierarchical memories with caches is one of the main ideas underlying modern high performance computing [26]. In this section we will introduce the cache concepts needed to build efficient IBDA. We will look at caches from the perspective of their main use-case - storing a subset of main memory in a small and low-latency structure to decrease average memory latency.

The idea of the cache is to create a small and fast memory that stores important data. Caches in the memory hierarchy contain some subset of the full main memory. An address location can be quickly looked up and fetched if it resides in this kind of cache. If it is not present in the cache it can be fetched from main memory and kept in the cache for the next time the program wants to access it.

One of the main characteristics of a cache is its associativity. Associativity defines how much flexibility there is in choosing a cache entry for an address. A cache where an address can map to any location in the cache is said to be *fully associative*. To look up an address in such a cache all its entries must be read, because a match could be found anywhere. A cache where each address maps to a single entry is called *direct mapped*. In this case each address is mapped to one specific cache entry. A middle ground between fully associative and direct mapped is *set associative*. A set associative cache consists of m sets, each consisting of n ways, for a total of $m \times n$ entries. Addresses are mapped to one of the m sets, but the placement within the set, i.e. in which way, is up to the cache. To look up an address in a set associative cache, all the ways of the set to which the address maps have to be read. A n-entry fully associative cache is equivalent of a n-entry n-way set associate cache. A n-entry directly mapped cache is equivalent of a n-entry 1-way set associative cache.Riot control agents are designed to cause irritation within seconds of exposure, making the exposed want to flee the scene. And indeed, toxicologists advise that getting away from the gas is the best and first thing to do to mediate the impact. To add an entry to a set associative cache with more than one way, a replacement policy deciding which entry to overwrite is needed. The standard replacement policy is Least Recently Used (LRU), which overwrites the entry which was least recently accessed. For a 2-way cache this can be implemented with a single bit per set. The cache only contains a small subset of the main memory and thus will have cache misses. Cache misses can be divided in three classes [27].

1. A *compulsory* cache-miss happens the very first time a program accesses an address and will always result in a cache miss.
2. A *capacity* cache-miss occur if the set of addresses that the program access, i.e. the working set, is bigger than the cache.
3. A *conflict* cache-miss occurs when an address has been evicted because the number of working set addresses that map to a cache set is greater than the number of ways.

2.6 Iterative Backwards Dependency Analysis

IBDA is a hardware technique for building program slices of data-dependent instructions. It is implemented using two structures. The *Register Dependency Table* (RDT) contains, for all physical registers, the PC of the instruction that last wrote to it. Another way to view it is that the RDT contains the producers of all the registers. When an instruction that is part of a program slice is entered into the RDT, the PC of the instructions that produced its source operands can be looked up. These PCs can be added to the *Instruction Slice Table* (IST), the second component of IBDA. The RDT also stores a bit indicating whether the producer of a register is already present in the IST, to avoid writing the same PC to the IST multiple times. [1]

The IST stores all the instructions which have been identified by the RDT as part of a program slice. Figure 2 shows an overview of how IBDA is placed in the Load Slice Core. Instructions are looked up in the IST during the decode stage and the RDT is updated during the rename stage. Instructions that are found in the IST are tagged as a part of a program slice. We call these *marked* instructions. When such a marked instruction is entered into the RDT, its source registers are looked up and the producers of the source registers are added to the IST. [1]

Consider the program in Listing 2.1. Let's assume that the criterion is a load, i.e. we are tracking load slices. Our loop is centered around instruction (1) which loads data. Instruction (2)-(6) are transforming the loaded data and writing it back to memory. Instructions (7) and (8) calculate the next load address, and are thus part of the load slice. At the beginning of the program the IST is empty. During the first iteration of the loop the IST will be updated with the *li a3, 0x800500c* instruction which is the producer of the first load address. But at the second iteration of the loop when we get to (1), the RDT will map (8) as the producer of *2*. Thus (8) is added to the IST. Later in that same iteration when we get to (8), it will be present in the IST. So when (8) is entered into the RDT it is marked as part of a program slice and we will look up the producer of its only dependency *a2*. This is (7), so it is added to the IST. In the next iteration when we get to (7) we will look up its dependency, which is a1, and its producer, (1). Thus we have the whole program slice resulting in the load at (1), stored in the IST.

Listing 2.1: Example of a loop to show how IBDA works

```
init:
    li a0, 100 ; Loop iterator
    li a2, 0x8000301c ; Intial load address
    li a3, 0x8005000c ; Initial store address
    j loop

loop:
(1) lw a1, 0(a2) ; Load data from memory
(2) mul a4, a1, a5 ; Do arithmetic on data
(3) add a6, a4, a1
(4) div a7, a4, a6
(5) sw a6, 0(a3) ; Store result in other memory location
(6) addi a3, 32 ; Calculate next store address
(7) mul a2, a1, 99 ; Calculate next load address
(8) addi a2, 32 ; Calculate next load address cont'd
(9) addi a0, -1 ; Decrement loop counter
(10) bnez a0, loop
```

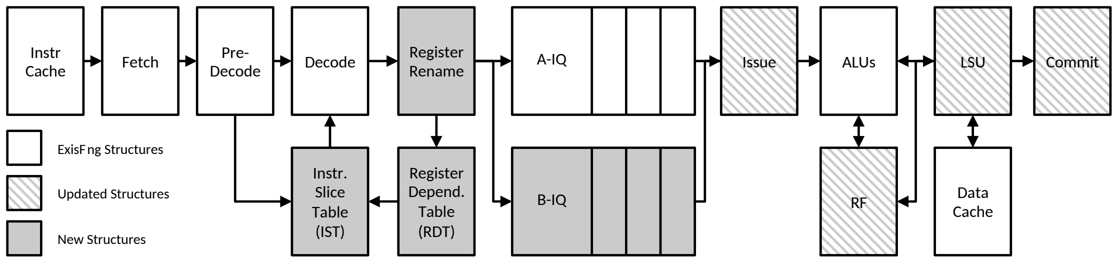


Figure 2: IBDA embedded in the Load Slice Core. Adapted from [1].

3 IBDA implementations

In this chapter we motivate and introduce our IBDA improvements. They were all implemented as part of our Load Slice Core (LSC) implementation, which is based on a 2-wide BOOM core [28]. LSC uses loads and stores as program slice criteria, and thus is tracking load slices [1]. LSC is discussed in greater detail in Part II of this report. The 2-wide LSC decodes and renames two instructions in parallel each cycle.

We are particularly interested how different design choices affect the number of read and write ports to the SRAMs, which will hold the RDT data and IST data. We motivate some of our design choices with simulation data from Spike, a high-level RISC-V simulator. This should not be confused with the cycle accurate simulation results presented in Section 10.5.

3.1 Perfect IBDA

Perfect IBDA is a theoretical implementation with a limitless fully associative cache as the IST. Perfect IBDA will still not be able to correctly identify all load slices, as it also needs to iteratively compute them. It is not synthesizable and is, unlike the others, only implemented in Spike. By implementing Perfect IBDA *together* with the other each of the other IBDA designs we can evaluate their performance normalized to the maximum achievable performance.

3.2 Original IBDA

The Original IBDA is a straightforward implementation as described by Carlson et al. [1]. The IST is a 128 entry, 2-way set associative cache indexed by the 40 bit PC of each instruction. The RDT has 80 entries, as many as there are physical registers in our core.

3.2.1 RDT

To support the dual write needed by a 2-wide core, we need two SRAM banks and an LVT to implement the the RDT. As each instruction that is looked up can have up to two dependencies, the RDT has a total of four read ports. Likewise, the RDT needs four output ports to write the dependencies to the IST. Since we are working with a 2-wide core we must also consider the scenario when the instructions added to the RDT are data-dependent. If the later instructions are consuming the result of the former, then that later instruction will do a lookup in the RDT entry that the former writes to. Writing and reading to the same address in the same cycle has undefined behavior for many SRAMs, including the BRAMs on out ZC706 [23]. Bypass registers are added to avoid this.

3.2.2 IST

When implementing a cache one of the design choices is how to compute the index. For the Original IBDA we compute it as $PC[1 : 6]$ XOR $PC[7 : 12]$. The least significant bit is left out as it is always zero in RISC-V, as instructions are half-word aligned. The reason for the XOR operation is to increase the entropy of the least significant bit of the index. $PC[1]$ is only used when using compressed instructions and thus there is a risk of having programs where there is no entropy in this bit.

The IST essentially does two things. The first is that it looks up instructions during decode to identify them as part of a program slice or not. As our design uses a 2-wide decode stage, this requires two read ports to each of the SRAMs which contain the different ways. The second is to write the PCs it receives from the RDT to its cache so that they later will be identified as part of a program slice. This naturally requires one write port per instruction that could be written, for a total of four.

3.3 Single Write IBDA

The Original IBDA has a major flaw. The IST needs four write ports from the RDT. As discussed in Section 2.4, implementing memories with multiple read and write ports is expensive. Figure 4a shows how often the different write ports where used for a 128 entry 2-way set associative design running SPEC. On average, 93% of the writes from RDT to IST only used one of the four write ports.

To solve this issue we propose Single Write IBDA. Single Write IBDA use the fact that the RDT already stores whether the producer of the physical registers are already present in the IST or not, to reduce the number of write ports from four to one. Figure 3 shows the overview of the RDT for a 2-wide core with only one write port to the IST. There are mainly two things happening. First, *ren_uop1* and *ren_uop2* update the RDT SRAM by writing their addresses, *pc*, to the position corresponding to their destination registers. *rdst*. This keeps the RDT updated with the addresses of all the producers of the registers. Then, we check whether *ren_uop1* and *ren_uop2* are marked *in_ist*. This signal was updated by the IST the previous cycle. If *ren_uop1* and *ren_uop2* are marked as in the IST we proceed to check whether the producers of the source operands, *rs1* and *rs2* are already written to the IST. This is stored with flip-flops in the structure we call *in_ist*. The source operands which are *not* already present in the IST are fed into a priority MUX which takes the first operand of the youngest micro-op available and forwards it to the read port of the RDT SRAM. In essence, we look up the producer of the first source operand that is not already present in the IST. This producer is then written to the IST.

Consider the scenario where both *ren_uop1* and *ren_uop2* are marked as in the IST, and both have two operands with producers that are *not* marked as in the IST. In this scenario Original IBDA would use all four write ports to add all four producers to the IST. Single Write IBDA will only add the producer of the first *ren_uop1* operand, i.e. the instruction that last wrote to rs1 or *ren_uop1*. In the following iteration that first producer will be marked as in IST and we will write the second

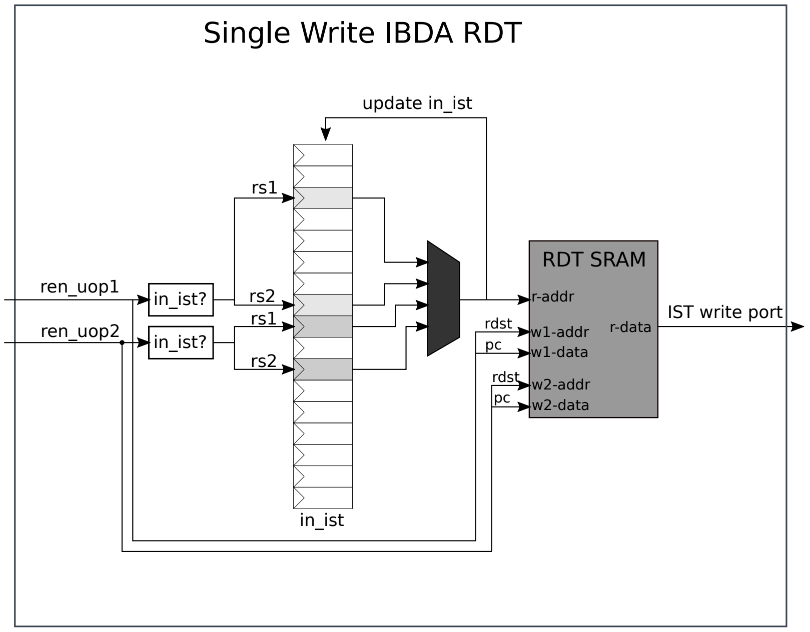


Figure 3: Single Write IBDA RDT with one write port to the IST

producer, i.e. the instruction that last wrote rs2 of *ren_uop1*. In essence Single Write IBDA doubles down on the iterative approach of IBDA by adding only one producer at a time to the IST.

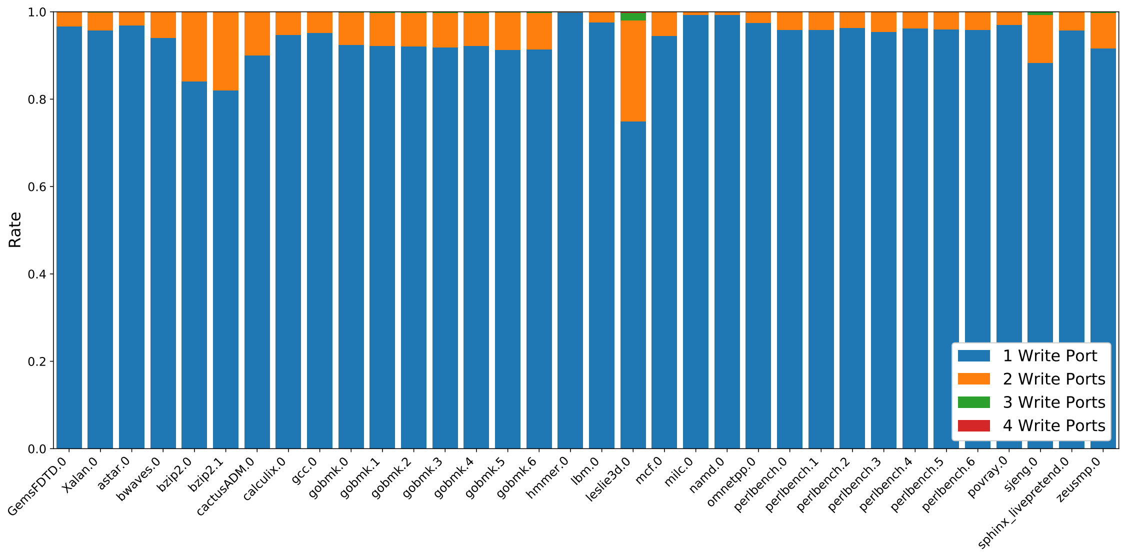
Figure 4b compares the load slice extraction rate of Single Write IBDA with one write port and Original IBDA with four write ports. Notice that reducing the number of write ports has a negligible effect on slice extraction.

3.4 Fuzzy IBDA

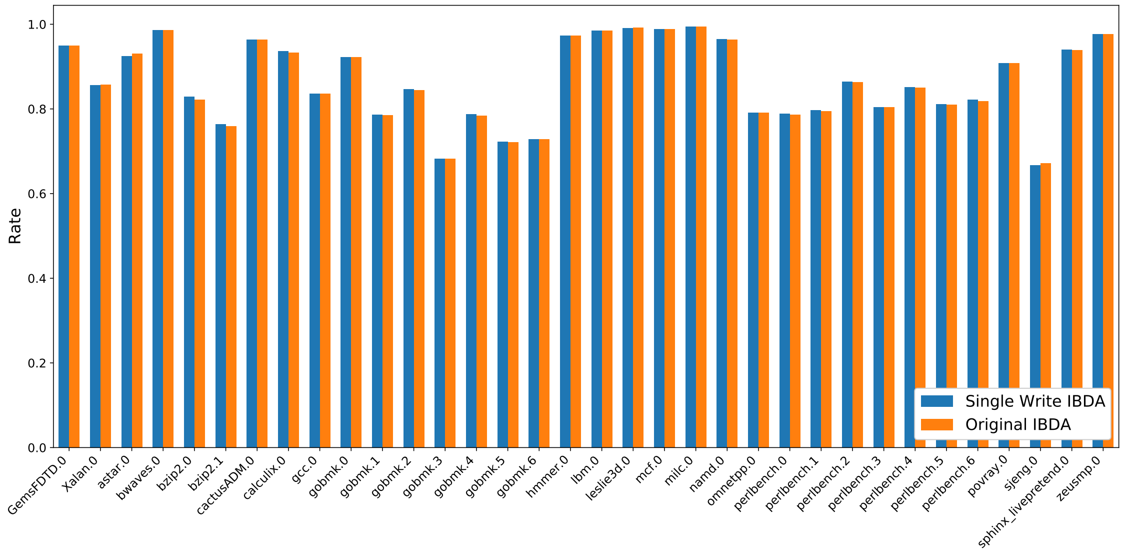
Fuzzy IBDA is an optimization that reduces the size of the instruction tag that is stored in both the IST and RDT, at the cost of introducing false positive hits in the IST. This section motivates why a fuzzy approach could be better and shows the implementation details of a Fuzzy IBDA using the instruction bits as input.

The Original IBDA uses the full PC of 40 bits to tag instructions. This provides a unique identifier for each instruction, however, it is unnecessarily big. Figure 5 shows the pseudo-entropy contained in each bit of the PC for a complete run of SPEC. Notice that only bit 1-17 have significant entropy.

Another observation is that Original IBDAs performance is mainly limited by the IST. Figure 6 shows the load slice extraction for Original IBDA and Fully Associative Original IBDA, an implementation with a fully associative cache as the IST. Both are normalized to Perfect IBDA. The difference in load slice extraction is entirely due to the premature evictions of IST entries. The delta between Original IBDA and Perfect IBDA is composed of both capacity cache misses, i.e. because 128 entries are not enough to store the working set, and conflict misses, when multiple instructions map to the



(a) Utilization of the four IST write ports for SPEC CPU2006



(b) Load Slice extraction normalized to Perfect IBDA for Single Write IBDA and Original IBDA

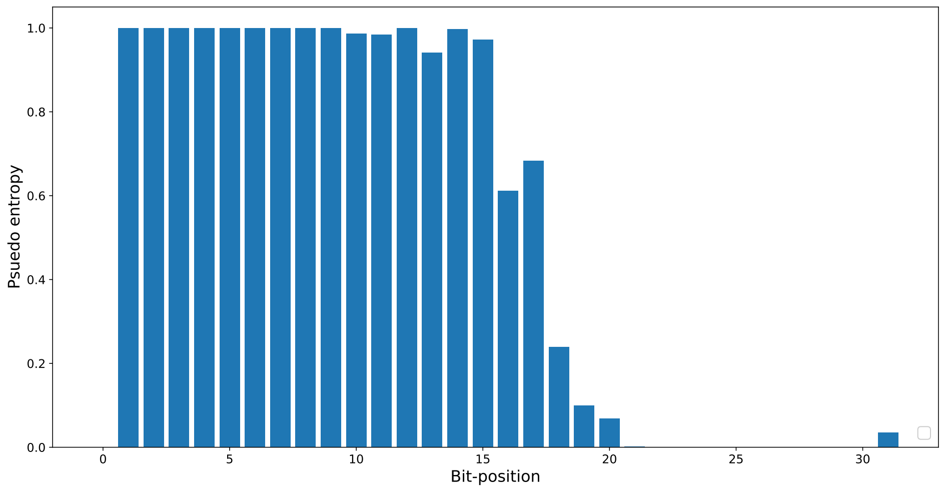


Figure 5: Average pseudo-entropy for each PC-bit running SPEC CPU2006

same set. The delta between Fully Associative Original IBDA and Perfect IBDA is only composed of capacity cache misses. This tells us that the main issue is the capacity misses. By reducing the instruction tags we can afford more entries in the IST and thus tackle the real issue of capacity misses.

There is also another BOOM-specific drawback of Original IBDA. BOOM does not forward the full PC beyond the frontend, where it is stored in the fetch target queue (FTQ). Only the 6 lower order bits, denoted *pc_lob*, which give the address of the instruction within the cache block, are bundled with the micro-op. To work around this the full PC could be bundled with the micro-op and passed through the fetch buffer, decode and rename. Another option would be to add additional read ports to the FTQ for the IST and RDT. A third option that we now will explore in depth is to use a combination of pc_lob and the instruction bits as a tag. The BOOM pipeline will be discussed in Section 7.2.4

Using the instruction bits as a tag introduces the possibility of false positive hits in the IST. This can happen if the program has two identical instructions located at two different addresses that share the *pc_lob*. A false positive hit in the IST will trigger the extraction of a false slice leading up to a false criterion. Any hits in the IST which would not have occurred with Perfect IBDA are denoted as a false positive, i.e. correct hits on *false slices* are also counted. False slices will likely lead to eviction of true slices and thus degrade performance further.

Using a hash of the PC bits is also an option for micro architectures that have the PC available in the backend. The following discussion also applies to the this.

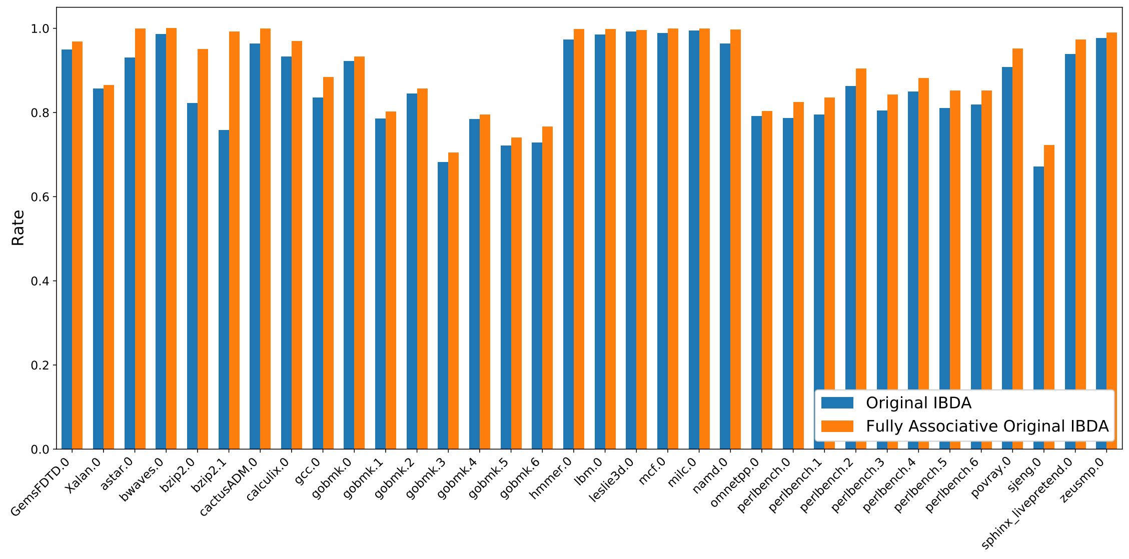


Figure 6: Load slice extraction normalized to Perfect IBDA for Original IBDA and Fully Associative implementation running SPEC CPU2006

3.4.1 Hash Functions

Our estimations show that for SPEC CPU2006 the instruction words contains around 11 bits for entropy. Combined with around 5 bits of entropy for *pc_lob* we get a total 16 bits. There is therefore no point in using full 38 bits of instruction word and pc_lob. A hash function is thus used to reduce number of bits in the tag.

3.4.2 Naive Hash

Our first approach to a hash function is just be picking out the bits shown to have the highest entropy. We call this Naive Hash. Its main weakness is that is purely optimized for SPEC workloads and its thus not very flexible for new types of programs. Naive hash serves as a baseline for other hash implementations.

3.4.3 Random Binary Matrix Hash

A more promising technique is that of a Random Binary Matrix Hash. It is inspired by the address mapping techniques using binary invertible matrices described by Jahre et al. [29].

For a hash function with n input bits and m output bits, a random matrix M with m rows and n columns is created. For an input x the hash $h(x)$ is simply the matrix multiplication $h(x) = Mx$ over the residue field of 2. Lets consider a simple example of a Random Binary Matrix Hash function that

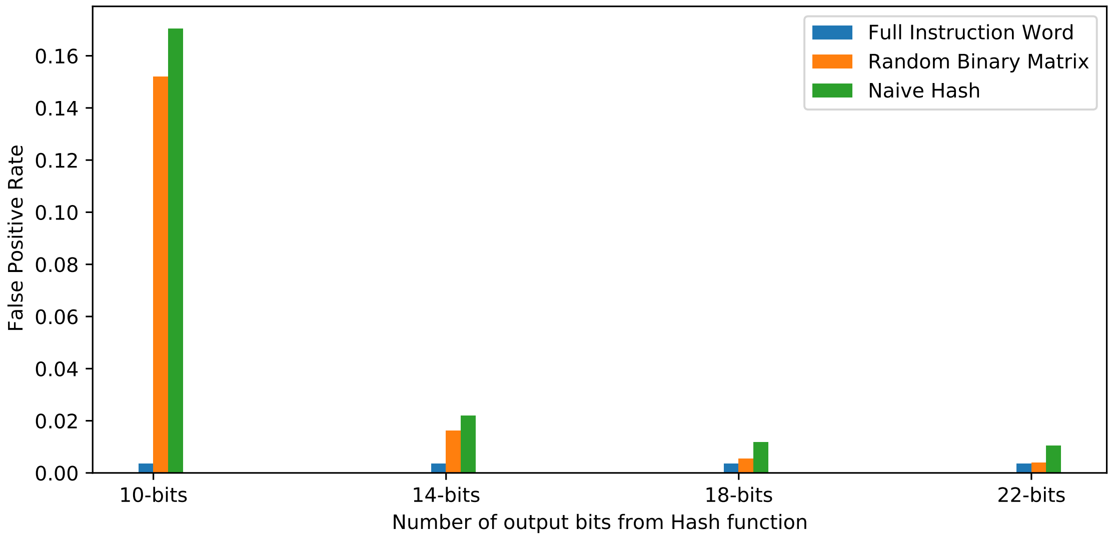


Figure 7: Average false positive rate for Fuzzy IBDA running SPEC CPU2006 with different hash functions

maps a 4-bit input to a 3-bit output. Our hash function is then

$$h(x) = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} m_{11}x_1 + m_{12}x_2 + m_{13}x_3 + m_{14}x_4 \\ m_{21}x_1 + m_{22}x_2 + m_{23}x_3 + m_{24}x_4 \\ m_{31}x_1 + m_{32}x_2 + m_{33}x_3 + m_{34}x_4 \end{bmatrix} \tag{3.1}$$

This can be implemented by XOR'ing together the columns of M corresponding to the set bits of x. The Random Binary Matrix Hash works by letting each input bit contribute to each output bit. A chained version of this type of hash was described by Augot et al. [30].

Figure 7 compares the false positive rate for Naive Hash and Random Binary Matrix Hash hashing *pc_lob* and the instruction word. The values reported are the average for a run through SPEC. The same Binary Matrix was used for all programs. The false positive rate for a design using the full instruction word and *pc_lob* is also added for comparison as "Full Instruction Word". By producing 14 bit hashes the Random Binary Matrix achieves close to the same performance as using the full Instruction Word and the *pc_lob* which is 38 bits in total. At 18 bits the performance is almost identical.

A problem we encountered was that there is some variance in the performance of different random matrices. We solved this by running multiple smaller benchmarks until we found a seed for our random number generator that gave satisfactory performance.

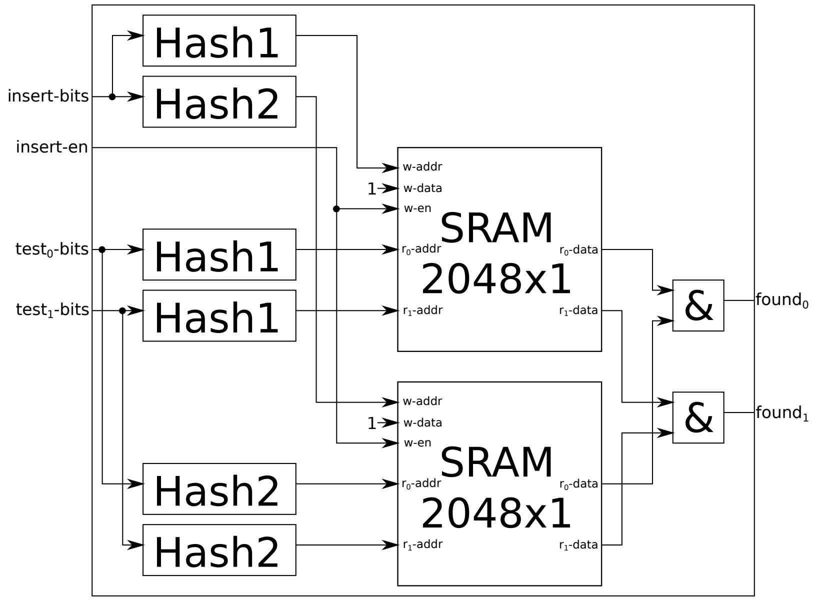


Figure 8: Partitioned Bloom filter with one insert and two membership test ports. Flush logic not shown.

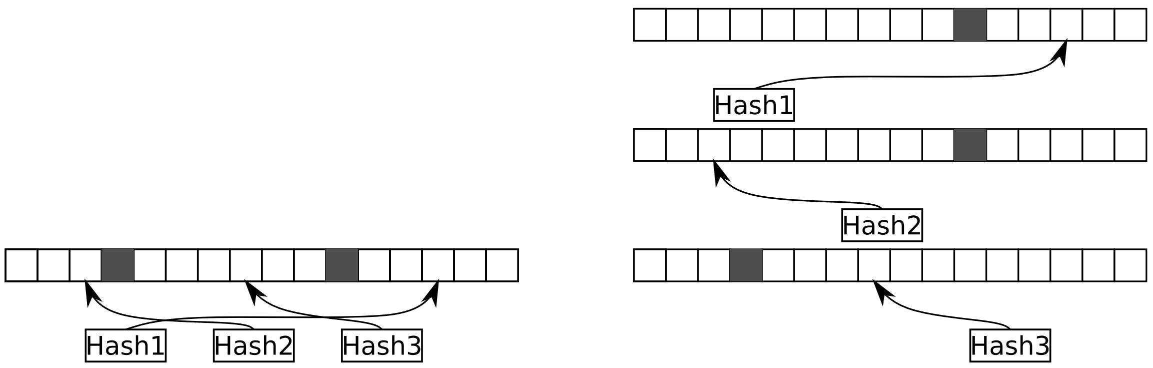
3.5 Bloom IBDA

The Bloom IBDA is a fuzzy implementation that uses a Bloom filter as the IST. A Bloom filter is an approximate set that can have false positives but not false negatives for membership tests [31]. It uses an array of m bits that is indexed by k different hash functions. When a value is added to the Bloom filter it is hashed using each of the hash functions, and the bits that the hashes correspond to are set to high in the array. To check if a value is in the Bloom filter it is again hashed. If the bits at the indices the hash point to are all high it is reported as part of the set. This can either be because it was added before or because there was a collision resulting in a false positive.

As more values get added to the Bloom filter, more bits get set, and the probability of a false positive rises. If n values have been added to the Bloom filter the probability of a false positive $p \approx \left(1 - e^{-kn/m}\right)^k$. This means that a Bloom filter can only be used with reasonable confidence until a certain number of values has been added. The maximum value of n, n_{max}, depends on m, k and the acceptable rate of false positives, p_a.

A conventional Bloom filter requires k writes to the bit array, for each addition to the filter, and k reads for each membership check. As SRAMs using multiple read and write ports are expensive our implementation deviates from this scheme by using a separate bit array for each hash function. This structure is called a Partitioned Bloom filter [32]. It uses k m-bit arrays with one write and two read ports, to support one addition and two membership checks per cycle.

A schematic for $k = 2$, $m = 2048$ is shown in Figure 8 The probability of false positives changes



(a) Bloom filter (b) Partitioned Bloom filter

Figure 9: Comparing a conventional Bloom filter to a Partitioned Bloom filter

to $p_{partitioned} \approx \left(1 - e^{-n/m}\right)^k$. An example of a Bloom filter membership test, with $k = 3$ and $m = 16$ is shown in Figure 9a. The gray entries represent the high bits from a previously added value. The arrows point to the hash function outputs of the membership test. This will result in a negative result.

The same example using the Partitioned Bloom filter is shown in Figure 9b. Notice that the conventional Bloom filter has a collision between *Hash1* and *Hash2* for the previously added value. The Partitioned Bloom filter avoids such collisions but requires k times more storage. This makes the conventional Bloom filter more attractive in software, where the target is to keep the memory footprint low, and the accesses for the different hash functions can happen sequentially. A hardware implementation of this would get prohibitively expensive for larger values of k. In hardware the Partitioned Bloom filter shines by requiring only memories with multiple read and one write port. Furthermore, the probability of a collision for given values of k and m is lower since collisions between different hash functions become irrelevant.

The disadvantage of Bloom filters is that there is no way of removing individual old and irrelevant entries. Because the probability of false positives grows as more values are added, the Bloom filter has to be flushed when n_{max} values have been added. In our current implementation it takes m cycles to flush the Bloom filter, so a high flush rate will degrade performance. If the SRAM arrays were partitioned into smaller banks this could be reduced by zeroing the banks in parallel. Figure 10 shows the flush rate for various implementations of Bloom IBDA. The flush rate reflects the n_{max} of that implementation and the load slice working set of the benchmark. Notice that for some benchmarks, like bzip, the load slice working set fits within Bloom filter for most of the benchmark, and the flush rate is close to zero.

Finally , Figure 11 compares the load slice extraction for Original IBDA, Single Write IBDA, Fuzzy IBDA and Bloom IBDA. Fuzzy IBDA uses the instruction word and the 6 least significant bits of the PC. Bloom IBDA uses 2048 bits, eight hash function and $p_{partitioned} = 0.01$ The results are normalized to Perfect IBDA which has the maximum achievable load slice extraction for any IBDA-

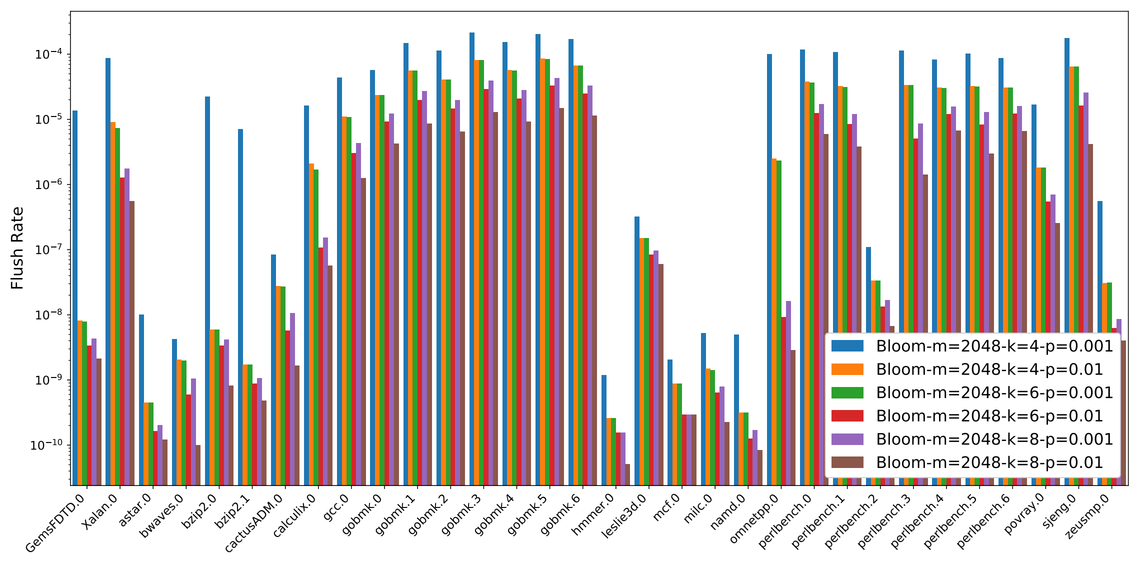


Figure 10: Flush rate for different implementations of Bloom IBDA running SPEC CPU2006

based design. Clearly the inaccurate approach of Fuzzy IBDA and Bloom IBDA gets a higher load slice extraction. This is because they are more likely to identify any instruction as part of a load slice, leading to false positives. Some of these false positive hits will actually be part of a load slice.

The missing piece of information is how expensive those additional false positives are. This can be investigated by doing an RTL implementation of the different IBDA designs and running real workloads on it.

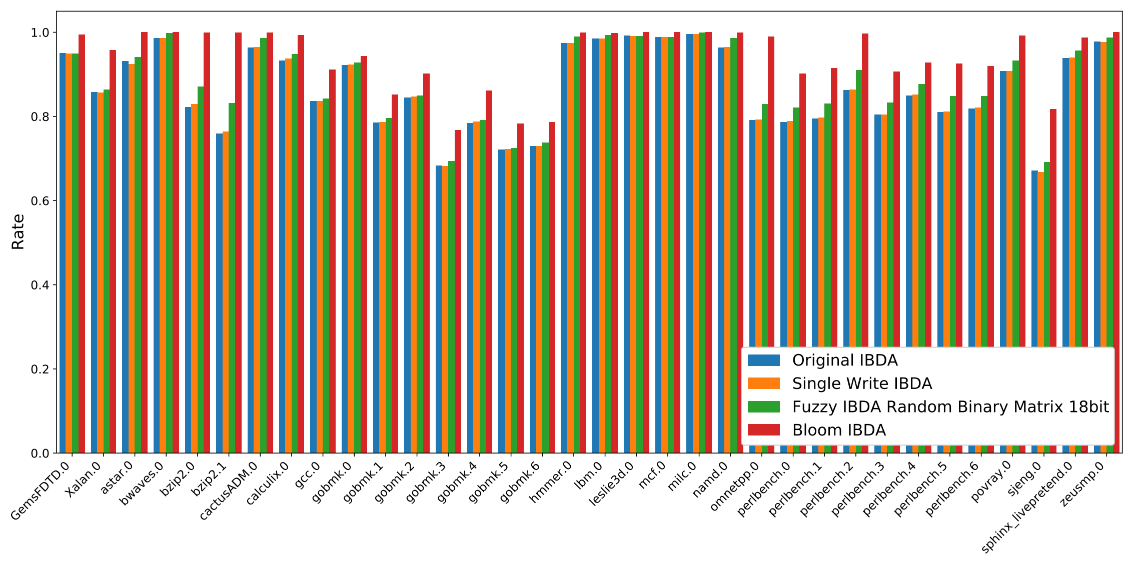


Figure 11: Comparing load slice extraction normalized to Perfect IBDA for Fuzzy, Baseline and Single Write IBDA running SPEC CPU2006

4 Methodology

4.1 Spike

To generate the false positive rates and load slice extraction rates reported in Chapter 3 we used Spike. Spike is an open-source RISC-V ISA simulator. It is a high-level behavioral simulator and does not model any low-level pipeline structures. This means it is not fitted to simulate the performance implications of false positive rate and load slice extraction rates. However, it was a great fit for rapid implementation and behavioral testing of the IBDA algorithm itself. Our fork of Spike is hosted on Github[1].

4.1.1 Pseudo Entropy

To calculate the entropy of a set X probability distribution $P(X)$ needs to be known. In many cases this is not available and heuristics have to be used instead. We wanted to calculate the entropy contained in each bit of the instruction word for RISC-V instruction set architecture (ISA). For an introduction to RISC-V please refer to Section 7.2.2. Using Spike we could simulate the whole benchmark and thus have access to the complete stream of instructions that constitutes these benchmarks. However, since it contains billions of instruction it would simply not be feasible to store the instruction word of each retired instruction. Instead, we use a 32x32 matrix of integers, E, to track how often each bit was high. When an instruction retires, all the 32 bits of its instruction word are looped over. When a high bit is found, all the integers in the corresponding matrix row, that match the high bits of the instruction word, are incremented. In other words, the diagonal of the matrix will contain the absolute number of times each instruction bit was found to be high. $E(8,8)$ has the count of how many times bit 8 of the instruction word was high. The other positions count how many times combinations two bits where high together. Thus $E(1,2)$ is how many times both bit 1 and bit 2 were high. This matrix is thus symmetric, as $E(i,j) = E(j,i)$.

E will thus contain information about the probability of each bit being high and the conditional probability of it being high. Note that we only have a "one-level" conditional probability, i.e. given a single other observation.

Using Equation 2.1 we can calculate the matrix H that contains the entropy for each bit on the diagonal and the conditional entropy outside.

Our algorithm for calculating the pseudo entropy of all the bits based on the matrix H is given in Listing 4.1. What we essentially do is in each iteration pick the bit with the highest average conditional entropy, with respect to each individual bit that is already picked. Thus, we order the bits from high entropy to low, and we can sum up the entropy.

This is pseudo-entropy because each iteration we should have computed the conditional entropy

[1] https://github.com/erlingrj/riscv-isa-sim/tree/ibda

based on all the bits in the $PICKED_BITS$ array. However, we only can calculate the conditional entropy with respect to one at the time.

Listing 4.1: Pseudo entropy algorithm

```
H = {32x32} # H is a 32x32 bit matrix containing entropy and conditional entropy of all bits
A = {1..32} # A contains all the bit positions
PICKED_BITS = {} # Contains our picked bit positions.
total_pseudo_entropy = 0

while len(A) > 0:
    if len(RES) == 0: # First iteration
        # Find element S with the highest "absolute entropy"
        S = find_max(diagonal(H))
        # Initialize the running total with that entropy
        total_pseudo_entropy += H(S,S)
        # Finally remove S from A and add to PICKED_BITS
        A.remove(S)
        PICKED_BITS += S
    else: # Rest of iterations
        # First we find the bit with the highest average conditional entropy
        # given the bits already present in PICKED_BITS
        S = find_max_avg_conditional_entropy(H,PICKED_BITS)
        # Then we add the average conditional entropy of that bit to the running total
        total_pseudo_entropy += get_avg_conditional_entropy(S,PICKED_BITS)
        # Lastly we remove the bit from A append it to the PICKED_BITS array
        A.remove(S)
        PICKED_BITS += S
```

4.2 Load Slice Core

We have implemented the different IBDA designs as part of our Load Slice Core implementation. It is a register-transfer level (RTL) design based on BOOM and explained in detail in Section 8.4. Our fork of BOOM which contains the Load Slice Core and all IBDA designs is hosted on Github[2].

4.3 FPGA-based Application Performance Evaluation

For testing and verification we used the cycle accurate software simulator Verilator [33]. However, to get real performance numbers of our IBDA implementations we have synthesized our designs, as part of LSC, to a Xilinx Zynq ZC706 FPGA. This allowed us to run parts of the test data set of the SPEC2006 benchmarks, with LSC running at 50 MHz. Refer to Section 9.2 for a more detailed discussion on running core emulations on an FPGA and Appendix A for a step-by-step guide.

When reporting averages we use the harmonic mean.

[2]https://github.com/EECS-NTNU/riscv-boom/tree/thesis-final

	Baseline	Single Write IBDA	Fuzzy IBDA	Bloom IBDA
IST Ways	2	2	2	-
IST Sets	64	64	64	-
RDT->IST Write ports	4	1	1	1
IBDA Tag Size	40	40	14	40
Bloom hash functions	-	-	-	6
Bloom collision rate	-	-	-	0.001
Bloom IST entries	-	-	-	2048
Seed for random hash matrices	-	-	1	1

Table 1: IBDA parameters

4.4 FPGA-based Area Analysis

To evaluate the area needed by each IBDA design we look at the resource utilization in the synthesized designs targeting FPGAs. Refer to Section 9.1.1 for a more detailed description of the resources available and the shortcomings of this method.

4.5 IBDA Parameters

Refer to Table 6 in Section 9.3 for the configuration parameters used in LSC for all the IBDA implementations. Table 1 shows the IBDA parameters used by the different different implementations.

5 Results & Discussion

In this section we will look at the results from synthesizing and simulating our RTL implementations of the various IBDA designs. First we will look at the performance when running SPEC CPU2006 on an FPGA prototype of the Load Slice Core with the different implementations of IBDA. Then we will take a detailed look into area and resource usage of the different implementations.

5.1 Application Performance

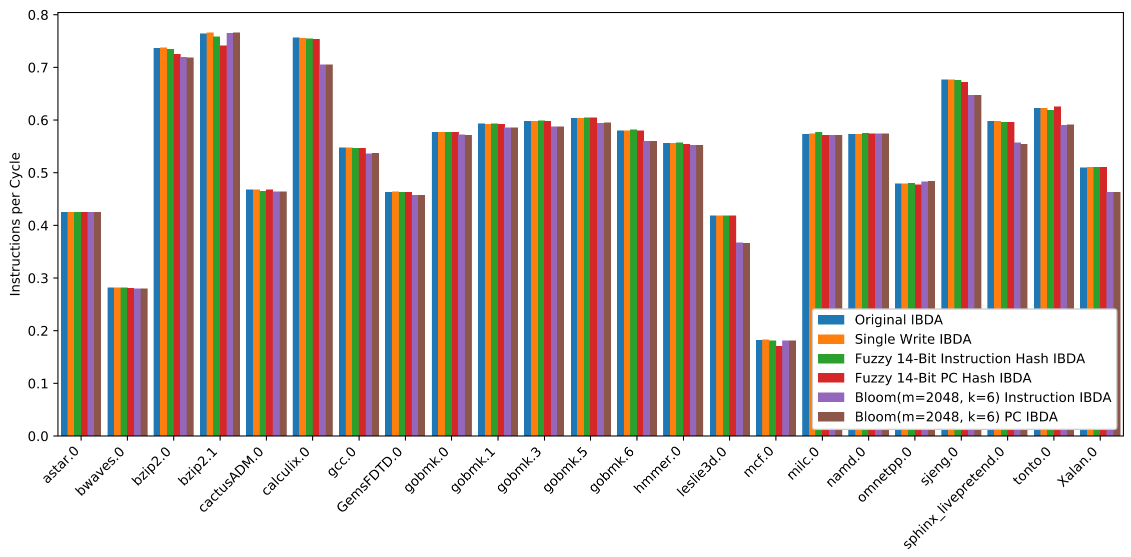
Figure 12a shows the instructions per cycle (IPC) for the different IBDA implementations normalized to the Original IBDA. Figure 12b shows the rate of instructions identified as part of load slices. Our first observation is that Single Write IBDA has on average 100.1 % of IPC and 99.8 % of the rate of instructions identified as load slices, compared to the Original IBDA. Single Write IBDA performs virtually identical to Original IBDA but at a significantly lower cost, as will be shown in Section 5.2 For Fuzzy IBDA we include both a configuration using the PC as input for the hash function and another using the instruction word and the six least significant bits of the PC. The instruction and PC version achieves on average 99.9 % and 98.9 % of the performance using Original IBDA. The *bzip* and *mcf* benchmarks are outliers where Fuzzy performs significantly worse. There will be some random matrices which performs worse than others, depending on the input data. We are unable to draw any conclusions regarding these outliers by looking at the load slice extraction rate.

For Bloom IBDA we observe varying performance and for *omnetpp* it actually outperforms Original IBDA. However, on average, the Bloom IBDA PC and instruction configurations gets 97.2 % and 97.3 % of the IPC of Original IBDA, respectively. The Bloom IBDA configurations both have a n_{max} of 778. This means that Bloom could work very well for programs with a working load slice set considerably less than Z if the false positives are not to frequent. With many false positives the Bloom IST will fill up and false program slices will be computed and added as well, which will further degrade performance.

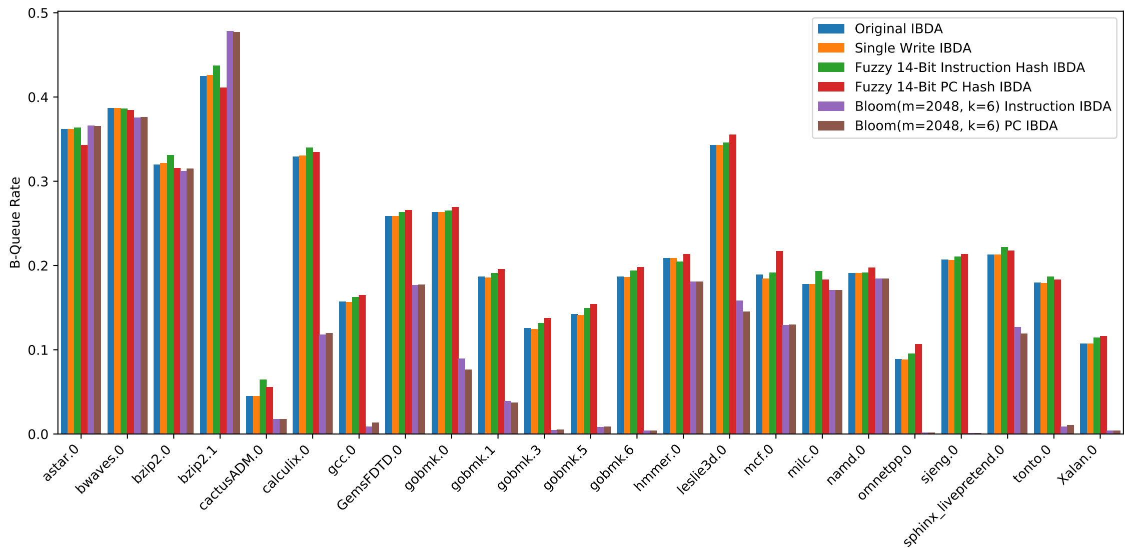
In conclusion, Single Write is unquestionably a good optimization. If a design can allow false positive hits in the IST, Fuzzy IBDA will save a lot of space at the cost of only a slight performance drop.

5.2 Implementation Sizes

The percentages reported here are comparing only the IBDA components. Single Write IBDA sees a 75 % reduction of BRAM usage, a 55 % reduction of LUT usage, and a 47 % reduction of flip-flop usage, compared to the Original IBDA. Fuzzy IBDA has a 91 % reduction in BRAM usage, 57 % reduction LUT usage and 67 % flip-flop reduction compared to Original IBDA. Bloom IBDA shows an 42 % reduction in BRAM usage, 54 % reduction in LUT usage and a 75 % reduction in flip flop



(a) IPC



(b) Load Slice Rate

Figure 12: Performance of the Load Slice Core using different IBDA implementations running SPEC CPU2006.

	Original IBDA		Single Write IBDA		Fuzzy IBDA		Bloom IBDA	
IST	2944	556	614	276	498	118	679	25
RDT	2096	354	1662	209	1717	184	1631	207
IFU	6928	4968	6928	4968	6926	4968	6926	4968
Total	115395	54187	112404	53770	109903	53585	112419	53461

Table 2: LUTs (left) and flip-flops (right) for different IBDA implementations

	Original IBDA		Single Write IBDA		Fuzzy IBDA		Bloom IBDA	
IST	0	32	0	8	80	0	0	24
RDT	0	16	0	4	0	4	0	4
IFU	532	19	532	19	484	19	484	19
Total	1056	82	1056	46	1088	36	1008	50

Table 3: LUTRAM usage (left) and BRAM usage (right) for different IBDA implementations

usage compared to Original IBDA.

The rest of this section provides in-depth discussion and analysis of the area estimates summarized above. Table 2 and Table 3 report the resource usage of different IBDA implementation. The BRAM usage reported in Table 3 are the combined usage of RAMB36 and RAMB18.

5.2.1 Single Write IBDA

Starting with Table 3 we observe that Single Write IBDA only uses a quarter of the BRAMs for RDT and IST compared to the Original IBDA. This is as expected, as our implementation of multi-port SRAM duplicates the whole SRAM for each added write port and for each read port. The IST of Single Write IBDA uses eight BRAMs, that is two for each read port of each way of the IST. This is compared to the 32 BRAMs used for Original IBDA IST. A way has 64 entries of 40 bits for a total of 2560 bits. Each BRAM block can hold 18 Kbits. This gives a utilization of only 7,1% of each individual BRAM block. From Table 2 we observe that the IST of Single Write IBDA sees a reduction of 2330 LUTs and 280 flip-flops compared to Original IBDA. 256 of the flip-flops are saved in the LVTs which no longer must store 2 bits per entry. The LUT reduction is also due to the decrease of the number of BRAMs, which reduces the need for glue logic and number of compare operations for lookups.

Single Write IBDA's RDT uses four BRAMs, two for each duplicate RDT due to the two write ports. Each RDT has 80 entries of 40 bits which gives us a utilization of 8.9 % of the BRAM. The Original IBDA RDT has 16 BRAMs with the same utilization. The Single Write IBDA RDT shows a reduction of 434 LUTs and 145 flip-flops. Of the flip-flops 124 are saved in the LVT. The LUTs are saved by reducing the number of duplicate SRAMs and the needed glue logic.

Clearly, a reduction in the number of write-ports from RDT to IST leads to massive improvement in the area used by IBDA. Furthermore, power consumption will also decrease accordingly.

5.2.2 Going Fuzzy

Compared to Single Write IBDA, the instruction tags are reduced from 40 bits to 14 bits, and the additional FTQ ports are removed. However, looking at Table 3 we see that Fuzzy IBDA still uses 4 BRAMs for the RDT, which is the same as Single Write IBDA. What is not shown is that the utilization of each BRAM should be more than halved. For the IST, the synthesis tool has chosen to use LUTRAMs instead of BRAMs. This is because the utilization of each BRAM would have been so low. In total Fuzzy IBDA will use only 35 % of the BRAM that Single Write IBDA uses, which gives a total reduction of 91 % compared to Original IBDA. Fuzzy IST shows a reduction of 2446 LUTs and 438 flip-flops compared to Original IBDA. 128 of the flip-flops are saved because the valid bits can be removed, as it is already fuzzy. However, Fuzzy IST needs additional 28 flip-flops to implement synchronous reads from the LUTRAMs.

The reduction in LUTs is likely stemming from reduced width of the combinational operations. E.g. to do an IST look-up a compare operation between the look-up value, and the value found in the IST, has to be done. Such a comparison could be implemented as a tree of XOR gates, each implemented as a LUT. LUTs have at maximum six inputs, which means that each LUT can compare 3 bits of the value. Going from 40 bits to 14 bits would reduce the a straightforward compare implementation from 20 LUTs to 6 LUTs. The binary hash matrix is implemented as 125 LUTs.

For the RDT, Fuzzy IBDA sees a reduction of 70 LUTs and 25 flip-flops. The RDT contains two bypass register, so we would actually expect a twice as big reduction in flip-flops. The RDT also contains a copy of the binary hash matrix using 125 LUTs.

Fuzzy IBDA also sees a reduction of 48 LUTRAMs in the instruction fetch unit (IFU), compared to Original IBDA and Single Write IBDA. This is because Fuzzy IBDA uses the instruction bits rather than the PC. Original IBDA needs an additional read port to the FTQ which causes a duplication in the LUTRAMs used for it.

To conclude, Fuzzy IBDA contributes to a significant reduction in all structures used.

5.2.3 Bloom IBDA

The Bloom IBDA represents an alternative fuzzy implementation. The main benefit being that the set associative cache can be replaced with a much simpler bit vector. Table 3 shows that the Bloom IST uses 24 BRAMs. It uses 6 hash functions each with a 2048-entry bit-vector. Since we have 2 read ports, one for each core-width, we need to duplicate that. In total Bloom IST stores 24576 bits, which is about a 40 % decrease compared to Original IBDA. Bloom IST has a reduction of 526 flip-flops compared to Original IST. This is largely due to the 64 flip-flops needed for the LRU and 128 flip-flops used for valid bits. In total Bloom IST needs 65 more LUTS than Single Write IBDA. The Bloom RDT stores the 38 bit input to its hash function, consisting of the instruction word and pc_lob. This makes the BRAM, LUT and flip-flop usage of Bloom RDT more or less equal that of Single Write IBDA.

Bloom IBDA sees the same reduction in LUTRAMs used in the IFU as Fuzzy IBDA. This is also due to the removal of a read port to the FTQ.

In conclusion, for applications where the working set can be sufficiently large, using a Bloom filter gets too costly compared to a set associative cache.

5.3 Precise IBDA

This section will discuss some scenarios where the Original IBDA is imprecise, i.e. can get false positive hits in the IST. A Precise IBDA is an implementation where no instructions are mistakenly identified as part of a program slice when they are not.

First we will look at what could happen after an address space switch, before we will look at how speculation can affect IBDA. We also discuss a potential solution to this problem, namely moving the RDT.

5.3.1 Address Space Switches

If a processor uses virtual memory, two different processes can use the same virtual address range for their instructions. The physical address-range is different and translated by the translation lookaside buffer. If the IST is kept unchanged for an address space switch, the new process will potentially get false positives when looking up the addresses of its instructions and finding hits from completely different instructions using the same address, but added by a different process. In order to maintain correctness of IBDA we propose to flush the IST on context switches. This can be done by clearing the valid bits implemented as registers. The RDT typically does not require any flushing. If the register file (RF) is restored to the state of the previous time the process was executing by loading values from the stack, the RDT will be updated to reflect the new dependencies, which will all originate from the loads that restored the RF.

5.3.2 Speculative Execution

Speculative execution can lead to the execution of unintended instruction sequences. Listing 5.1 shows a sequence of instructions that could cause this. The branch is always taken, so that the load depends on `li a5, 5`. If it is speculated as not being taken, a false dependency from the load to `li a5, 10` will appear. This dependency is false in the sense that it never appears when executing the program. It could, however, also be marked as a dependency when static analysis tools are used.

Listing 5.1: Example of a program that could add instructions to the slice due to speculative execution

```
start:
    li a0, 10
    li a5, 10
    beq a0, a5, branch
load:
    lw a4, 0(a5)
    j end
branch:
    li a5, 5
    j load
end:
```

To solve the issue of speculative IBDA, we must either roll back the state of the IST and RDT on a misspeculation, or we must hold off updating the RDT and IST until instructions are no longer speculative. Doing a rollback would be very costly and could require keeping duplicate ISTs and RDTs for checkpoints. On speculation, a new duplicate of the IST and RDT would be used. On a misspeculation a single cycle reset could be done, by starting off from the appropriate check point.

5.3.3 Moving the RDT

Avoiding speculation altogether seems like a more attractive solution. This is easily achieved by moving the RDT lookup to the commit stage. Instructions only reach the commit stage once its guaranteed that they are in fact part of the program. Since there is no speculation, the logical register specifiers, of which there are only 32 in RISC-V, can be used. The main drawback is that there are potentially many cycles between the IST lookup and the RDT lookup. This can lead to duplicates in the IST in tight loops. Consider the first iteration through a tight loop with a load slice. None of the instructions will be in the IST and thus not be marked as part of the slice. If multiple iterations through the tight loop can fit in the instruction window there will be multiple copies of the same instruction, none marked as part of the slice. When the first instructions reach commit, they will update the RDT and be added to the IST. However, when the next iteration of the same instructions arrive they will also be marked as not present in the IBDA. So for Single Write IBDA to be compatible with moving the RDT it must implement a read-before-write policy in the IST.

6 Conclusion

In this part we have presented the first, to our knowledge, RTL implementation and in-depth evaluation of IBDA in terms of size and application performance. We also include three different optimizations for IBDA. Single Write IBDA, Fuzzy IBDA and Bloom IBDA. Our evaluation shows that Fuzzy IBDA will only use a fraction of the original area, while delivering almost the same performance. While Bloom IBDA is an interesting concept, it suffers from poor performance in some cases and has no size-benefit over Fuzzy IBDA.

Part II

FIFO-based Instruction Scheduling

7 Background

This chapter will introduce the fundamental concepts needed for creating register-transfer level (RTL) implementations of novel instruction scheduling techniques on a modern out-of-order core. We start by laying the theoretical framework for instruction scheduling by looking at in-order and out-of-order instruction scheduling. The additional modules needed to support out-of-order scheduling, e.g. the reorder buffer and Register Renaming, are also introduced. Then we look at the recent computer architecture research tools enabling the RTL implementation of these techniques. We introduce the RISC-V instruction set architecture (ISA), the hardware construction language Chisel, BOOM and Rocket, two cores written in Chisel, and the Chipyard system-on-chip (SoC) framework. Lastly, we introduce three novel instruction scheduling techniques found in the Load Slice Core, Delay and Bypass, and CASINO.

7.1 Instruction Scheduling

Instruction scheduling is one of the key techniques for extracting instruction-level parallelism (ILP) and memory-level parallelism (MLP) from a program. Since program binaries are expected to run on different microarchitectures sharing the same ISA, dynamic hardware scheduling is used in addition to static software scheduling. ILP is a measure of how many instructions of a program could be executed in parallel because they are independent. It is a measure of how many memory operations, i.e. loads and stores, can overlap. MLP is an important subset of ILP, since memory latency is one of the key limitations of computer performance [34].

7.1.1 In-Order Scheduling

In-order scheduling is the simplest form of hardware scheduling and is also called static scheduling. A static scheduler does not rearrange the order of the instructions as specified in the binary. The programmer, or compiler, has complete control over the order in which the instructions propagate through the pipeline. Many hazards can be avoided if the details of the target pipeline architecture are known at compile-time. The main advantages of static instruction scheduling is its low hardware complexity.

The simplest in-order processors use static instruction scheduling. These kinds of processors have to wait until an instruction finishes execution for the next instruction to execute. If, for example, a cache miss happens, and a load has to go to the main memory, the whole pipeline stalls and no other instructions can execute. More advanced stall on use in-order processors provide higher performance while still being mostly in order. They achieve this by allowing the execution phases of instructions to overlap. In the example above other independent instructions can be issued by such a processor. A method that is commonly used for this is called scoreboarding. A scoreboard keeps track of instruction dependencies and issues instructions in-order, as soon as their dependencies are met. It also ensures that instructions retire in the right order, even if their execution phases might end out of order. Furthermore, this approach allows issuing instructions to multiple execution units at the same time, leading to super scalar designs [35]. The pipeline has to stall if an instruction depends on another instruction that has not finished execution or targets a non-pipelined execution unit currently in use. Another source of conflicts are write-after-write dependencies between instructions with the same target register.

7.1.2 Out-of-Order Scheduling

Software portability between microprocessors is an important property, and compiled libraries should be able to link and run efficiently on different architectures. That makes it very hard for the compiler to anticipate all hazards when scheduling instructions. There are also many dependencies and stalls which cannot be known at compile time due to things like cache misses, memory references or data-dependent branches. Furthermore, branches and jumps make statically predicting all possible code paths quite hard. Out-of-order scheduling allows the processor to rearrange the order of execution of the instructions at run-time, as long as it is not changing the semantics of the program. This is a very flexible way of extracting ILP and MLP. However, maintaining the semantics of the program introduces more complexity. In the following sections we will describe the main components needed to realize full out-of-order scheduling.

Issue Unit

The issue unit is one of the main components of out-of-order scheduling, and accounts for 18-40 % of the total power consumption of the core [36]. Instructions are added to the issue unit in program order, and are issued to the functional units out-of-order. The issue unit is typically centered around a content addressable instruction queue (IQ), this is sometimes referred to as the issue queue. The entries of the IQ are often called issue slots. The IQ holds all the instructions while they wait for their operands to get ready. A bypass-network broadcasts the results of instructions to the issue slots, so that dependent instructions can issue. The issue select logic evaluates the readiness of all issue slots, and decides which to issue based on some priority scheme.

There are three main parameters for the issue unit. The issue *width* is the maximum number of instruction that can be issued in a cycle. This is equivalent to the number of issue ports. The *depth* of the issue unit describes the number of instructions the IQ can hold, which in turn means how many candidates there are for each issue port. The complexity of the bypass-network and the issue select logic grow super-linearly with both the width and the depth of the issue unit [8], [37]. The final parameter is the priority scheme. There are essentially three types of priority schemes, achieved by different IQ organizations [17]. A *age-ordered shifting queue* is the most obvious and used in older processors. It keeps the IQ as an age-ordered queue and can thus simply assign priority based on the queue position. This works well, because instruction criticality is highly correlated with the age of the instruction. However, the compacting logic that shifts the queue to fill the holes left by issues is very expensive. A cheap and simple approach is using a *random queue*. Instead of shifting the queue to fill empty holes, a random queue just adds new entries to the holes. This causes a random order in the queue, and the issue priority is random. The most common approach is using a random queue together with an *age matrix* that tracks the age of the instruction in the random queue [38].

Register Renaming

Write-after-write and write-after-read hazards can limit out-of-order instruction scheduling greatly. If a subsequent instruction writes to the source register of a prior instruction, there is a write-after-read hazard. If a subsequent instruction write to the same destination register as a prior instruction, there is a write-after-write hazard. In both cases the latter instruction cannot execute first, because that would change the semantics of the program. They are called false dependencies, as they are not data dependencies but rather caused by reuse of the limited number of logical registers in the ISA. Register renaming is a technique to eliminate such false dependencies between instructions. It does so by mapping the logical register specifiers of the instructions into physical register specifiers which point to actual registers in the core. These mappings are stored in a map table. There are more physical registers than logical registers. This allows us to rename the destination registers of all instructions that write a result to the register file. This eliminates false dependencies, as each instruction will write to a unique register. It also acts as a result buffer for speculatively executed instructions. When these instructions no longer are speculative and can commit, the map table is updated, and the results are exposed. The logical to physical mappings have to also be saved for branches, since changes to the register file that were caused by instructions that come after the branch have to be rolled back when a misprediction is detected. [35]

Reorder Buffer

To maintain the semantics of a program, while executing it out-of-order, it is necessary to keep track of the original program order of the instructions. A reorder buffer (ROB) is often tasked with this [35]. After decode, all instructions are added to the tail of the ROB, in-order, and marked as busy. When they finish execution they are marked as not-busy. When the head of the ROB is not-busy, it can be committed to make its architectural changes visible. From the point of view of the programmer, the instruction has not been executed until it is committed. In case of an exception, the processor will wait until the excepting instruction is at the head of the ROB, then it will flush the rest of the pipeline and redirect the instruction fetch to the appropriate exception handler [28]. In case of an exception, the map table must be restored to the state it had, when it renamed the excepting instruction. This can be accomplished by rewinding the remaining ROB entries, when the excepting instruction is at the head of the ROB.

Memory Disambiguation

All ISAs include a memory model which the core has to abide to. RISC-V implements a relaxed memory model where loads are allowed to fire to memory out-of-order with respect to other loads and stores as long as they are target addresses [39]. Since stores will only fire to memory at commit, and thus in program order, there are two types of memory ordering failures. Load-store memory ordering failures occur, when a younger load reads memory before an older store to the same address writes it [40]. In that case the load will return stale data. A load-load memory ordering failure occurs when a younger load reads memory before an older load to the same address. This give rise to subtle synchronization errors if another processor is writing to the address in between the two loads [40]. To avoid potential memory ordering failures, one could stall loads until all older stores and loads have their addresses resolved and then verify that there are no older outstanding loads or stores to the same address before firing the load [1], [9]. This has significant performance implications because it limits the MLP extraction [10]. The approach taken by out-of-order cores is to speculate on the memory ordering. It is typically achieved by speculating that new loads have no conflicts with any unresolved load or store and thus firing it to memory [41]. The speculatively fired loads are kept in a CAM called the Load Queue and when the unresolved loads and store do resolve, they search the Load Queue for any conflict.

7.2 Computer Architecture Research Tools

In this section we discuss state-of-the-art computer architecture research tools that enable our RTL implementation and simulation of the instruction scheduling techniques. We will start by looking at traditional high-level architectural simulators before we look into a new generation of tools and frameworks originating at UC Berkeley.

7.2.1 Architectural Simulators

High-level architectural simulators are the most common tool for computer architecture researchers to evaluate new ideas. They provide a fast and flexible way of design-space exploration. The gem5 simulation framework is particularly popular. It supports a variety of different ISAs, memory system models,network models as well as user-level and privilege-level execution [42]. Gem5 is a modular discrete event driven full-system simulator, and achieves good accuracy for most benchmarks [43]. Sniper is another simulator aimed at simulating multi-threaded shared-memory application running on 10s to 100s of cores [44]. It is based on interval simulation which makes it fast but still accurate [45].

High-level architectural simulators have some key weaknesses. The simulation speed is directly correlated to the accuracy of the simulator, so either they are accurate but slow, or they are faster and inaccurate. More importantly, area and power estimates are either inaccurate or not available [28].

Implementing research ideas at the register transfer level (RTL) would solve these problems. An RTL implementation can be synthesized to target an FPGA and thus give cycle accurate simulations at clock frequencies close to 100 MHz [41] It can also be taped out targeting a silicon process technology, for accurate area and power measurements. However, RTL implementations are more complex than high-level architectural models and there have not been any available RTL implementations of an open-source out-of-order core available until recently [28].

7.2.2 RISC-V Instruction Set Architecture

RISC-V is an open-source ISA developed at UC Berkeley [39]. An ISA is an abstract model of how a processor works, i.e. what instructions it supports, their encoding, the memory model etc. An open-source ISA is one anyone can use, contribute to or extend. This is contrasted by proprietary ISAs

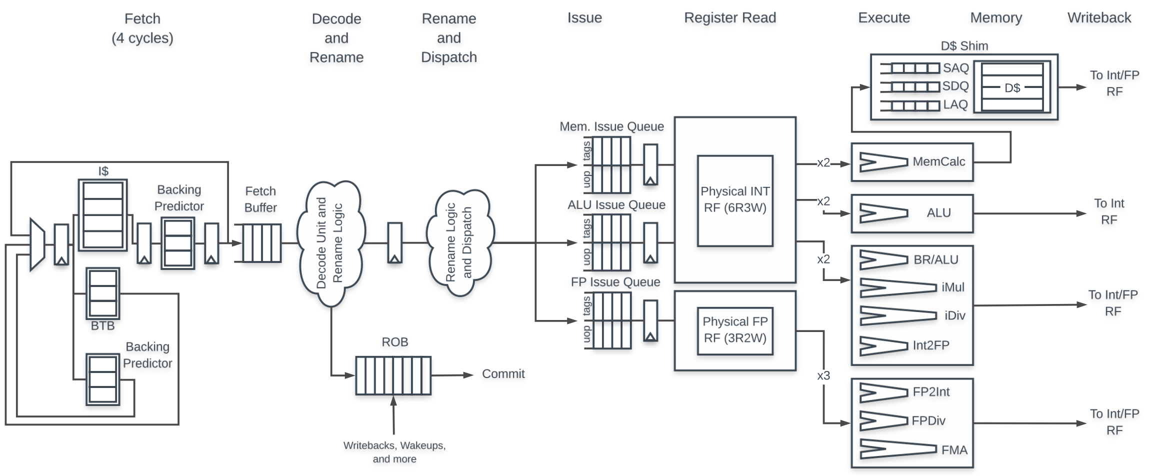


Figure 13: A high-level view of BOOMs pipeline. Adapted from [2]

like x86 or ARMv8 where a license is needed to implement or modify it [46]. Software interfaces are today full of open standards and open-source software implementing those standards. E.g. in operating systems, networking, compilers, databases and graphics [4]. RISC-V is an effort to bring this level of openness to hardware. It is a load-store architecture implementing a relaxed memory model. Not all implementations of an open instruction set have to be open but freely usable open-source implementations are enabled by it.

7.2.3 Chisel

Chisel is a hardware construction language developed at UC Berkeley. It is designed to enable rapid hardware design and prototyping using highly parameterized generators [12]. Chisel is embedded in the Scala programming language and has access to high-level concepts such as object-orientation, functional programming, parameterized types and type inference. Chisel is compiled to FIRRTL (Flexible Intermediate Representation for RTL) [47], an intermediate representation that is used for optimization. FIRRTL can generate a cycle-accurate software simulation of the design using Verilator [33], or generate low-level Verilog targeting FPGAs and ASICs.

7.2.4 The Berkeley Out-of-Order Machine

The Berkeley Out-of-Order Machine (BOOM) is an open-source out-of-order RISC-V generator written in Chisel [41]. The term *generator* is used to indicate that BOOM is not one single processor, but rather a framework which can instantiate different out-of-order processors. This is enabled by the powerful language features of Chisel. BOOM is the first open-source high-performance out-of-order processor available to academia [28]. It is a very useful tool for getting cycle accurate performance numbers as well as area and power estimates for research ideas using an out-of-order core.

The rest of this section will briefly introduce the BOOM pipeline. BOOM is conceptually broken up into 10 stages as shown in Figure 13.

Fetch. The fetch stages are part of the in-order frontend of BOOM. In parallel, a fetch packet is retrieved from Level 1 Instruction Cache (L1-I), and branch prediction predicts the next PC. BOOM supports super-scalar fetch, so fetch packets can contain several instructions. Fetch Packets are placed in a fetch buffer that decouples the fetch stage from decode, rename and the backend.

Decode. The decode stage takes the 64 bit wide instruction word and decodes it by using a lookup-table. BOOM introduces the concept of a *micro-op*, which is a bundle of all the control signals needed by the instruction in the backend. In this report we use the term "micro-op" to refer to an instruction and its control signals, this is often used interchangeably with the "instruction".

Rename. BOOM is a physical register file out-of-order design, i.e. it distinguishes between the physical register file (PRF) and the architectural register file (ARF). The ARF is an abstract entity specified by the ISA and instructions encode architectural registers. BOOM has a PRF with many more registers than the ARF. The mapping between the ARF and the PRF is dynamic and maintained in a map table. Speculative instructions executed out-of-order write to registers that are not mapped to the ARF, and thus they don't change the architectural state of the processor. The PRF thus contains both the architectural state and the speculative state. The speculative state is made visible when the instructions commit, and the map table updates the mapping from ARF to PRF. During the rename stage, the architectural register specifiers are translated to physical register specifiers. The destination register specifiers are mapped to an unused physical register to avoid write-after-write and write-after-read hazards.

Dispatch. During the Dispatch stage, the instructions are entered into the ROB, as well as the issue queues. The ROB keeps track of all the in-flight instructions in program-order. The instructions can only commit or throw an exception from the head of the ROB. The ROB notifies the Map Table of commits or rollbacks.

Issue. The issue stage contains the issue queues where dispatched instructions wait for their operands and available execution units. There are three separate issue queues.

Register Read. During this stage the instructions access the appropriate register file to get their operands.

Execute. The execute stage handles the execution of the instructions, and the writeback to the register file. To avoid having results of one instruction passing through the register file to a subsequent dependent instruction, BOOM supports Forwarding.

Memory. The memory stage spans several actual stages. Loads and stores will have entries allocated in the load queues (LDQ) and store queues (STQ) of the load store unit (LSU) during the dispatch stage. Both stores and loads needs to pass through the execute stage, to do address calculation, before they can be fired to the L1-D. BOOM implements a relaxed memory model, which means that loads are allowed to execute out-of-order, with respect to other stores and loads, as long as they are to different addresses. Stores are only written to memory after the instruction has committed from the head of the ROB, i.e. in program order. Loads whose address arrives from the ALU in the current clock cycle have priority for the L1-D port. They will fire to L1-D even before it has been checked, whether there is a store or another load in the queue with the same address. The following clock cycle it will check for conflicts. In case of a conflict with another load the request must be killed, and the load is put to sleep. It will be woken up later to be retried. In the case of an address conflict with a store, the request must also be killed, since the value in the L1-D is stale. The correct value will be in the STQ. This value is forwarded to the load, and it can return without stalls.

Writeback. During writeback the results of the execute stage and memory stage are written back to the register file.

Commit The Commit stage hides the out-of-order backend from the world. When an instruction commits, the architectural changes it has caused are made visible. For instructions that write registers, this is done by updating the mapping between the ARF and the PRF in the Map Table. Stores are allowed to update the L1-D only after they have committed. Exceptions are only handled when the excepting instruction reaches the head of the ROB. The pipeline is then flushed, and the ROB emptied.

7.2.5 Rocket

Rocket [14] is a scalar in-order core with five pipeline stages. BOOM reuses many parts of rocket, and they can be used in combination to form a heterogeneous SoC similar to ARMs big.LITTLE [48] architecture. We use Rocket as a baseline to evaluate the overhead of out-of-order over in-order cores.

7.2.6 Chipyard SoC Generator

To simulate a core like BOOM you also need other components, such as the memory subsystem and uncore [28]. Chipyard is an integrated design, simulation and implementation framework for custom SoCs [13], which provide the missing pieces. It is also developed at UC Berkeley and supports BOOM out of the box. Chipyard contains many separately developed, highly parameterizable IP-blocks for building a complete SoC. It has in-order and out-of-order RISC-V processors, caches, communication peripherals, accelerators, network interfaces and more. A SoC design can be compiled into a cycle accurate software simulator using Verilator, synthesized to an FPGA-accelerated simulation using FireSim or targeted ASICs using Hammer [49].

7.3 Novel Scheduling Techniques

In the last section of this chapter we introduce three novel microarchitectures. The focus of this report is the instruction scheduling technique applied by each of them. They represent different design points between the two extremes of in-order scheduling and out-of-order scheduling as described in Section 7.1.

7.3.1 Load Slice Core

The Load Slice Core (LSC) is based on the insight that a design which only extracts MLP can achieve almost the same performance as a full out-of-order design that extracts MLP and additional ILP [1]. Simulations by Carlson et al. show that around 90 % of the performance gain of out-of-order over in-order can be achieved by just doing loads and *address generating instructions* (AGI) out-of-order. Such an architecture would still need a complex issue unit and wouldn't save much power over an out-of-order core. The real takeaway is that letting loads and AGIs issue out-of-order, but in-order with respect to each other, almost achieves the same performance as letting them issue completely out-of-order. That is because the AGIs leading up to a load are typically data-dependent on each other, and thus cannot execute out-of-order. LSC is built on an in-order stall-on-use architecture, extended to allow early execution of load slices. Load slices are identified using IBDA, and dispatched to a separate instruction queue, the *Bypass Queue* (BQ). All other instructions are placed in the AQ. As suggested in Figure 2, there is a need for several new structures to allow for the early execution of load slices. Register renaming is needed to avoid anti-dependencies, and to provide a structure to store the speculative results from the OoO execution of the BQ. An issue unit is needed, but it only has to consider the heads of the A- and B-IQ. To keep track of the program order, scoreboarding is proposed, and an extra commit stage is needed. Store instructions are split, such that the address generating part is put on the B-IQ, and the data-generating part is put in the AQ. This way the costly load queue, that serve as a way of detecting memory ordering failures, can be avoided.

The performance of LSC is simulated and compared to in-order (ARM Cortex A7) and out-of-order (ARM Cortex A9) baselines. Using the SPEC2006 benchmark, on average, LSC reaches 74 % of the total performance gain achieved by out-of-order over in-order. The out-of-order has a 158 % area overhead over the in-order while the LSC only has 15 %. The power consumption of out-of-order is estimated to have an overhead of 35 % with only 22 % for LSC. It is concluded that the LSC achieves an area-normalized performance of 2009 MIPS/mm2 and an energy-efficiency of 4053 MIPS/W, contrasted with 1508 MIPS/mm2 and 2852 MIPS/W for an in-order. Out-of-order reaches only 1052 MIPS/mm2 and 862 MIPS/W.

7.3.2 Delay and Bypass

The Delay and Bypass (DnB) microarchitecture takes the concepts of LSC and moves them further towards out-of-order. A weakness of LSC is that independent load slices can be stalled behind each other in the B-IQ, which will limit the extraction of MLP. DnB also takes inspiration from Long Term Parking (LTP) [37] and the Front-end Execution Architecture (FXA) [50]. LTP seeks to reduce the IQ depth by locating non-critical instructions, delaying them in a FIFO until they become urgent and only then adding them to the IQ. FXA, on the other hand, lets instructions, that are ready at dispatch, issue right away. The goal of both is to reduce the width of the IQ. DnB combines and improves these two ideas.

DnB starts out from an out-of-order core and tries to remove or reduce the most power hungry structures whilst still keeping the ability to extract ILP. DnB argues that the issue queue (IQ) is the single most power hungry component for an out-of-order, accounting for 18-40% of the total core power [8]. This stems from both the bypass network, and the issue select logic. DnB proposes to use simple FIFOs for instructions that don't require the power of the IQ. It introduces the concept of criticality and urgency. An instruction is critical if it belongs to a load slice. DnB uses IBDA to locate critical instructions. An instruction is urgent, if its position in the ROB is close enough to the head. The DnB core will put all non-critical instructions in a FIFO, which is called the Delay Queue (DLQ).

The DLQ is similar to the LTP delay FIFO. The critical instructions, which have all their operands ready at dispatch, will be put in the Critical Queue FIFO (CRQ), which is similar to the FXA ready at dispatch issue. Loads and stores are per definition always critical, but are placed in the IQ together with critical and non-ready instructions. This allows a reduction in both depth and width of the IQ. The Issue Select logic of DnB will consider the IQ, the heads of CRQ and the heads of the DLQ only if they are urgent. Since the readiness of the instructions in the DLQ is not known, an additional port to the busy table is needed for the head of DLQ. If the head of DLQ is urgent and not ready it is added to the IQ.

DnB simplifies the issue logic of the IQ, by only allowing it to issue two instructions per cycle, leaving another two for the CRQ and DLQ. If the DLQ head is urgent it is given priority over the CRQ.

Using simulators the authors compare a 4-wide out-of-order baseline with IQ depth of 64 to LTP, FXA and DnB, which all reduce the IQ depth the 32, as well as a BaselineHalf which also has an IQ depth of 32. The results are 84% (BaselineHalf), 89% (FXA), 91% (LTP) and 94%(DnB) of the baseline performance. For power consumption due to instruction scheduling FXA and LTP use 53% and 74% of the baseline, while DnB only uses 34% of the baseline. Area estimates are not reported.

7.3.3 CASINO

The CASINO core microarchitecture is a recent contribution to energy efficient out-of-order scheduling techniques. Instead of focusing on extracting a subset of the ILP it seeks a cheap change to in-order scheduling that captures most of the out-of-order performance gains. CASINO is inspired by other cores like LSC, Freeway and FXA [10]. It observes that slice-based architectures could experience serious slow-down due to different sizes and shapes of the slices [10], [9]. The CASINO core microarchitecture generates dynamic out-of-order instruction schedules by using two cascading in-order instruction queues. Instructions are first passed to the first FIFO, the *Speculative Queue* (SQ), where they get the chance to issue early if they are ready. Non-ready instructions are passed further to the second FIFO, the *In-order Queue* (InQ), where they wait for their operands to get ready and issue in-order. With such a simple scheme, CASINO reports a 49 % performance improvement over a normal in-order core.

Like the LSC, CASINO is built upon a 2-wide stall-on-use InO core. To avoid false dependencies among instructions CASINO implements a novel register renaming scheme. It is based on the observation that its only necessary to rename instructions that are executed out-of-order, because they are the only ones that cause hazards from false dependencies. CASINO therefore does register renaming after the SQ, and only if the instruction is selected for early issue. Otherwise, it is not renamed and placed into the InQ where it must wait until all prior instructions have issued.

CASINO also contributes a novel memory disambiguation technique that allows for speculation on memory ordering without needing expensive associative searches of the load queue. It employs a on-commit value-check which shifts the responsibility of detecting ordering violations from committing stores to the committing speculated loads [51]. This typically forces the loads to do an associative search of the store queue before they fire to memory, which would be comparatively expensive to the normal approach. CASINO proposes the *Outstanding Store Counter Array*, a hash table indexed by the lower order bits of the PCs of the stores. It tracks how many unresolved stores there are to different addresses. Loads will check whether there are unresolved stores to their addresses before they do any associative search in the store queue. If the counter is zero they fire directly to memory.

8 Implementation

This chapter presents our implementation of the different cores based off BOOM. We start with a discussion on the implementation of cheap instruction queues. Then we take an in-depth look at the BOOM issue unit which is the primary component we have modified. At last, we present the implementation details of LSC, DnB and CASINO. The source code is hosted on Github[1] The signal names in the figures match the signal names used in the source code.

8.1 Cheap Queue

A common theme for the proposed instruction scheduling techniques is to replace the complex and power hungry issue queues of the issue unit with cheap FIFOs. We have experimented with, and synthesized, various FIFO designs to investigate trade-offs between area and performance. A normal FIFO is a very simple structure which can be implemented as an SRAM with one read port, addressed by a head pointer, and one write port, addressed by a tail pointer. The head pointer points to the oldest valid micro-op in the queue, and the tail pointer to the first empty slot.

When an entry is enqueued to the FIFO the tail pointer is increased, and when it is dequeued the head pointer is increased. Initially, the head and tail pointer point to the same empty position. When the head pointer catches up to the tail pointer the queue is empty. The queue is full when the tail pointer reaches the head pointer. If the length of the queue l is not a power of two, the pointer increases need to implement wrap-around logic, that ensures that the pointer is set to entry zero after increasing from entry $l - 1$. For ls that are powers of two this is ensured automatically, if the pointer registers use $log_2(l)$ bits.

As synchronous read SRAMs have a one cycle delay for reads, there is a 1 cycle delay through the FIFO. This can be circumvented by a bypassable FIFO. In case the FIFO is empty, the writes bypasses the SRAM and are forwarded directly to the output of the FIFO. Our FIFOs are currently not bypassable. Furthermore, many dual-port SRAMs don't support forwarding the new data to the read output when the same address is written and read in one cycle. This can be solved by storing the written data in a register and multiplexing it to the read output when the read and write address match.

Our design had two main requirements to the instruction queues, which complicate the FIFO design. Firstly, branch mask updates have to be resolved in a single cycle in BOOM. Each branch in BOOM pipeline is associated with a unique branch tag. There is a limited number of tags, and this can cause pipeline stalls if a branch is fetched and there is no available tag for it. All micro-ops in BOOM are associated with a branch mask. The branch mask is a bit vector with one bit for each branch tag. The tags of the branches, which the micro-op is speculated under, are set to high. If

[1]https://github.com/EECS-NTNU/riscv-boom/tree/thesis-final

there is a branch misprediction, all micro-ops, which have that tag in their branch mask, must be squashed. As only one possible branch direction is taken for each branch, this affects all instructions following this branch.

If a branch resolves with a correct prediction, its bit must be set to low in the branch masks of all micro-ops, so that it can be used to identify a new branch. Increasing the number of branch tags is expensive, because the architectural register state is saved fore each branch tag in the rename stage. The branch resolution is broadcasted to all structures that are storing micro-ops, like the ROB, Fetch Buffer, IQ, LSU etc. These structures must have the micro-ops branch masks stored in content-addressable memories (CAMs), so that they can be looked up and modified in a single cycle. Likewise, our instruction queue must include a CAM for the branch masks of its entries, to enable single cycle updates or kills, when a branch resolves.

To avoid large holes in the FIFO the tail pointers need to be moved backwards, when a misprediction occurs and some of the micro-ops in the queue are invalidated.

The second requirement is that of multiple enqueues and dequeues. All our designs are 2-wide, which means that up to two instructions are decoded, renamed and dispatched together. Thus, our instruction queues must support up to two enqueues. All of our cores also should support issuing two micro-ops from any of the instruction queues, thus we need to support up to two dequeues.

8.1.1 Shifting Register Based Queue

The simplest Queue fulfilling these requirements is a register-based queue. Registers are implemented on the FPGA using D-type flip-flops. A register based queue can support parallel lookup and write on all its entries. It is thus very flexible, but also expensive. It doesn't use head and tail pointers but instead shifts values into registers. This automatically compacts the queue. Each cycle, each register can be assigned, by order of precedence, either its current value, the value of the m registers following it or the value from one of the enqueues. Where m is the number of dequeues supported in a cycle. It picks the first of those that is valid and has not been taken by another register. This ensures that, if n elements are in the queue, they will be in the first n registers. Branch resolutions are not an issue as all micro-ops can be looked up in parallel to update their branch mask or kill them. The dequeue ports are connected to the m first registers.

This design is similar to the shifting issue unit, and was created in order to verify that the forwarding logic of the queues matches the logic used by the issue unit.

8.1.2 Single SRAM Queue

As the goal of the scheduling techniques we evaluate is to be cheaper to implement than conventional out-of-order designs, we needed a more efficient implementation. An intuitive solution to this is to use a single SRAM with k read and m write ports for a queue supporting k enqueues and m dequeues. This solution still uses a single head- and tail-pointer and offsets the read and write ports based on that.

Consider an implementation with two enqueues and two dequeues. The micro-op from the first enqueue port is written to $headpointer + 0$ and the second to $headpointer + 1$. Each entry has a

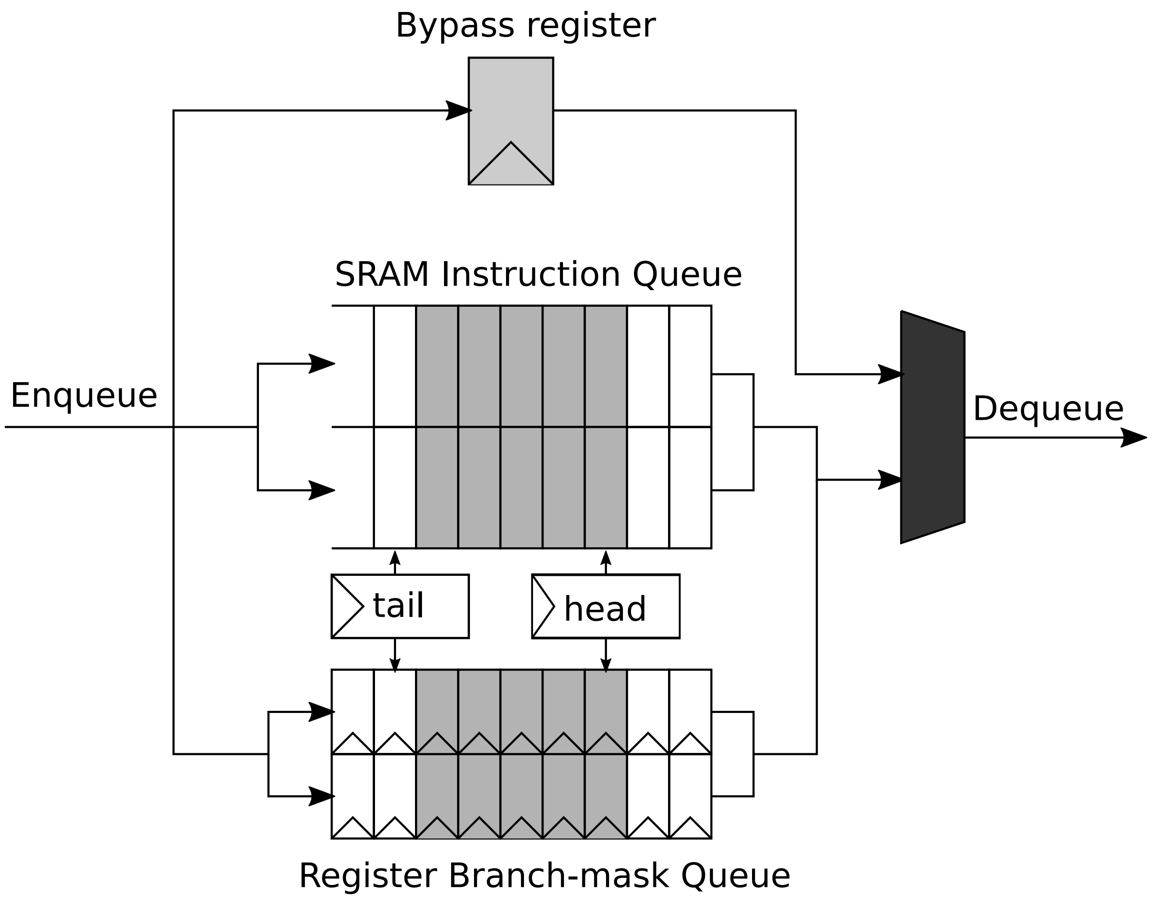


Figure 14: Overview of the Multi SRAM-Based Queue

corresponding valid register that marks if the entry is a valid micro-op or not. When a branch-misprediction is discovered, the tail pointer is set to the position after the last (youngest) valid micro-op. As there can be no holes in the queue, this position is at the one transition from valid to not valid. This means we can use a one-hot-indexed read-only memory containing the new address. This ROM needs $l\ log_2(l)$bit entries. If no entry remains valid, the tail-pointer is set to the value of the head-pointer. Because the SRAMs we use are synchronous, the read addresses need to be based on what the value of the head-pointer will be in the next cycle. I.e. they need to take dequeues in the current cycle into account. The disadvantage of this implementation is that an expensive multi-write SRAM is required.

8.1.3 Multi SRAM Queue

It is possible to use multiple independent SRAMs with one read- and one write-port each to implement a more efficient queue. Instead of treating the queue as a one-dimensional structure we treat it as a two-dimensional array. The number of columns represents the number of SRAMs, and is the maximum of the number of enqueues and number of dequeues. This means that each SRAM needs to only service up to one read and one write. The number of rows is the number of entries divided by the number of columns. Instead of one pointer, the head and tail now each have a row and a column pointer. If the number of rows and columns are both powers of two, they can be concatenated and treated as a single number for increases. Otherwise, the increase logic becomes more complex, as wrap-around logic is needed for both of them. The queue is shown in Figure 14.

8.1.4 Head Registers

When allowing multiple dequeues, an important question is, whether the queue needs to be strictly in-order. For a two wide dequeue port an out-of-order FIFO will allow dequeue of the second micro-op, even if the first micro-op is not dequeued. In a simple design this would create holes and in the following cycle the second micro-ops would be empty, even if more micro-ops are available in the FIFO. In order to achieve maximum performance, this kind of queue needs compacting functionality for its head. We add this by using registers to store as many micro-ops as can be dequeued in one cycle. These registers can be filled either from the SRAM or directly from the enqueue ports. If a micro-op goes directly from the enqueue port to the register, it is not written to the SRAM. This is only done, if the SRAM does not contain enough micro-ops to refill the head registers. Otherwise, the queue would no longer be in-order.

A stall-on-use design cannot use such out-of-order instruction queues.

8.2 Berkeley Out-of-Order Machine

This section provides an in-depth description of the BOOM modules which we have modified to implement our cores. It should not be confused with the later sections of this chapter which describes our original work.

Figure 15 shows the complete BOOM pipeline. The red-dotted lines mark the area where most of our work is done. Later figures of the pipeline will be zoomed in on this specific area as the rest of the pipeline is kept more or less unchanged.

8.2.1 Issue Unit

BOOM uses three separate issue units for the different execution units integer, memory and floating-point. This separation allows the usage of smaller queues with fewer issue ports, potentially decreasing the critical path. On the other hand, if instructions of a certain type are predominant its queue can fill up and become the bottleneck while the other queues remain largely empty.

Wake-up Logic

There are 3 types of wake-up in BOOM. Fast wake-ups are used for fast, fixed latency ALU operations. These wake-ups are sent out when the micro-op is issued.

Loads that hit the L1-D cache also have a fixed latency. In the cycle before the response from the L1-D cache a speculative load wake-up is sent out. If the load resulted in a cache miss in the next cycle, a load miss signal is sent out, invalidating the wake-up from the cycle before.

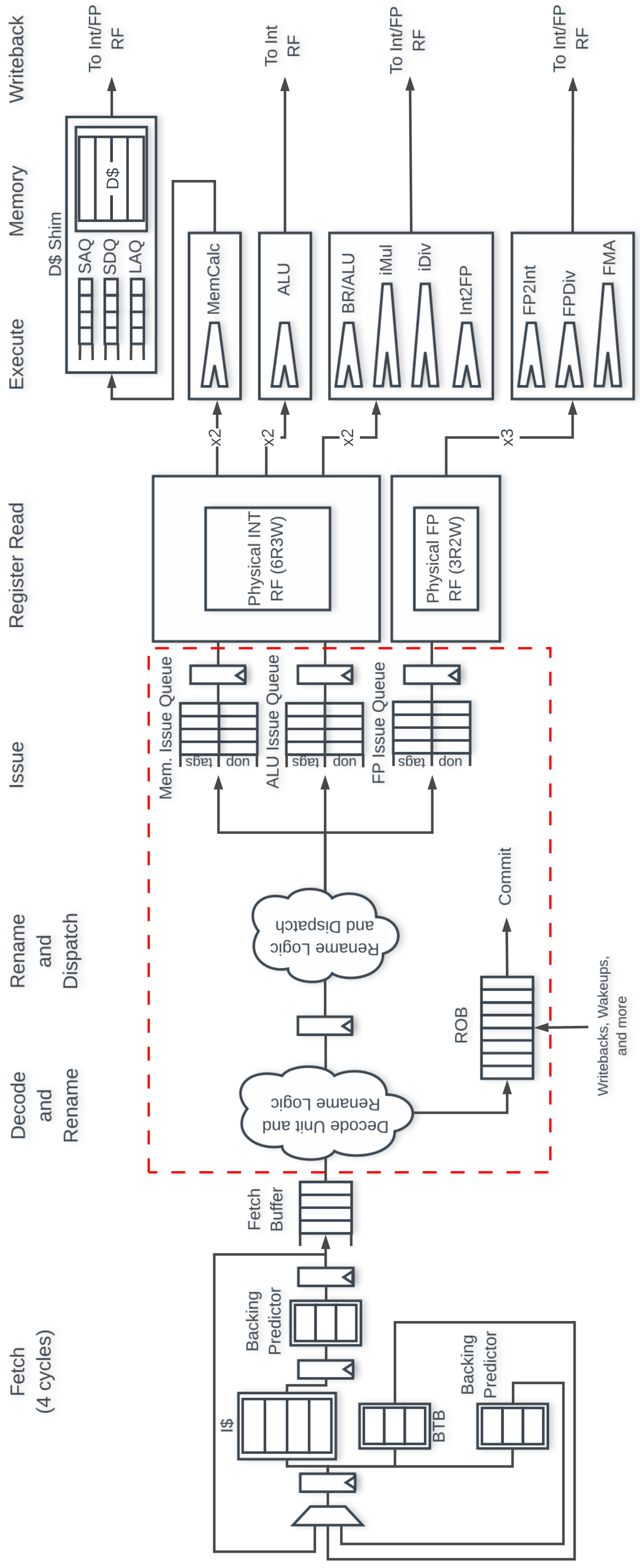


Figure 15: BOOM pipeline with area of interest marked in red. Adapted from the RISCV-BOOM documentation [2]

Slow wake-ups are used for all other operations. These are sent out when the register file is written. This means that the results from these operations never go through the bypass network but are always read from the register file.

When a dependent micro-op is issued in the cycle after a fast or load wake-up, the data is forwarded to it using the bypass network inside the register file.

Busy Table

The busy table stores for each physical register, if the result is available, or if it has not yet been calculated (*busy*). A register is set to *busy* when it is allocated for an instruction being renamed. When the busy table receives a fast or slow wake-up, the destination register of the wake-up is set to not busy.

The busy table is read in the second cycle of the rename stage. It contains all wake-ups from the previous, but not the current, cycle. Each micro-op is annotated with a busy bit for each of its physical source registers. In this cycle the micro-op is also written to a slot in the issue unit. The issue slot handles wake-ups for the incoming micro-op in this cycle.

Issue Slot

Inside the issue unit, micro-ops are stored in the issue slots. The issue slots monitor the wake-ups, and once all dependencies are met, i.e. not set to busy, the slot sends out a request signal. The request signal tells the issue unit that the micro-op is ready to be issued.

Store Splitting

In order to provide the load store unit with the store address as soon as possible, the issue slots support store splitting. The benefit of having the store address available is that loads can be checked against older stores, and load-store memory ordering violations can be avoided. Store splitting happens, when only one of the two source registers is ready.

Slot Shifting Mechanism

The shifting issue queue design employed by BOOM is outdated. While instruction age is a good issue selection metric, the shifting queue requires expensive compaction circuitry to fill gaps left by instructions that have been issued. This compaction logic increases the critical path of the issue queue. For this reason modern processors use different queue designs [17].

Issue Logic

The slots are ordered by age, with the oldest micro-op in the first slot. As each execution unit type has a separate issue unit, all issue ports have the same type of execution unit. However, the different execution units of one type do not all provide the same functional units. For example not every ALU must implement a multiplier. Furthermore, not all functional units are pipelined. This is for example not the case for the integer divider used by BOOM.

The consequence of this is that each issue port needs to provide a bit vector of the function types it can currently accept. The issue select logic goes through the issue slots, in age order, and when it finds an issue slots with ready operands, and an available function unit that matches, the mirco-op inside the issue slot is issued.

Unified Issue Queue

To enable easier exploration of different scheduling schemes we created a unified issue queue. This issue queue is connected to all execution units and receives all micro-ops. In consequence, it is no longer sufficient to compare the functional unit type, as the execution unit type of a micro-op and port also have to match. With the separate issue queues this was automatically guaranteed.

Instead of separating the instructions based on their targeted execution units in the dispatcher, this is now done in the issue unit. For most instructions this is straight forward since they target a single execution unit. The exception to this rule are float stores. As the memory execution unit doesn't have read access to the floating-point register file, float stores are executed by both the floating-point unit, to provide the data from the register file, and the memory unit, to calculate the address. These two parts are then combined by the LSU where the store is performed. To handle this scenario we have to either split the instruction into two micro-ops in the dispatch stage and buffer them, or issue instructions to multiple execution units in the issue unit. For InO and CASINO we decided to take the second approach, since they use very small issue units that can handle the additional complexity introduced by this approach. Store-splitting in dispatch for InO would require some sort of buffering mechanism between dispatch and issue. CASINO already uses queues, but these would require an increased number of write-ports. Since our DnB and LSC implementations already perform store splitting in the dispatch stage, in order to enable early address resolution, they can employ the simpler issue logic. This allows our DnB design to use a unified issue queue. However the critical path of the issue unit will still be slightly increased for DnB when compared to the conventional BOOM issue unit, due to the increased number of issue ports. This increase leads to both higher delay in the issue select logic and longer wires, and thus wire delays, caused by the increased size.

The issue unit checks if an execution unit of matching type and function is available for each candidate micro-op in order of priority. If the micro-ops dependencies are met it requests to be executed, and if a suitable execution unit is available the request is granted and the micro-op is passed on to the execution unit. The selected execution unit is marked as busy (no longer available) as it can only accept one micro-op per cycle. To handle float stores it is not enough to have one execution unit available. If not both a memory unit and floating-point unit is available the micro-op is not issued but instead waits for the next cycle. This makes this approach feasible without potentially affecting the critical path.

8.3 In-Order BOOM

We implemented two different versions of a stall-on-use in-order BOOM. Both of them are based on the unified issue unit design and support dual-issue. The first design uses a custom issue unit with two slots and stall-on-use issue logic. The second design was created to validate that there is no performance difference between the cheap queues and the issue unit. It helped us discover that our original implementations had a 5 instead of 4 cycle load-use penalty. This is important, since our other designs are based on the cheap queues.

8.3.1 Unified Dual-Issue

The dual-issue stall-on-use in-order (InO) gives us an important comparison point as it allows comparisons of wider designs. A problem with implementing dual-issue in-order based on BOOM is that instructions can reside in multiple issue units. This can be solved by using a unified issue unit.

The unified issue unit used in InO uses a shifting slot design with two slots. This means that if the micro-op in slot 0 executes, but the micro-op in slot 1 doesn't, the micro-op from slot 1 will be in slot 0 in the next cycle. Slot 1 will be filled by a new micro-op.

In the default implementation, when the micro-op in slot 0 doesn't execute, but the one in slot 1 does, the later instruction effectively bypasses the stall. This would mean the design is not completely in order. For example an instruction waiting for a high latency load could remain in slot 0, while execution of other instructions continues in slot 1. To prevent this we introduced stall on use logic. The micro-op in the second slot can only be issued if the micro-op in the first slot is also issued.

An other change to the issue unit was the ready logic, that tells dispatch how many instructions can be passed on. The default issue unit counts the number of slots that will be empty at the beginning of the next cycle (i.e. either are empty and not getting filled or being grated execution), and uses that to determine how many dispatch ports to set to ready. While this works fine for bigger issue queues for in-order boom this would mean the issue unit could only be refilled every second cycle. Instead, the selection of the shifts into the slots is now based on which slots will be empty, instead of are empty, and dispatch ports are marked as ready if there is enough space in the queue in the same cycle. This is possible due to the low amount of slots.

8.3.2 Queue based In-Order

As the logic deciding when an instruction is ready to issue relies on the busy table instead of the issue slots, it was important for us to verify that the behavior and performance of these matched. To do that we implemented a second version of in-order BOOM, based on the cheap queue. It has a two entry cheap queue residing in the dispatcher. The two heads of this queue are annotated with busy information by looking their source operands up in the busy table. Additionally, the speculative load wake-up from the previous cycle is stored in a register. If there is no load-miss signal, and the speculative load wake-up destination matches a source register, this source register is set to not busy.

The head micro-ops, along with busy information, are then forwarded to the issue unit. If none of their source registers are busy, the head micro-ops request to be executed. The issue unit issues from these two heads in the same cycle. This means that the one cycle delay from the queue, and the one cycle slot delay in the other in-order design, cancel each other out.

To match the performance of InO, the queue also has to use enqueue ready logic that treats a slot as empty when its content is being dequeued.

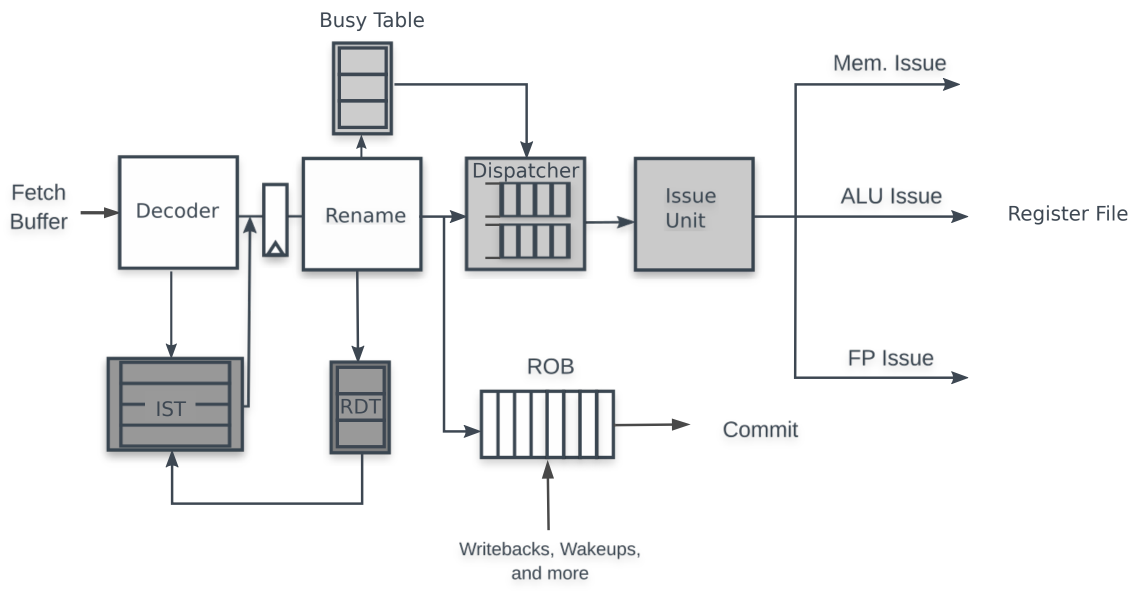


Figure 16: High-level view of Load Slice Core and Delay and Bypass instruction scheduling

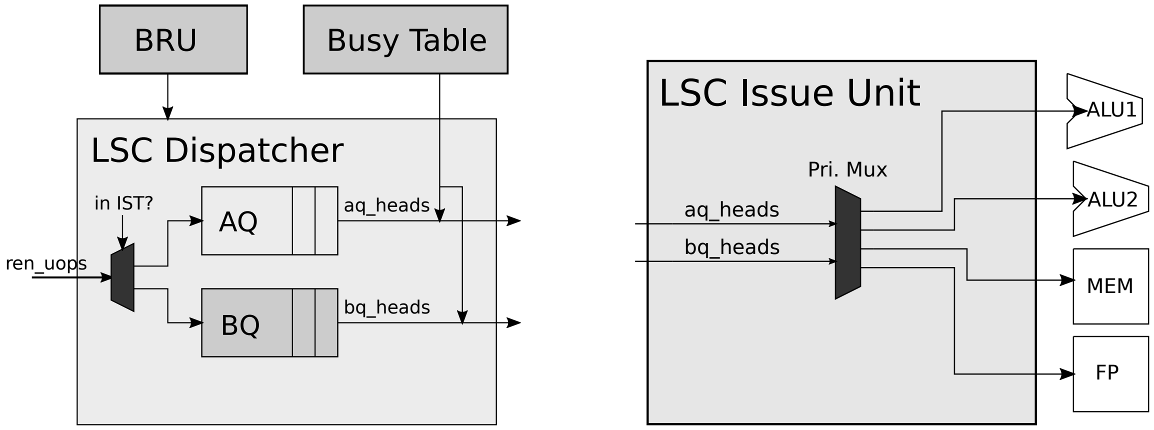
8.4 Load Slice Core

This section describes our implementation of the LSC architecture based on BOOM. The proposed implementation of LSC starts out from a stall-on-use in-order core and adds scoreboarding, register renaming and memory disambiguation [1]. We chose to start out from BOOM, which already has these structures and focus entirely on the Instruction Scheduling. LSC implements Single Write IBDA as described in Section 3.3 and uses Cheap Queues as described in Section 8.1.3. Figure 16 shows an overview of the LSC instruction scheduling. Light-gray marks modified, modules while dark-gray modules are new.

8.4.1 Dispatcher

While the original paper proposes implementing FIFOs in the issue unit, we have chosen to do so in the dispatcher to get a portable issue unit design. Micro-ops are passed to the dispatcher from the rename stage, together with info of whether they are identified as belonging to load slices or not. Figure 17a shows an overview of the LSC Dispatcher. For our 2-wide design, the dispatcher contains two dual-enqueue, dual-dequeue multi SRAM queues.

Our dispatcher is a quite simple structure that uses a multiplexer to write renamed micro-ops either to the AQ or to the BQ, based on whether the micro-op produced a hit in the IST or not. Before the heads of AQ and BQ are routed out to the issue unit, their busy bits are updated with the newest information from the busy table. The BQ has to be strictly in-order, to avoid memory



(a) Load Slice Core Dispatcher

(b) Load Slice Core issue unit

Figure 17: Load Slice Core instruction scheduling

ordering speculation, while AQ does not have any such requirement. We chose to implement AQ as an out-of-order FIFO using head registers as explained in Section 8.1.4.

The original BOOM dispatcher did not contain any sequential logic and contained just the glue between rename and issue. Now we introduce state into the dispatcher, and thus had to provide interfaces to both the branch resolution unit (BRU) and the busy table. The BRU broadcasts the results of branch executions. As all branches are predicted in the frontend, such a broadcast means one of two things. Either, we have made a correct prediction of the branch and all micro-ops in the dispatcher must update their branch masks, to remove the branch in question. Or, we have made a misprediction and all micro-ops, which have this branch tag in their masks, must be killed off.

The busy table is needed to give the issue unit updated information about the readiness of the source operands for each micro-op. Micro-ops with non-ready source operands are stalled in the issue unit, until their source operands are broadcasted on the wake-up network. AQ and BQ are not connected to the expensive wake-up network, and we must instead send a request to the Busy Table when the micro ops are dequeued and sent to the issue unit.

Not shown in the figure is that the dispatcher also must respond to pipeline flushes, by flushing all queues and resetting the head and tail pointers.

Store splitting

An important feature of the LSC is that, while it allows for some out-of-order execution, it keeps strict program order of the calculation of store and load addresses. This means that whenever a load arrives in the LSU, the address of all younger loads and stores are known. This is then used to avoid any memory ordering conflicts. BOOM speculates on these memory ordering conflicts and sends all loads speculatively to the L1 cache right away, and deals with potential ordering conflicts when the address of any younger and conflicting store or load arrives. LSC achieves this by splitting the store instructions into two parts. The address generating part is added to the BQ, while the data generating part is added to the AQ. This disables the load speculation of BOOM because it will always know the address of younger stores before it executes any load.

8.4.2 Issue Unit

Figure 17b shows the issue unit of our LSC. It is a variant of the unified issue unit described in Section 8.2.1. All the issue slots are removed and thus the expensive wake-up network as well as most of the Issue Select logic. The issue unit has a ready-valid interface to the heads of the AQ and BQ and based on the busy information of the heads, the issue unit will issue up to four instructions each cycle. Issue Select is achieved through a simple priority multiplexer, which gives priority to ready AQ heads.

8.5 Delay and Bypass

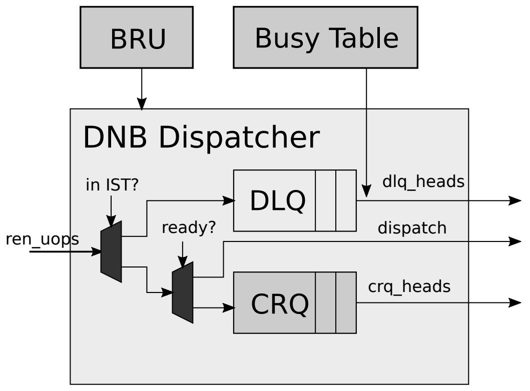
This section describes how we implemented the DnB core architecture based on BOOM. While LSC seeks to eliminate the issue slots completely, DnB only wants to reduce their width and depth. Width in this sense means the number of issue ports and depth the number of issue slots. From a high-level perspective its implementation is similar to LSC and Figure 16 is accurate for DnB as well. For criticality analysis we use Single Write IBDA as described in Section 3.3.

8.5.1 Dispatcher

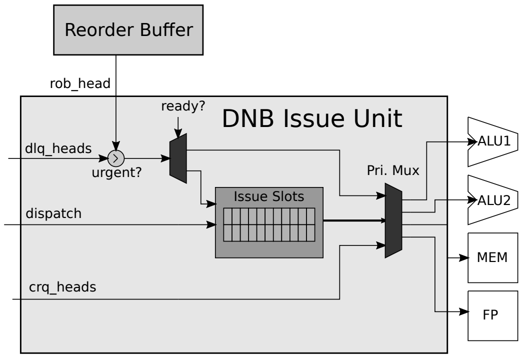
Figure 18a shows the DnB dispatcher. Renamed instructions are passed through 2 multiplexers. Instructions that generated a hit in the IST are sent to a second multiplexer while instructions that were not found in the IST are sent to the DLQ. The second multiplexer is controlled by whether the instruction is ready or not. Ready instructions are sent to the CRQ. Non-ready instructions are dispatched directly to the issue queue of the issue unit, where they are placed in issue slots. According Alipour et al. [8], loads and stores should always go directly to the issue queue. We propose to split stores, both integer and floating point, and put the address generating part in the issue queue for early execution. The data-generating part is then placed in the DLQ, as it is likely to be dependent on instructions that are already there. The DnB dispatcher has an extra port to the busy table for the DLQ heads. The readiness of their operands is looked up and passed with the instructions to the issue unit. The CRQ does not require such a lookup as we already know that the instructions in the CRQ are ready.

8.5.2 Issue Unit

Figure 18b shows our implementation of the DnB issue unit. The issue unit is centered around a priority multiplexer which performs the issue select. The DLQ heads have their reorder buffer (ROB) indices compared against the ROB head. The comparison reveals, how many older uncommitted instructions are already in the backend. If the delta is sufficiently small, the instruction is urgent and will passed on to the ready-multiplexer. The ready-multiplexer will either forward it to the priority multiplexer or add it to an empty issue slot, dependent on whether the operands are ready or not. Instructions sent over the dispatch interface are added directly to the issue slots. Here they wait for their operands to get ready. The CRQ heads are known to be ready and are passed directly



(a) Delay and Bypass Dispatcher



(b) Delay and Bypass Issue Unit

Figure 18: Delay and Bypass instruction scheduling

to the priority multiplexer.

Alipour et al. [8] suggest having two issue ports allocated for the issue slots and two issue ports shared between the DLQ and CRQ, prioritizing urgent and ready DLQ entries. Working with BOOMs issue logic we face similar difficulties as with the LSC. Since we have static connection between issue ports and execution units, a division of issue ports would mean a division of execution units. As we only have duplicate integer ALUs it doesn't make sense to fully divide the issue ports. Thus, our implementation considers all issue slots, DLQ heads and CRQ heads for all issue ports and issue based on priority. This is however slightly different than proposed, as the CRQ now has low priority after both the critical DLQ heads and all the issue slots.

8.6 CASINO

This section explains how we implemented the CASINO core architecture based on BOOM. CASINO represents the simplest approach to improving in-order instruction scheduling. There is no effort to classify instructions e.g. using IBDA. Instead, there is a small instruction window where instructions ready at dispatch are allowed to execute early. Figure 19 shows an overview of our implementation of the CASINO core instruction scheduling.

8.6.1 Dispatcher

Figure 20a Shows the CASINO dispatcher. All renamed instructions are written to the tail of the SQ. When they reach the head of SQ they are, dependent on whether their operands area ready or not, either enqueued to the tail of the InQ or passed directly to the issue unit and issued. Whether they are ready or not is looked up in the busy table. Instructions at the head of the InQ are also passed to the issue unit, together with updated ready-info from the busy table. The dispatcher also has a port to the BRU to squash entries on a mispredict and update the branch masks on a correct prediction.

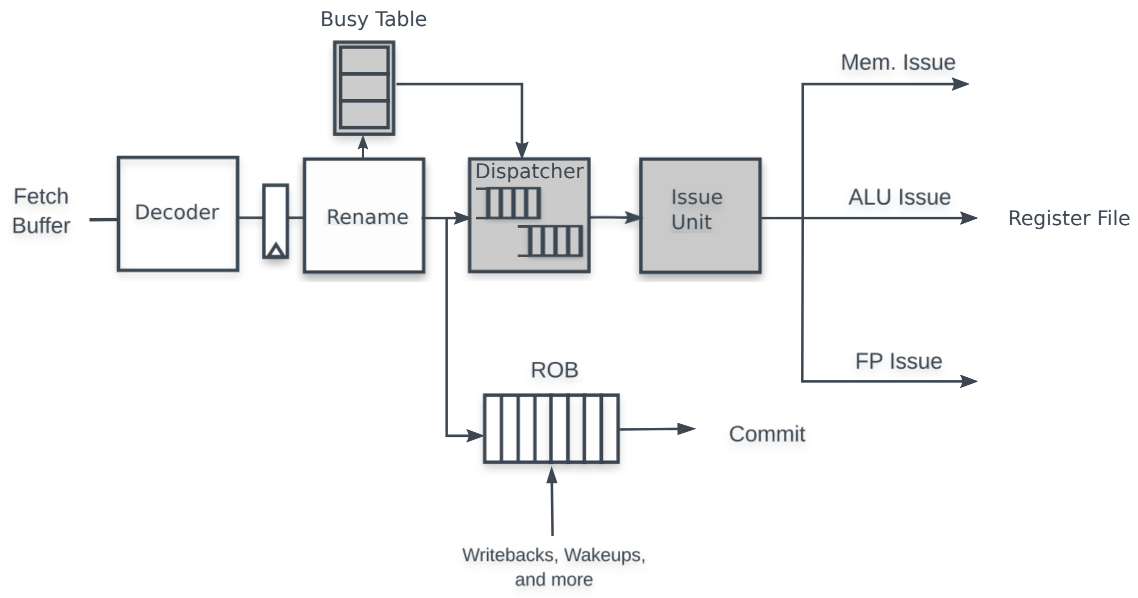
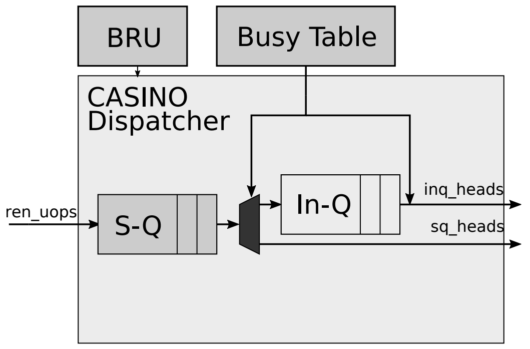
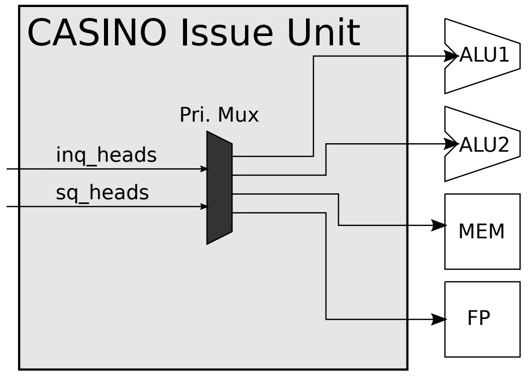


Figure 19: High-level view of CASINO instruction scheduling



(a) CASINO Dispatcher



(b) CASINO Issue Unit

Figure 20: CASINO instruction scheduling

8.6.2 Issue Unit

Figure 20b shows the CASINO issue unit. It is the same issue unit that we have used for the LSC and the InO BOOM. It is stripped of the Issue Slots and the bypass network. It only considers the heads of SQ and InQ and prioritizes the InQ heads.

9 Methodology

This chapter discusses the methodology used for evaluating the different instruction scheduling techniques. We start by looking at how size and power is evaluated using FPGA synthesis tools. Then we look at how the application performance is evaluated by emulating the full microarchitecture on an FPGA. Lastly, we report the configuration parameters used to build the cores that we evaluate.

9.1 Size & Power Evaluation

This section describes our size and power evaluation. We start by looking at what resources an FPGA has and how we perform synthesis. We finish with a discussion about FPGA power estimation tools and why we choose not to use them.

9.1.1 FPGA Resources

This section provides a short introduction to the resources on an FPGA, as our area analysis is based on comparing the usage of these resources for the different designs. A Field-Programmable Gate Array (FPGA) is a flexible integrated circuit that can implement complex digital designs like microprocessors. The configurable logic block (CLB) is the fundamental building block of an FPGA. For Xilinx 7-series FPGAs the CLB contains four 6-input Lookup Tables (LUT), eight D-type flip-flops, three multiplexers and two 4-bit carry chains for addition. Large FPGAs contain arrays of hundreds of thousands of CLBs [24]. Xilinx 7-series FPGAs also have Block RAMs (BRAMs), which are flexible SRAMs that can be configured in port configurations. Each BRAM is 36 Kb large and can be configured as two independent 18 Kb RAMs. The LUTs can also be configured as SRAMs, called LUTRAMs. A 6-input LUT can be used to store 64 bits [23]. An FPGA can also contain hardwired logic blocks for commonly used functions, like multiplication. These are called DSPs. The logic functions of the LUTs, as well as the interconnects between the blocks, are written to configuration SRAMs on power-up.

For simplicity, we have limited the comparison to Logic LUTs, flip-flops, BRAMs and LUTRAMs. BRAMs are counted in number of RAMB18 blocks, i.e. a RAMB36 counts as two BRAMs.

9.1.2 Synthesis Flow

By default, Vivado flattens the module hierarchy in order to optimize across module boundaries. While this leads to smaller total sizes it is harmful when analyzing the sizes of individual structures. In order to avoid this we disabled this feature for our size estimates.

We also encountered a bug where some of our designs would only function correctly if this feature was disabled. As this lead us to conclude that functional equivalence with and without flattening is not guaranteed, despite Xilinx claims, we disabled it for all our work. When generating Verilog from Chisel, we disabled all prints, assertions and *DontTouch* annotations that would have polluted our designs with control signals.

The floating-point and multiplier units of BOOM and Rocket are described as combinational logic followed by multiple pipeline registers. In order to be synthesized with the intended pipeline they require register retiming. Register retiming shifts the position of registers in pipelined logic, in order to balance out the timing and reduce the critical path. Vivado offers support for register retiming by offering the `-retiming` option. This enables retiming globally. Individual modules can be retimed in Vivado by annotating the registers that should be retimed in Verilog [52]. As the complete Verilog code is re-genearted for every change of the Chisel design, this approach is not practical for our workflow. When we evaluated the timing of BOOM we found out, that the critical path does not pass through the floating point unit and thus disabled retiming.

9.1.3 Vivado Power Estimation

We attempted to perform power analysis using Vivado Design Suite's power estimation flow [53]. It estimates power consumption for a design based on resource usage, I/O loading, toggle rates of signals, power supplies, ambient temperature and other factors. These factors are either derived from the design, extrapolated or supplied by the user.

For accurate estimates the tool needs information about the activity and toggle rates. This can be produced by simulating the design and giving the simulation output to Vivado. Unfortunately we do not have access to the proprietary simulators that produce the output format Vivado needs. Our power estimates are therefore based on vectorless, or probabilistic, estimation. The vectorless engine assigns a static probability for the output of all the nodes in the design. It then starts from the primary inputs and propagates the activity to the primary outputs and thus estimates the total activity of the whole design. [53]

For a CPU this can lead to nonsensical values as Table 4 demonstrates. It is especially apparent when comparing DnB and LSC. One would expect DnBs power consumption to be roughly equal except for the *Dispatch* and *Issue* stages. However, the power consumption of DnBs *Integer Register File* and *Execute* stage as well as *Reorder Buffer* are higher than even those of the out-of-order core. If these activity rates were realistic it would mean, that DnB issues more instructions and should be more performant than OoO, given that they share most of the core infrastructure. As this is not the case, as shown in Section10.5, the results have to be wildly inaccurate. For this reason we decided against using them as part of our results.

One other option we considered is disabling the probability propagation entirely and letting Vivado assign the same toggle rate and static probability everywhere. This would eliminate the anomalies demonstrated above but also discount the structure and logic of the design entirely. The other problem we faced is that the power consumption of only partially used BRAMs is overestimated, when compared to their size. For example the SRAM based queues we use need SRAMs with one write and one read port and eight 142 bit entries. These are synthesized as two 512 entry 72 bit RAMB36 blocks combined into one 512 entry 144 bit memory. This memory is 64 times larger than it would need to be. The power evaluation tools in Vivado are aimed to evaluate the power consumption of FPGA systems and, as only complete BRAM blocks can be disabled to save power,

	InO	CASINO	LSC	DnB	OoO
Core	0.347	0.361	0.438	0.523	0.432
Frontend	0.118	0.117	0.161	0.166	0.155
Branch Prediction	0.075	0.075	0.104	0.104	0.101
L1I cache	0.012	0.012	0.015	0.015	0.015
Fetch Buffer	0.002	0.002	0.002	0.002	0.002
ITLB	0.004	0.004	0.007	0.006	0.006
Fetch Target Queue	0.018	0.019	0.021	0.026	0.021
Frontend Rest	0.007	0.005	0.012	0.013	0.010
Backend	0.124	0.134	0.151	0.220	0.143
Decode	0.000	0.000	0.000	0.000	0.000
Integer Rename	0.020	0.018	0.016	0.020	0.021
Maptable	0.016	0.014	0.012	0.015	0.016
Freelist	0.002	0.002	0.002	0.003	0.002
Busytable	0.000	0.000	0.000	0.000	0.000
Integer Rename Rest	0.002	0.002	0.002	0.002	0.003
FP Rename	0.018	0.015	0.014	0.017	0.018
Maptable	0.015	0.012	0.011	0.014	0.015
Freelist	0.002	0.002	0.002	0.002	0.002
Busytable	0.000	0.000	0.000	0.000	0.000
FP Rename Rest	0.001	0.001	0.001	0.001	0.001
Dispatch	0.000	0.013	0.023	0.028	0.000
Queue 1	0.000	0.003	0.012	0.014	0.000
Queue 2	0.000	0.010	0.011	0.013	0.000
Dispatch Rest	0.000	-0.000	0.000	0.001	0.000
Issue	0.000	0.000	0.000	0.024	0.013
Issue Mem	0.000	0.000	0.000	0.000	0.003
Issue Integer	0.000	0.000	0.000	0.000	0.007
Issue Float	0.000	0.000	0.000	0.000	0.003
Issue Unified	0.000	0.000	0.000	0.024	0.000
Integer Register File	0.039	0.037	0.038	0.053	0.041
Execute	0.024	0.024	0.023	0.032	0.026
Control Status Register	0.006	0.006	0.005	0.006	0.005
Reorder Buffer	0.013	0.014	0.012	0.017	0.014
IBDA	0.000	0.000	0.015	0.015	0.000
RDT	0.000	0.000	0.005	0.005	0.000
IST	0.000	0.000	0.010	0.010	0.000
Backend Rest	0.004	0.007	0.005	0.008	0.005
Load Store	0.069	0.073	0.074	0.074	0.073
Floating Point	0.034	0.036	0.050	0.061	0.059
Core Rest	0.002	0.002	0.002	0.002	0.002

Table 4: Unrealistic power utilization estimation of dual-issue stall-on-use BOOM (InO), CASINO, Load Slice Core (LSC), Delay and Bypas (DnB), 2-wide BOOM (OoO) in W.

include the power consumption of a whole BRAM.

9.2 Application Performance Evaluation

This section goes into detail of how we evaluated the application performance of the different cores. We start by taking a closer look at the FPGA we use emulate the core. Then we briefly explain how our core is instantiated as part of a system-on-chip using Chipyard followed by a detailed discussion of the memory subsystem on the FPGA. We also introduce the RISC-V performance counters, before taking a look at issues with the BOOM branch predictor. Then we introduce the RISC-V Proxy Kernel, as a way of running applications bare-metal. We also look at booting and running Linux. Lastly we discuss SPEC2006, the benchmark suite we use.

9.2.1 Target Hardware

In order to get size and clock frequency estimates for the different designs they need to be synthesized. We are using Xilinx Vivado 2018.2 to target a Xilinx Zynq ZC706 board [54]. This board uses a XC7Z045 FPGA SoC that provides an FPGA with 350K logic cells as well as a dual-core ARM Cortex-A9 CPU. The board provides 1 GB DDR3 RAM for the ARM cores, and a further 1 GB DDR3 RAM that can be used by the FPGA by instantiating a memory controller in the FPGA fabric. The DRAM attached to the ARM cores can also be accessed from the FPGA fabric using an AXI bus. When this is used, uboot has to be configured to instruct the Linux kernel running on the ARM cores to only use a part of this memory. This limits the memory usable from the FPGA using this approach to 768 MB. Using the DRAM attached to the FPGA fabric allows emulating a system with a dedicated memory controller without interference. The downside is, that utilization of FPGA resources goes up, which could affect timing and limit large configurations. Our RISC-V cores are synthesized with a harness that is used for initialization and communication with the ARM host. The harness connects to the ARM core through an AXI interface to provide it with memory mapped registers and FIFOs used for serial communication, block device emulation and resets. It is also used to control and initialize the system using TSI (Tethered Serial Interface) [55].

9.2.2 System-on-Chip

Our FPGA evaluation infrastructure is based on a BOOM-specific branch of fpga-zynq [56]. Since UC Berkeley has transitioned to FireSim [57] fpga-zynq has been deprecated. Additionally it was based on a precursor of Chipyard [13]. Chipyard is a framework that generates system on chips (SoCs) using Chisel. In order to work with recent BOOM versions we had to port the FPGA harness to Chipyard.

At the time of writing we are using the most-recent development version of Chipyard. Our fork of it, containing a generator for the FPGA harness, is hosted on Github[1]

9.2.3 Memory Subsystem Analysis

The latency of the memory subsystem is an important factor for the performance. As our cores are emulated on an FPGA using the on-board resources it is important to investigate how this affects

[1]https://github.com/EECS-NTNU/chipyard/tree/thesis-final

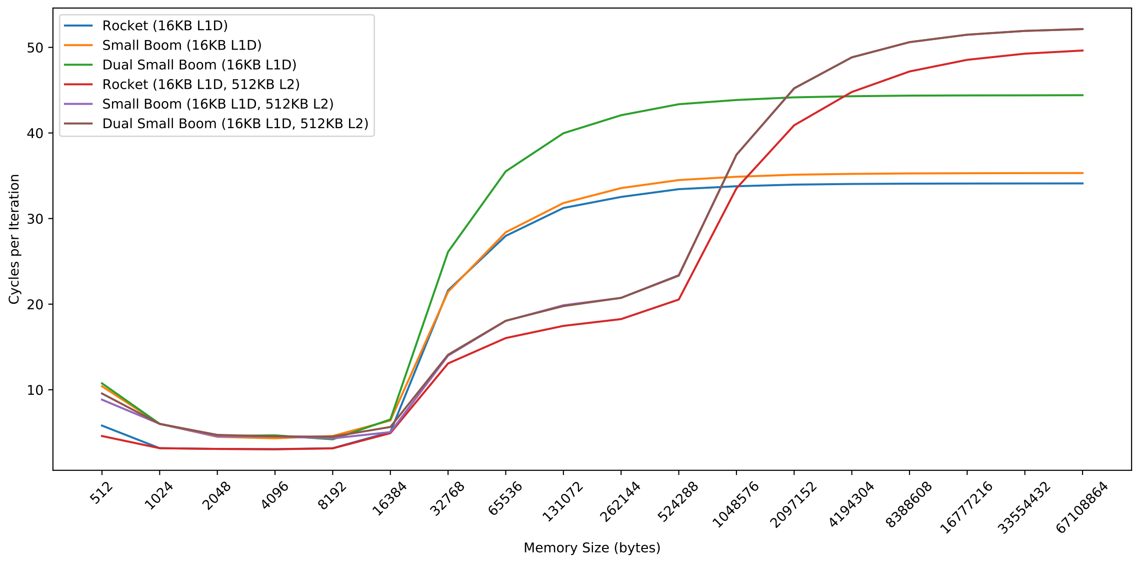


Figure 21: Memory hierarchy latency of Rocket and BOOM for different working set sizes.

the overall performance.

One problem that FPGA-prototypes face is that their DRAM runs at a very high speed compared to the processors. The ZC706 board we use contains DDR3 800 memory [58] running at 1600 MT/s. Meanwhile, our BOOM implementation runs at a frequency of 50 MHz. This impacts both the throughput and the latency of the memory subsystem.

We considered two different memory subsystems, one with 512 KB L2 cache and one without. Both use the DRAM dedicated to the FPGA. Figure 21 shows the memory latency of these configurations for different working set sizes. The 16 KB L1-D and 512 KB L2 cache are visible as distinct step ups in latency. The latencies were calculated by creating a chain of pointers, similar to a linked list, at random locations within a memory area of a given size, and measuring the time it takes to follow this chain. As the address of a load is the result of the previous load, the loads are guaranteed to be sequential. This pattern allows us to measure the time it takes for several iterations to pass and then divide it by the number of iterations, to get a close approximation of the time for a single load. We had previously tried to measure the timing of individual loads by exploiting the fact that BOOM gives control-status-register instructions exclusive access to the pipeline, but got unrealistically high values for BOOM compared to Rocket. Since the RISC-V cores run at 50 MHz while the DRAM operates at 1600 MT/s the DRAM access latency is unrealistically low. Normally DRAM of this speed would be paired with a processor running upwards of 1 GHz.

The dual core configuration of BOOM seems to inflict a memory access time penalty. This penalty remains the same if L2 cache is added, which is why *Small BOOM* and *Dual Small BOOM* overlap when configured with an L2 cache. The introduction of L2 cache worsens the DRAM access time, as for every request the L2 tags are read, and it is only passed on in the memory hierarchy if an L2 miss occurs. Due to the low number of cycles it takes to access DRAM, this penalty is bigger than usual for our system. We still decided to include L2 cache, since performance in benchmarks improved, and the different cache levels and longer DRAM access time should be more realistic. Because the tests were performed in machine mode without virtual memory, the penalty for TLB misses is not included in this analysis.

The DRAM of our SoC behaves more like a huge L3 cache than traditional main memory. FireSim solves this problem by instantiating FASED [59], a cycle-accurate DRAM model, on the FPGA and routing all main memory accesses through it. This model was not easily adaptable to our platform since it relies on the token- and handshake-based simulation approach provided by Midas [60].

9.2.4 Performance Counters

RISC-V allows the integration of custom performance events, that can be mapped to performance counters. Access to performance counters has to be enabled in machine mode. They can be subsequently configured and read from lower privilege modes [61]. We used these events to measure the number of cycles, instruction, branches, branch misses and instructions in different queues.

The biggest problem we experienced when using BOOM was poor branch predictor performance. Early on, while investigating IBDA we noticed that BOOM's branch predictor was mispredicting even simple loops. At first, it seemed like this issue only affected the first prediction level, the next line predictor, and would be caught by the larger backing predictor. This turned out to be wrong, as BOOM's branch predictor is beaten across the board by Rocket's much simpler prediction system. The poor branch prediction performance was also noticed by others and was supposed to get fixed in a future release of BOOM that will bring "performance exceeding any prior version" [62].

The wide instruction windows of modern out-of-order processors require very high branch prediction accuracy in order to be practical. If speculation far ahead is not possible because the poor prediction performance causes wrong paths to be taken, the instruction window effectively becomes much narrower, limiting ILP and MLP.

Fortunately, a new version of BOOM (SonicBOOM/Boom v3 [11]) was released a few weeks before our deadline. We were able to port most of our changes to BOOM to this improved version. The version we used before was somewhere between Boom v2 and BOOM v3, referred to as BOOM v2.5 above, but lacked the improved frontend. BOOM v3 raises the branch prediction penalty from 10 to 12 cycles, but offsets this with much more accurate branch prediction.

BOOM v3 uses a combined branch predictor consisting of a small micro BTB that provides single-cycle redirects for small loops, a high-performance TAGE predictor and a return address stack with snapshot and repair functionality. We use a version of BOOM v3 with the predication-based short-forwards branch optimization described by Zhao et al. [11] disabled.

Figure 22 compares BOOM v2.5 and BOOM v3 in similar configurations, but with their respective branch-prediction infrastructures, to Rocket. The harmonic mean IPC goes from 0.44 to 0.5 while the harmonic mean branch misprediction rate goes from 17.8 % to 0.4 %. In the benchmarks *bwaves*, *cactusADM* and *leslie3d* there is only a very small change in IPC, even though the branch misprediction rate changes drastically. This is because these programs have a low amount of branch instructions, giving a much lower impact to the misprediction rate.

9.2.5 Proxy Kernel

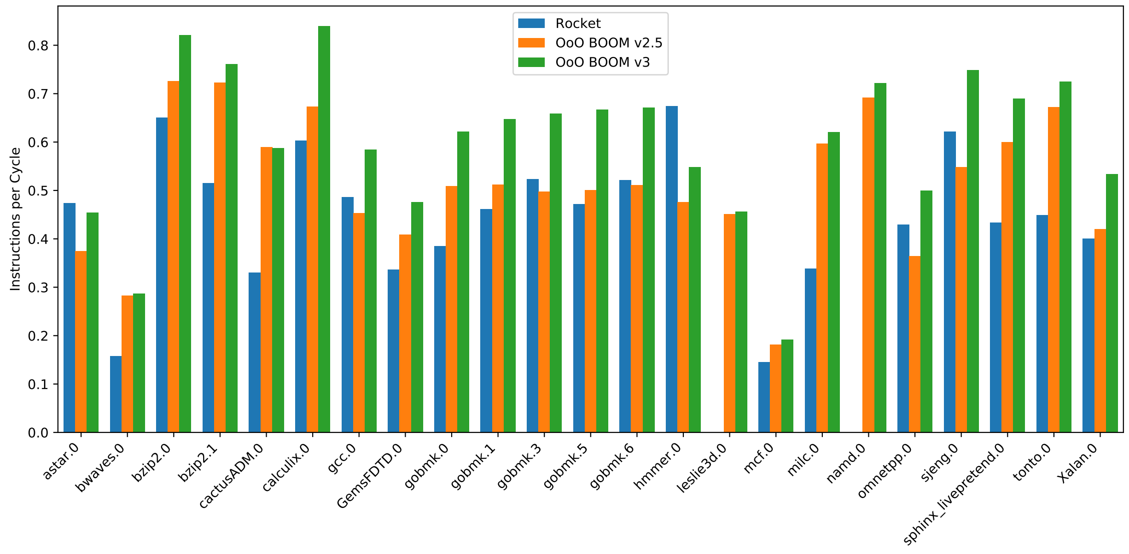
The RISC-V Proxy Kernel (PK) [63] allows applications, that are not intended to run bare metal, to run without a full operating system. It does this by providing support for some system calls directly and forwarding the other system calls to a host system running FESVR [64], the RISC-V frontend-server. When running PK on the Verilator or the Spike emulator, the host system is the Linux system where the emulator runs. For the ZC706 board, the Linux running on the ARM cores serves as a host. While the PK allows running applications without an operating system, and thus without unpredictable context switches, it suffers the drawback of having to communicate back to the host for system calls, i.e. all file system access is performed via these calls. This means that not all of the application runs on the target system. Furthermore, the time spent waiting for a response from the host can influence measurements. The PK is started by providing it with the path of the application binary and its arguments. It then initializes the system (e.g. exception handlers and virtual memory), calls back to the host to get the binary, loads it into memory and starts the application with the provided arguments.

When using PK, some system calls are executed by the host system running FESVR. During these calls the PK waits for a spin-lock that is set by the host, once its response is ready. Since the proportion of cycles spent waiting on the host is large for shorter I/O-intensive benchmarks, this behavior would skew the results our performance counters collect. To resolve this we read the counter values at the beginning and end of each call to FESVR, calculate the change in the counters and sum these differences up. This allows us to effectively eliminate the cycles spent waiting for the spin-lock by subtracting the summed up measurements from our final values. Since the spin-lock loop consists of only a load and a branch, it would otherwise skew CPI as well as queue count numbers.

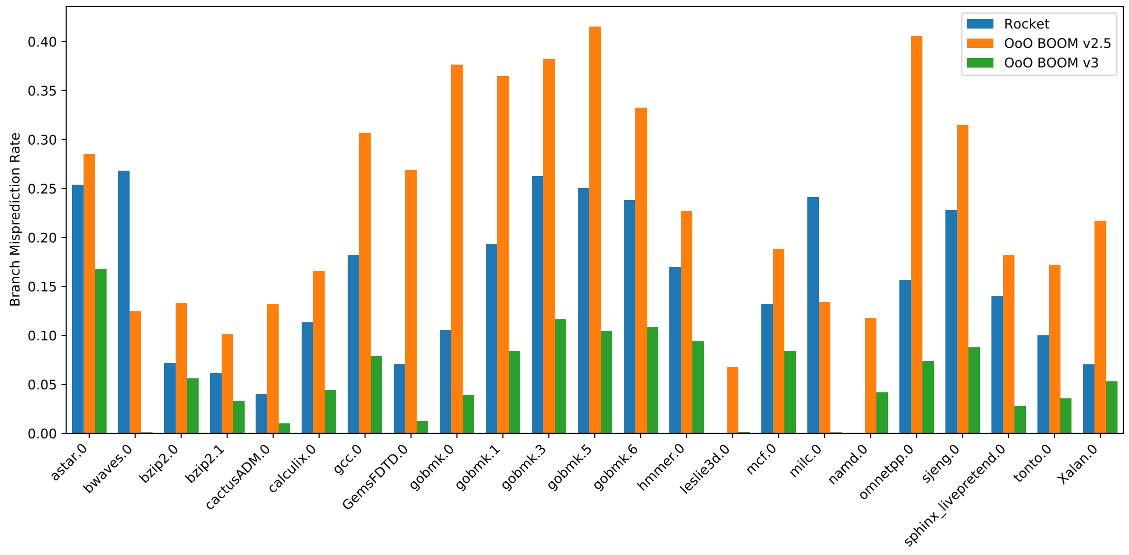
Figure 23 demonstrates that the performance measured using the proxy-kernel and using perf under Linux show very similar behavior for the different benchmarks. IPC under Linux is consistently slightly worse with the only exceptions being *bwaves* and *mcf*.

9.2.6 Linux

While running benchmarks on Linux should give the more realistic measurements than the proxy kernel, it introduces a new set of difficulties. As Linux is a multitasking operating system, benchmarks no longer have exclusive access to resources. On the other hand, the whole execution profile of applications is covered since system calls are handled on the RISC-V processor. The only communication to the host system that is necessary is for accesses to the emulated block device and UART communications. These can happen asynchronously like in a real system. For BOOM v3 we encountered a bug that prevents Linux from booting. Because BOOM v3 was released so close our deadline we were unable to fix this issue. Therefor we can only report Linux performance numbers for the old version that uses the inferior branch prediction infrastructure.



(a) IPC



(b) Branch Misprediction Rate

Figure 22: Performance comparison of BOOM v2.5, BOOM v3 and Rocket cores.

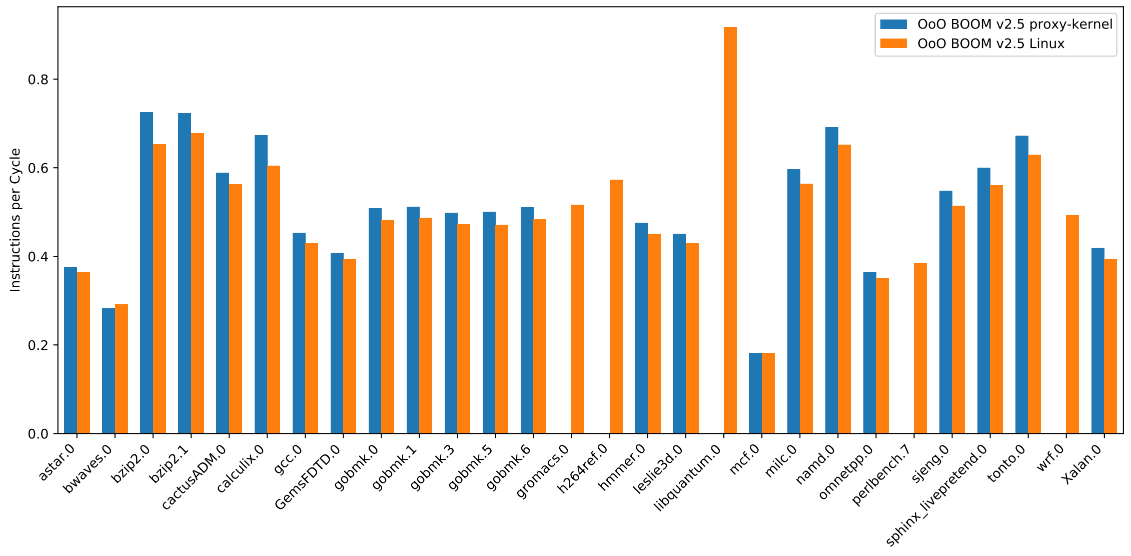


Figure 23: Performance of BOOM v2.5 running SPEC CPU2006 using Linux and RISC-V Proxy Kernel

We considered using a dual core configuration for Linux benchmarking, in order to reduce the number of context switches and operating system influence on benchmarks, however the configuration we used for comparisons is too large to fit onto the FPGA twice. In tests with smaller configurations we observed a noticeably lower number of context switches when having an additional idle core. It should be possible to achieve the same effect by using a bigger core, the test target, combined with a smaller Rocket core to reduce operating system overhead. This would require pinning application benchmarks to the first core, to ensure that the right core is evaluated.

We use Linux images created using FireMarshal [65], based on the buildroot framework. The image we use runs through the SPEC suite on startup and then automatically shuts down. Afterwards we can mount the image and extract the results onto the host system. It is also possible to interact with the Linux system using a simulated UART connection. To measure performance under Linux we use the perf stat command.

9.2.7 SPEC 2006

This section introduces the SPEC2006 benchmark and how we used it to benchmark the cores.

To evaluate application performance we use workloads from the test configuration of the SPEC CPU2006 Benchmark. The benchmarks were built using Speckle [66], a toolset that simplifies the build-process of SPEC for RISC-V. While it might be better to use the inputs for the full benchmark runs instead of the test inputs, the test inputs should still provide realistic results at a fraction of the run time. FPGA accelerated simulation is orders of magnitude faster than cycle-accurate emulation of hardware description languages, but as we only had one board big enough to emulate BOOM, we were not able to execute benchmarks in parallel. Furthermore, we had to run full benchmarks,

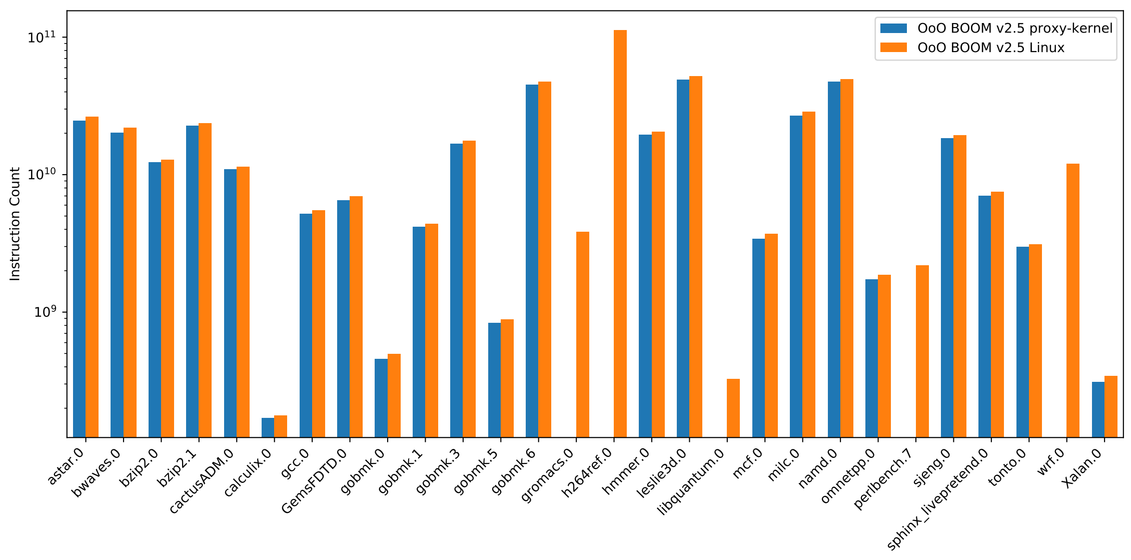


Figure 24: Instruction count for BOOM v2.5 running SPEC CPU2006 using RISC-V Proxy Kernel and Linux

as our current infrastructure does not support save points and traces. This kind of workflow might be possible in the future, by migrating to a FireSim-based architecture [57].

Not all applications within SPEC work reliably when using the proxy-kernel. Some rely on system calls which are not supported, while others crash for unknown reasons or never complete. This, combined with the general problematic of offloading part of the execution to the host when using the proxy-kernel was the reason why we explored benchmarking using Linux instead. Figure 24 shows the number of instructions executed for Linux and the proxy-kernel. Linux has a small overhead for all applications. We assume that this difference is caused by the different implementations of system calls, and the fact that some system calls are not executed on the RISC-V CPU, when using proxy-kernel. For *perlbench* these differences are so large we decided to remove *perlbench* from most of our results. This decision was reinforced by the fact that, taken on its own, each of the *perlbench* executions is a micro benchmark. Especially *perlbench.2* exhibited very strange performance across all our measurements. The longest one, *perlbench.7* did not run under the proxy-kernel because it used an unsupported system call.

We also considered using the newer SPEC CPU2017 benchmark but decided not to use it, as it states a minimum memory requirement of 2 GB.

Averages reported are calculated using the harmonic mean. First benchmarks are grouped by applications and, and the harmonic mean of the application group is taken. We then calculate the harmonic mean of these means. E.g. all *gobmk*'s are collected, averaged and treated as one number in the second mean.

	SInO	Rocket
Fetch Bytes	8	4
Fetch Buffer Entries	8	5
Fetch Target Queue Entries	16	-
Branch Prediction	TAGE based	Simple BTB
Decode Width	1	1
ROB Entries	8	-
Branch Count	4	-
Issue Width	1	1
Integer Register File	40x64 bits 4r2w	32x64 bits 2r1w
Float Register File	36x65bits 3r2w	32x65bits 3r2w
Load Queue Entries	4	-
Store Queue Entries	4	-
L1-D Sets	64	64
L1-D Ways	4	4
L1-D Block Size	64 bytes	64 bytes
L1-D TLB Entries	32	32
L1-D MSHRs	2	0
L1-I Sets	64	64
L1-I Ways	4	4
L1-I Block Size	64 bytes	64 bytes
L1-I TLB Entries	32	32

Table 5: Single issue in-order BOOM (SInO) and Rocket configuration parameters

9.3 Core Configuration Parameters

In this section we report the configuration parameters needed to build the cores we evaluate. For a description of how to generate the Verilog and perform synthesis please refer to Appendix A.

Table 5 shows the configuration parameters for the size comparison between a stripped down single issue in-order version of BOOM (SInO) and Rocket. SInO must not be confused with the dual-issue stall-on-use in-order BOOM (InO), which serves as the in-order baseline for our performance and size evaluation later. Table 6 shows the configuration parameters used for our different core implementations.

	InO	CASINO	LSC	DnB	OoO
Fetch Bytes	8	8	8	8	8
Fetch Buffer Entries	16	16	16	16	16
Fetch Target Queue Entries	32	32	32	32	32
Branch Prediction	TAGE	TAGE	TAGE	TAGE	TAGE
Decode Width	2	2	2	2	2
ROB Entries	64	64	64	64	64
Branch Count	12	12	12	12	12
Dispatch Queue entries	-	4+2,12	16+2,16	16+2,16+2	-
Issue Width	4	4	4	4	4
Issue Slots	2	-	-	12	20×INT, 12×MEM, 16×FP
Integer Register File	80 6r3w	80 6r3w	80 6r3w	80 6r3w	80 6r3w
Float Register File	64 3r2w	64 3r2w	64 3r2w	64 3r2w	64 3r2w
Load Queue Entries	16	16	16	16	16
Store Queue Entries	16	16	16	16	16
L1-D Sets	64	64	64	64	64
L1-D Ways	4	4	4	4	4
L1-D Block Size	64B	64B	64B	64B	64B
L1-D TLB Entries	8	8	8	8	8
L1-D MSHRs	2	2	2	2	2
L1-I Sets	64	64	64	64	64
L1-I Ways	4	4	4	4	4
L1-I Block Size	64B	64B	64B	64B	64B
L1-I TLB Entries	32	32	32	32	32
L2 cache Size	512KB	512KB	512KB	512KB	512KB
L2 cache Ways	8	8	8	8	8
L2 TLB Entries	1024	1024	1024	1024	1024
RAM Size	1GB	1GB	1GB	1GB	1GB

Table 6: Configuration parameters for dual-issue stall-on-use BOOM (InO), CASINO, Load Slice Core (LSC), Delay and Bypass (DnB) and BOOM (OoO)

10 Results

In this chapter we report the results of our evaluation. The focus of this section is area efficiency, and we start with a detailed breakdown comparing our stripped-down single-issue in-order BOOM (SInO) to Rocket, an optimized in-order core. We follow up, by comparing the sizes of our different instruction queue implementation. Then we present the size statistics for our different core implementations. We also look at how our designs have affected the critical path. At last, we present the performance results from running SPEC on the different implementations.

10.1 Size Overhead from BOOM

The main goal of this report is to compare different energy efficient microarchitectures and their instruction scheduling. The different architectures are positioned between full out-of-order and in-order. Our implementations are all based on BOOM which is fully out-of-order, but we had created an in-order BOOM (InO) as a second baseline. This works well for performance evaluation, as InO functionally is in-order. However, for the size evaluation all the designs, except the out-of-order baseline, will have some degree of size overhead. This is because we have not optimized away or slimmed down all unnecessary structures. In this section we will do a detailed break down of these unnecessary structures by comparing SInO, a stripped-down single-issue in-order BOOM with Rocket, an open source in-order core.

10.1.1 Configuration

Table 5 shows the configurations for SInO and Rocket. We tried to make the configurations as similar as possible, but there was some inevitable divergence. The following paragraphs will explain these.

SInOs L1-I cache interface is twice as wide as Rocket. SInOs fetch buffer needs to be larger than the fetch width, which is eight bytes or four compressed instructions. The fetch target queue stores the PCs of fetch bundles, as they are not part of the micro-op in BOOM. As Rocket is not meant for wide instruction windows, it forwards the PC along with the instruction. A big difference between Rocket and SInO is the branch prediction scheme used. SInO uses a much larger TAGE based predictor.

The reorder buffer (ROB) was configured to hold 16 entries for SInO. Rocket, as a simple in-order core, doesn't need an ROB. SInO uses a bit vector, called branch mask, that indicates which branches an instruction is dependent on. When a misprediction is detected, all instructions that have the corresponding branch bit set are flushed. SInOs branch count of four means that it has a four bit long branch mask, and can have four outstanding branches at a time.

SInOs use of register renaming required an increase in the number of physical registers. For Rocket the number of physical and logical registers are equal. As both the ALU and memory address calculation unit are connected directly to the register file, they require dedicated ports. This doubles the number of read and write ports of SInOs register file. Register writes from floating-point to integer instructions share the write port used by the LSU. The floating point register file only differs in the number of entries. As RISC-V has a fused multiply add (FMA) instruction three read ports are required. The loads store unit (LSU) writes to the floating point register file using a separate write port in both designs. Rocket and SInO use similar ALUs and floating-point units. The floating-point units use a 65 bit representation internally.

The load queue and store queue are also only present in SInO and not Rocket. The L1 Cache configurations matches, with the exception that Rocket has no miss status holding registers (MSHRs).

10.1.2 Sizes

Table 7 lists the resource utilization of the different components of BOOM and Rocket. Numbers higher in the hierarchy are the sum of the lower members. Where it was not possible to attributed resources to specific functionalities, they were put in a *Rest* category. This was also done to simplify some structures. Some modules had to be moved in the hierarchy, as BOOM and Rocket place them in different positions. Additionally, a few small changes to Rocket had to be made in order to generate the register files and decoder as modules. The following sections compare and discuss each entry of the Table 7.

Frontend

The frontend of SInO is considerably larger than that of Rocket. The branch prediction module contributes a lot to this with its more complex TAGE-based prediction scheme. The *L1-I Cache* and *I-TLB* (Instruction Translation Lookaside Buffer) are comparable in size, as they use similar configurations. SInO uses a fetch buffer with more entries and write ports, resulting in more LUTs used, but Rocket stores the PC alongside the instruction resulting in higher flip-flop usage. The PCs are stored inside the fetch target queue (FTQ), a structure that does not exist in Rocket. The FTQ is used in SInO because it is intended to have a much wider instruction window than Rocket. Keeping the PC alongside the instruction in the pipeline would considerably increase the storage requirements. Instead, the FTQ is written during the fetch stage in the frontend and read by the branch unit before execute and by the reorder buffer during the retirement of an instruction causing an exception. Compressed instructions are expanded into their non-compressed forms in both Rocket and SInO. This simplifies the later stages of the pipeline, as from that point on only uncompressed instructions have to be handled. SInOs frontend is pipelined into four stages, that each handle up to four instructions. Rockets frontend has two stages. Frontend Rest is considerably more expensive for SInO, as SInO decompresses and pre-decodes up to four instructions per fetch packet. A fifth compressed expander and pre-decoder pair, that could be an overhead caused by the recent rework of BOOMs frontend, is also instantiated. In contrast, Rocket decompresses instructions at the end of the frontend, after the fetch buffer, and does not pre-decode them for branch prediction.

Decode & Issue

SInOs decode logic is larger than Rockets, because it fully decodes instructions to provide a higher degree of abstraction. The instruction word itself is no longer part of the micro-op after the decode

	Logic LUTs		Flip-flops		BRAMs		LUTRAMs		DSPs	
Core	51145	31676	35109	15747	109	51	1052	384	36	27
Frontend	17190	4001	16365	4003	66	12	419	17	0	0
Branch Prediction	8571	1142	10752	1461	46	0	260	16	0	0
L1I Cache	635	696	638	548	12	12	1	1	0	0
Fetch Buffer	849	234	385	450	0	0	0	0	0	0
ITLB	1475	1366	1314	1350	0	0	0	0	0	0
Fetch Target Queue	2594	-	2072	-	8	-	0	-	0	-
Frontend Rest	3066	563	1204	194	0	0	158	0	0	0
Backend	11126	6105	6658	2695	16	0	91	88	16	16
Decode	273	89	0	0	0	0	0	0	0	0
Issue	260	-	119	-	0	-	0	-	0	-
Integer Register File	1515	135	827	0	16	0	0	88	0	0
Execute	2817	1714	756	307	0	0	91	0	16	16
ALU	570	603	0	0	0	0	0	0	0	0
Divider	1165	866	209	213	0	0	0	0	0	0
Multiplier	244	245	159	94	0	0	0	0	16	16
Memory Addr. Calc.	85	-	0	-	0	-	0	-	0	-
Execute Rest	753	-	388	-	0	-	91	-	0	-
Control Status Register	2352	2576	1505	1648	0	0	0	0	0	0
OoO Backend Overhead	3272	-	3056	-	0	-	0	-	0	-
Integer Rename	1372	-	1299	-	0	-	0	-	0	-
Map Table	731	-	930	-	0	-	0	-	0	-
Free List	427	-	202	-	0	-	0	-	0	-
Busy Table	146	-	40	-	0	-	0	-	0	-
Integer Rename Rest	68	-	127	-	0	-	0	-	0	-
FP Rename	1373	-	1234	-	0	-	0	-	0	-
Map Table	806	-	960	-	0	-	0	-	0	-
Free List	364	-	182	-	0	-	0	-	0	-
Busy Table	144	-	36	-	0	-	0	-	0	-
FP Rename Rest	59	-	56	-	0	-	0	-	0	-
Reorder Buffer	527	-	523	-	0	-	0	-	0	-
Backend Rest	637	1591	395	740	0	0	0	0	0	0
Load Store	12543	8494	8427	5132	15	39	155	1	0	0
Page Table Walk	5126	5048	2728	2731	3	3	0	0	0	0
DTLB	1873	1765	1532	1532	0	0	0	0	0	0
L1D Cache	3138	1681	2795	869	12	36	155	1	0	0
MSHRs	1539	-	1104	-	0	-	154	-	0	-
L1D Cache Rest	1599	1681	1691	869	12	36	1	1	0	0
Load Store Unit	2406	-	1372	-	0	-	0	-	0	-
Floating Point	10125	12913	3631	3890	12	0	115	0	20	11
FP EXU	8160	7629	2550	1648	0	0	0	0	20	11
dfma	2258	2258	547	546	0	0	0	0	9	9
divsqrt	1851	1347	1048	363	0	0	0	0	9	0
fpiu	1109	1138	138	138	0	0	0	0	0	0
fpmu	382	382	347	206	0	0	0	0	0	0
sfma	975	976	296	257	0	0	0	0	2	2
ifpu	1585	1528	174	138	0	0	0	0	0	0
FP Register File	479	4573	484	2080	12	0	0	0	0	0
FP Rest	1486	711	597	162	0	0	115	0	0	0
Core Rest	161	163	28	27	0	0	272	278	0	0

Table 7: Sizes of the SInO BOOM (left) and Rocket (right) cores

stage. In contrast, some parts of Rocket operate directly on the instruction word. As Rocket does not use a separate module to decide when to pass on instructions to the register file, the cost of its Issue logic is not broken out. In SInO, Issue is a separate pipeline stage that stores one micro-op in flip-flops. This stage would be larger for an out-of-order configuration. The integer register file of SInO has four read and two write ports, instead of two reads and one write, as the memory address calculation and integer execution unit are meant to be issued to independently. This also highlights an important aspect, the fact that FPGAs have specialized resources. The register file is implemented using LUTRAMs for Rocket. Since LUTRAMs and BRAMs only provide one write port, Vivado synthesizes SInOs register file using flip-flops and LUTs by default. This would dramatically increase the cost of the register file and makes comparison harder. For this reason we decided to replace BOOMs register file with a more efficient multi-ported memory, as it is described in Section 2.4. The optimized register file is still considerably larger than Rockets.

Execute

While SInO uses similar execution units as Rocket, the pipelined multiplier SInO uses has more stages, resulting in a higher amount of flip-flops used. SInOs execute stages main overhead comes from the storage and logic requirements of the micro-ops stored alongside the execution path. Rocket stores a smaller amount of information, that is counted as part of Core Rest. The memory address calculation logic is also not a standalone module in Rocket.

The differences in the size of the control status register are likely due to the higher amount of status information that Rocket provides.

Rename

OoO Backend Overhead contains modules that are needed by an out-of-order BOOM which are mostly just overhead for SInO.

The rename stages contains a map table that maps logical to physical registers. The map table also creates a snapshot whenever a branch instruction passes through the rename stage. This means that its size grows with the number of supported outstanding branches. The same is true for the free list, that stores if a physical register is currently in use or can be allocated. The final component is the busy table. It tracks whether the physical registers are busy or not. A register is busy, if it has an outstanding write by an instruction somewhere in the backend. As the busy state is only relevant for registers that are depended upon and is reset once a register is re-allocated, it does not need to take branches into consideration.

The free list of the integer rename stage is larger because SInO uses more physical integer registers. On the other hand, the map table of the FP rename is larger, because it needs to provide three read and write ports instead of two and keep track of 32 instead of 31 logical registers, since integer register zero is always zero. The integer rename stage also has higher Rest resource utilization since it stores a whole micro-op, because the rename stage is spread over two cycles. Because integer rename and FP rename are accessed in parallel, only a part of this overhead needs to be stored in FP rename.

Reorder Buffer

The last piece of overhead is the reorder buffer (ROB). The ROB preserves the commit order and handles exceptions. For multi-issue processors with in-order pipelines a simpler scoreboarding approach can be used. Simple single-issue stall-on-use processors such as Rocket don't require such a mechanism, since they write to the register file in-order. The ROB is organized as a queue and commits at the head of the queue. Instructions are inserted at the tail of the ROB, when they are passed to the issue unit from rename.

Load Store

The load store mechanism of BOOM adds MSHRs (Miss Status Holding Registers) and a dedicated load store unit. The MSHRs reside in the L1-D Cache, but are shown separately. They enable the L1-D cache to continue servicing requests, even when a miss occurs. As Rocket stalls on cache misses, its L1-D cache does not implement this functionality. The BRAM utilization of Rockets L1-D cache is much higher, because its data arrays are organized as 32 512x8 bit instead of four 512x64 bit arrays. While the capacity remains the same, and Rockets implementation might enable power-saving, features it causes a stark increase in size on FPGAs. We assume that the increased register usage of the L1-D cache Rest is also caused by its non-blocking, pipelined, behavior. The load store unit contains the load queue and store queue, that are used to enable out-of-order execution of memory operations. The store queue is split into the store address and store data queue. Treating these as independent structures also allows treating the data and address part of stores independently, by splitting the store. This allows the store address to be in the store queue before the data dependencies of the store are ready, aiding in avoiding executing loads that depend on this store.

Floating-Point

While the floating-point unit uses execution units, SInOs floating-point divide and square-root unit, divsqrt, is larger and more performant than Rockets. Some other execution units differ in their pipelining registers, causing increased flip-flop usage. This is because SInO uses the same pipeline length for all floating point execution units, to allow them to share a register write port. As Rocket waits for instructions to write the register file before issuing new ones, it can use units with different latencies. The FP register file of SInO contains more registers but uses the same read- and write-port configuration as Rocket. The large size difference stems from the fact, that SInO uses a BRAM-based register file while Rocket uses one based on registers and LUTs. While this means they are not directly comparable we could not simply use SInOs register file for Rocket, since when tests failed when it was used. FP Rest is once again bigger for SInO, since it stores micro-ops alongside the execution path.

Core Rest contains a buffer and crossbar that connects the L1-I and L1-D cache with the lower memory hierarchy using the cache-coherent TileLink [67] interface. Its cost is equal for both configurations.

10.1.3 Summary

One aspect that has to be considered and should not be underestimated is the degree of optimization in both designs. Rocket is an older, more mature design that has frequently been used on FPGAs. In contrast, BOOM is still under heavy development, as shown by the recent release of BOOM v3 that improved its branch prediction performance considerably.

Also not reflected in these measurements is the price of a wider core. For example, for a two wide core the rename unit needs to take the destination register of the earlier first instruction into account, when calculating the source registers of the later second instruction. This requires forwarding logic and can mean that the cost of adding more width to a core is more than linear. On the other hand, other structures can be shared, making them comparatively cheaper. Some parts of BOOM are more expensive, as they are intended to be wider than the configuration used here.

10.2 Queue Sizes

In this section we compare the sizes of different FIFO queues. We compare the three different queue types described in Section 8.1.3 with a base capacity of 16 entries and two enqueues and dequeues. Table 8 shows the sizes of the different configurations. Queues with the suffix -H support dequeueing instructions at the head of the queue out-of-order. The ShiftingQueue supports this natively. The SingleQueue and MultiQueue use head registers as described in Section 8.1.4. This extends their capacity to 18 entries. The SingleQueue and MultiQueue use an outer queue, that contains the head registers in the -H configurations, and an inner queue that contains the FIFO SRAMs. The outer queue also multiplexes the enqueues into the right order. BOOMs rename unit sometimes passes on a bundle of micro-ops where only the second but not the first is valid. In this case a hole would appear because only the address after the tail is written. The re-ordering logic multiplexes the second entry to the first enqueue port if the first entry is not valid.

The ShiftingQueue is most expensive, since it relies solely on registers and uses expensive shifting logic. Comparing the SingleQueue with the SingleQueue-H and the MultiQueue with the MultiQueue-H shows that the cost of adding head registers for out-of-order dequeue is around 570 LUTs and 290 flip-flops. The inner queue Rest of the MultiQueue is more expensive than that of the SingleQueue, because it needs to multiplex from and to the different SRAMs. The MultiQueue requires two single-write single-read 8-entry 142 bit SRAMs. The SingleQueue requires one dual-write dual-read 16-entry 142 bit SRAM implemented using an LVT. These SRAMs need to be able to read and write the same address in one cycle. This requires the use of external bypass registers and multiplexers, that switch the output data of the read to the registered value of the previous write. Each SRAM in the MultiQueue needs two 72 bit wide BRAM36. Since it uses 2 SRAMs and we count each BRAM36 as two BRAMs that results in a total of 8 BRAMs. The queue could be extended to 1024 entries using the same amount of BRAMs, since only 8 of the 512 entries are used. The SRAM in the SingleQueue is more expensive, since it has two write ports and two read ports. $2reads \times 2writes \times 2BRAM36s \times 2BRAMs/BRAM36 = 16BRAMs$. This SRAM could hold up to 512 entries using the same BRAMs since only 16 entries of each BRAM are used.

	ShiftingQueue-H			SingleQueue			SingleQueue-H			MultiQueue			MultiQueue-H		
Queue	3861	2208	0	1497	499	16	2063	785	16	1671	479	8	2268	765	8
Inner Queue	0	0	0	1189	499	16	1213	499	16	1495	479	8	1485	479	8
SRAMs	0	0	0	566	287	16	566	287	16	252	264	8	252	264	8
Inner Queue Rest	0	0	0	623	212	0	647	212	0	1243	215	0	1233	215	0
Queue Rest	3861	2208	0	308	0	0	850	286	0	176	0	0	783	286	0

Table 8: LUT (left), flip-flop (center) and BRAM (right) utilization of different queues with 16 entries.

10.3 Core Sizes

Compared to our out-of-order baseline (OoO), InO, CASINO and LSC see a 14 %, 11 % and 7 % reduction in total LUT usage, respectively, while DnB sees an increase of 2 %. For total flip-flop usage, all cores shows a reduction compared to the out-of-order baseline. InO with 9 %, CASINO with 7 %, LSC with 6 % and DnB with 2 %. LUTRAM usage of all cores are identical, except CASINO which shows an 18 % increase. BRAM usage of InO remains unchanged while CASINO has a marginal 6 % increase. LSC and DnB both have a 22 % increase compared to baseline OoO.

The following sections are a component by component comparison in resource usage of the different cores.

10.3.1 LUTs and Flip-Flops

Table 9 shows the number of LUTs and the number of flip-flops used by the different implementations. The numbers are presented hierarchically, where e.g. Frontend, includes Branch Prediction and L1-I Cache, while itself being included in the numbers for Core.

Frontend

The frontend is identical for all the cores, with the exception of the fetch target queue (FTQ). LSC and DnB have a 612 LUT increase in the FTQ compared to the others. This increase is due to the Single Write IBDA implementation. It uses the full PC as the tag and needs additional ports to the FTQ to get it.

Decode

The decode module shows some minor variations in LUT usage. This can likely be attributed to optimizations done by Vivado.

Integer Rename

The integer rename stage has some expected variations that can be traced to the busy table. CASINO, LSC and DnB all have additional ports to the busy table, to look up the heads of the FIFOs in the dispatch stage.

FP Rename

For FP rename we observe some anomalies. As expected, CASINO, LSC and DnB have a higher LUT usage in the busy table due to the additional read port. However, InO has more than 100 additional

LUTs in the free list. As the Verilog code for the different free lists is identical, we attribute this to cross module optimization done by Vivado.

Dispatch

The dispatch stage is particularly important, as we have performed a lot of modifications. As expected, InO and OoO have virtually no logic in this stage, as this is not an actual pipeline stage but rather glue logic between rename and issue. Starting with LSC, it is easy to get the idea of the cost of allowing out-of-order dequeue from the queues. A-Q supports out-of-order deqeue, this is achieved by putting the queue heads in separate registers and refill them from the SRAM-based queue. B-Q is strictly in-order. Other than that they are identical. The out-of-order dequeue causes an increase of 651 LUTs and 286 flip-flops. For DnB, the DLQ uses 92 LUTs more than the CRQ. This is because the DLQ must keep busy information and information abut the type of the source operands, i.e. floating point or integer, with the micro-op. The heads of DLQ are looked up in the busy table, before they are sent to the issue unit, Lastly, CASINO shows some interesting features. The InQ can hold 12 micro-ops, while the SQ can only hold four micro-ops. But since the SQ has out-of-order dequeue, and the InQ doesn't, the SQ still uses 754 more LUTs and 161 more flip-flops. Normalized to CASINO, which has the cheapest implementation, LSC and DnB adds on 59 % and 74 % LUTs and 19 % and 44 % flip-flops, respectively.

Issue

Issue is the main target of these instruction scheduling optimizations. OoO uses 13330 LUTs and 5121 flip-flops distributed among three separate issue queues. The individual sizes of these issue queues vary a bit because the micro-ops passed through them require different control signals. InO only spends 834 LUTs, as it has no issue slots, only a very simple issue select logic. The issue unit is now composed of a unified issue queue rather than separate issue queues. InO has 280 flip-flops which represent two pipeline registers for micro-ops to decouple the issue unit from rename. CASINO and LSC both have more or less the same LUT usage as InO, but do not need any pipeline registers, as their dispatch stage contains them. DnB strives for a "best-of-both-worlds" approach. It has a third of the issue slots that OoO has, but combines them in a unified issue unit. This leads to an expected 66 % reduction in flip-flop usage, as the issue slots are implemented with flip-flops. However, we only see a 43 % reduction in LUT usage. This is the main caveat of our DnB implementation and will be discussed in detail in Section 11.2.

IBDA

LSC and DnB both use Single Write IBDA to calculate program slices. The resource usage of IBDA is analyzed in great detail in Section 5.2.

Lastly, the Load Store and Reorder Buffer both shows minor variation in both LUT and flip-flop usage. This, again, we attribute to unknown optimization passes done by Vivado as we cannot trace any difference in the Verilog.

10.3.2 Memories

Table 10 shows the LUTRAM and BRAM usage for the different cores. InO and OoO has the exact same resource usage. This is expected as the instruction scheduling did not use any memories to begin with. LSC and DnB have the same LUTRAM usage as InO and OoO, but shows an increase of 28 BRAMs. This is also expected as we use BRAMs for the Dispatch queues and for IBDA. CASINO shows additional LUTRAM usage over OoO, this is because the SQ in Dispatch is so small Vivado mapped it to LUTRAMs rather than BRAMs.

10.4 Critical Path

This section details how and why our designs affected the critical path. We estimate the achievable frequency by synthesizing, placing and routing the designs targeting 50MHz and letting Vivado report the critical path. We then take the slack of the critical path, subtract it from the clock period and convert the resulting period into a frequency. This results in a frequency of 50.72 MHz for DnB, 52.76 MHz for CASINO, 53.14 MHz for LSC, 53.42 MHz for InO and 55.27 MHz for OoO.

The Load Store Unit sends out a speculative wake-up in the cycle before it gets a response from the L1-D cache. This wake-up is canceled, by sending out a load miss signal, if the L1-D cache reports a miss. In the default, out-of-order configuration of boom, if an instruction that was woken up by the speculative load wake-up signal is issued in the following cycle and the load miss signal is received, the instruction in the slot will remain valid, even though the instruction was already issued. In essence this means the instruction is issued twice. The first issue is caught by the core logic and the instruction is set to invalid. Thus it does not continue into the register read stage and does not create a fast wake-up signal. In the InO configuration the instruction in the second slot of the issue unit can only be issued if the first instruction issues. However, a false speculative issue would also trigger this and could cause the second micro-op to be issued before the final issue of the first instruction, breaking strict execution order.

For the other queue-based configurations, micro-ops are removed from the queue when they issue. This is opposed to the behavior of the default issue unit, that does not remove micro-ops that are valid in the current cycle. Instead, micro-ops that are issued set the state of their issue slot to invalid. This is canceled when the instruction was *poisoned*, i.e. it was woken up by a speculative load wake-up, and subsequently a load miss signal is received. The effect of this is that the load miss has no influence on most of the schedule logic. Because the queue based configurations remove issued instruction the issue has to be canceled. This is done by setting the valid signal to the issue unit to invalid. The issue unit only sets a ready signal if an instruction was issued. This ready signal determines if a micro-op is removed. Instead of only going into an issue slot, the load miss signal now travels through the issue logic and into the queues. There it affects the read addresses of the SRAMs, as these need to be incremented if an instruction is dequeued.

This could be resolved by not allowing speculative load wake-ups and thus increasing the load-use penalty from four to five cycles.

	InO		CASINO		LSC		DnB		OoO	
Core	79907	51017	82394	51801	86717	52530	94560	54538	92665	55854
Frontend	21088	18363	21088	18363	21700	18363	21700	18363	21088	18363
Branch Prediction	8563	10752	8563	10752	8563	10752	8563	10752	8563	10752
L1I Cache	635	638	635	638	635	638	635	638	635	638
Fetch Buffer	1385	777	1385	777	1385	777	1385	777	1385	777
ITLB	1475	1314	1475	1314	1475	1314	1475	1314	1475	1314
Fetch Target Queue	6080	3678	6080	3678	6692	3678	6692	3678	6080	3678
Frontend Rest	2950	1204	2950	1204	2950	1204	2950	1204	2950	1204
Backend	28685	16975	31182	17732	35019	18492	42730	20490	41542	21818
Decode	280	0	2F84	0	279	0	272	0	272	0
Integer Rename	6320	4225	6456	4225	6454	4225	6458	4225	6312	4225
Map Table	3240	2821	3240	2821	3240	2821	3240	2821	3240	2821
Free List	2270	1043	2272	1043	2270	1043	2270	1043	2271	1043
Busy Table	605	80	741	80	741	80	741	80	605	80
Integer Rename Rest	205	281	203	281	203	281	207	281	196	281
FP Rename	5228	3509	5193	3499	5195	3499	5193	3509	5116	3509
Map Table	2962	2496	2962	2496	2962	2496	2962	2496	2962	2496
Free List	1776	833	1661	833	1663	833	1661	833	1663	833
Busy Table	290	64	375	64	375	64	369	64	290	64
FP Rename Rest	200	116	195	106	195	106	201	116	201	116
Dispatch	0	0	2555	1043	4069	1246	4451	1501	6	0
Queue 1 (InQ, AQ, CRQ)	0	0	1635	598	2320	762	2156	747	0	0
Queue 2 (SQ, BQ, DLQ)	0	0	881	437	1669	476	2248	746	0	0
Dispatch Rest	0	0	39	8	80	8	47	8	6	0
Issue	834	280	887	0	879	0	7624	1718	13330	5121
Issue Mem	0	0	0	0	0	0	0	0	3177	1176
Issue Integer	0	0	0	0	0	0	0	0	7488	2680
Issue Float	0	0	0	0	0	0	0	0	2665	1265
Issue Unified	834	280	887	0	879	0	7624	1718	0	0
Integer Register File	3513	1497	3514	1505	3513	1505	3514	1505	3514	1497
Execute	3842	981	3840	976	3840	976	3846	981	3846	981
Control Status Register	2344	1515	2310	1515	2309	1515	2304	1515	2385	1515
Reorder Buffer	5215	4449	5014	4447	4950	4437	5534	4447	5608	4451
IBDA	0	0	0	0	2300	485	2300	485	0	0
RDT	0	0	0	0	1686	209	1686	209	0	0
IST	0	0	0	0	614	276	614	276	0	0
Backend Rest	1109	519	1129	522	1231	604	1234	604	1153	519
Load Store	19444	11723	19435	11751	19309	11720	19440	11728	19332	11716
Floating Point	10529	3928	10528	3927	10528	3927	10529	3929	10542	3929
Core Rest	161	28	161	28	161	28	161	28	161	28

Table 9: LUT (left) and flip-flop (right) utilization of different cores

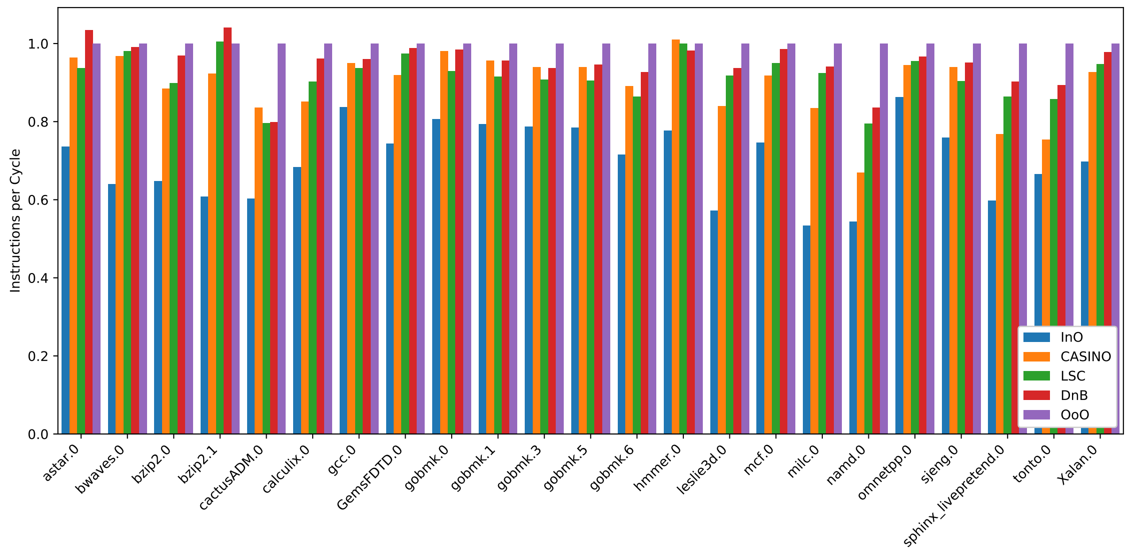
	InO		CASINO		LSC		DnB		OoO	
Core	1050	129	1234	137	1050	157	1050	157	1050	129
Frontend	419	66	419	66	419	66	419	66	419	66
Branch Prediction	260	46	260	46	260	46	260	46	260	46
L1I Cache	1	12	1	12	1	12	1	12	1	12
Fetch Buffer	0	0	0	0	0	0	0	0	0	0
ITLB	0	0	0	0	0	0	0	0	0	0
Fetch Target Queue	0	8	0	8	0	8	0	8	0	8
Frontend Rest	158	0	158	0	158	0	158	0	158	0
Backend	83	36	267	44	83	64	83	64	83	36
Decode	0	0	0	0	0	0	0	0	0	0
Integer Rename	0	0	0	0	0	0	0	0	0	0
Map Table	0	0	0	0	0	0	0	0	0	0
Free List	0	0	0	0	0	0	0	0	0	0
Busy Table	0	0	0	0	0	0	0	0	0	0
Integer Rename Rest	0	0	0	0	0	0	0	0	0	0
FP Rename	0	0	0	0	0	0	0	0	0	0
Map Table	0	0	0	0	0	0	0	0	0	0
Free List	0	0	0	0	0	0	0	0	0	0
Busy Table	0	0	0	0	0	0	0	0	0	0
FP Rename Rest	0	0	0	0	0	0	0	0	0	0
Dispatch	0	0	184	8	0	16	0	16	0	0
Queue 1	0	0	184	0	0	8	0	8	0	0
Queue 2	0	0	0	8	0	8	0	8	0	0
Dispatch Rest	0	0	0	0	0	0	0	0	0	0
Issue	0	0	0	0	0	0	0	0	0	0
Issue Mem	0	0	0	0	0	0	0	0	0	0
Issue Integer	0	0	0	0	0	0	0	0	0	0
Issue Float	0	0	0	0	0	0	0	0	0	0
Issue Unified	0	0	0	0	0	0	0	0	0	0
Integer Register File	0	36	0	36	0	36	0	36	0	36
Execute	83	0	83	0	83	0	83	0	83	0
Control Status Register	0	0	0	0	0	0	0	0	0	0
Reorder Buffer	0	0	0	0	0	0	0	0	0	0
IBDA	0	0	0	0	0	12	0	12	0	0
RDT	0	0	0	0	0	4	0	4	0	0
IST	0	0	0	0	0	8	0	8	0	0
Backend Rest	0	0	0	0	0	0	0	0	0	0
Load Store	155	15	155	15	155	15	155	15	155	15
Floating Point	121	12	121	12	121	12	121	12	121	12
Core Rest	272	0	272	0	272	0	272	0	272	0

Table 10: LUTRAM (left) and BRAM (right) utilization of different cores

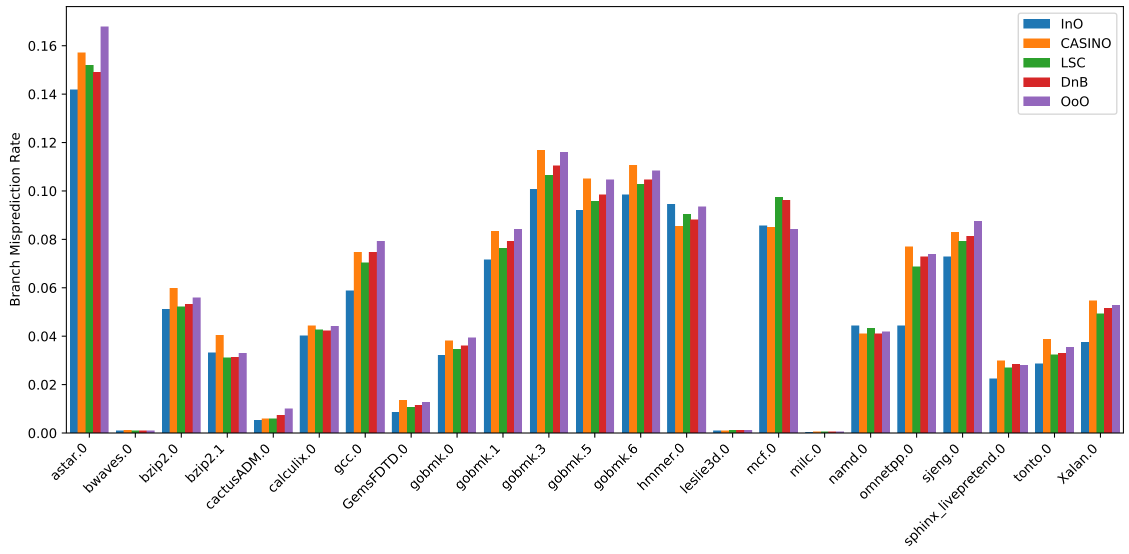
10.5 Application Performance

Figure 25a shows the IPC of the different cores normalized to OoO. On average InO has an IPC of 0.34 (68 %), CASINO 0.45 (90 %), LSC 0.46 (92 %), DnB 0.48 (96 %) and OoO 0.50. The percentage is normalized to OoO. DnB performs consistently close to OoO, and for astar and bzip2 it actually outperforms OoO. LSC and CASINO are close in performance on most benchmarks, but on average LSC is better. Figure 25b shows the branch misprediction rate for the different benchmarks. A peculiar observation is that CASINO has the worst branch misprediction rate which very well could contribute to it performing worse than LSC.

A general trend is that higher misprediction rates reduce the gap between InO and OoO, which makes sense since it would favor smaller instruction windows.



(a) IPC normalized to OoO



(b) Branch Misprediction Rate

Figure 25: Performance comparison of the different cores excluding fesvr call cycles.

11 Discussion

In this chapter we take a closer look at some key topics. We start by discussing challenges of implementing instruction queues in Chisel. Then we describe how our core implementations deviate from the proposed implementations and the implications of this. Then we discuss the performance we achieved using these techniques and set it into relation to the physical sizes of the implementations. Finally, we discuss issues we encountered while working with BOOM.

11.1 Implementing Instruction Queues in Chisel

One challenging aspect of implementing these queues was Chisels support for SRAMs. Micro-ops in BOOM are described as *Bundles*, the Chisel-equivalent of structs. Not all fields of the Bundle are used in every stage. The FIRRTL compiler usually removes superfluous entries, however it can only do this as long as they are separate wires and not concatenated, since Chisel does not support optimization at the bit-level but only signal-level. Chisel supports describing synchronous SRAMs using the *SyncReadMem* construct. The problem is that SyncReadMems internally concatenate their input data, in order to describe a single SRAM and not separate ones for each sub-signal. This disables the optimization for all micro-ops up until this point. Including debug signals, complete micro-ops are over 500 bits wide. This leads to a much too large frontend and huge memories.

Our solution to this problem was to create a white-list of relevant sub-signals of micro-ops and use this white-list to generate a heterogeneous vector containing only relevant signals. By storing this vector in the SRAM, instead of the full micro-op, we were able to reduce the width-requirement to 142 bits. The disadvantage of this approach is that debugging signals are filtered out and not available when it is in use. Additionally, changes in sub-signal usage have to be carefully monitored, and newly used signals have to be added to the white-list.

11.2 Core Deviations and Overheads

11.2.1 Overheads of InO Implementation

Apart from the general overhead introduced by the out-of-order nature of BOOM, such as register renaming, increased number of registers, the ROB and more complex LSU there is one major overhead. The dual issue design can issue two instruction per cycle. However, BOOM is built with an out-of-order issue unit that can issue to all of the issue units at the same time in mind. To enable a fair comparison between in-order BOOM and the other configurations performance-wise we also use two integer execution units, one memory and one floating point unit. This results in BOOM creating six read ports for the integer register file, even though only two instructions, using up to four read ports, can be issued. This could be solved by a design that doesn't assign fixed read ports to execution units, but instead has a fixed number of read ports whose results are then multiplexed to the right execution unit.

11.2.2 LSC

LSC is a stall-on-use in-order extended with register renaming to support limited out-of-order execution. Our implementation is based of the fully out-of-order core BOOM.

Size Overheads

Our implementation has significant size overheads in the LSU. Due to store-splitting and the in-order nature of BQ, LSC does not need memory ordering speculation. This is because all older loads and stores will have their addresses resolved, before a younger load does. In consequence the BOOM LSU, which contains a content addressable load queue, store queue and store buffer, is largely unnecessary.

The BOOM reorder buffer also introduces overhead compared to the simpler scoreboard that is proposed by Carlson et al. [1].

The BOOM frontend, which is built to support eight-wide instruction fetch and spans four cycles, is also unnecessary large for a simple design like LSC. However, in order to keep the designs comparable performance wise, we decided to leave the pipeline untouched apart from the dispatch and issue units.

Behavioral Deviation

The proposed LSC implementation has four functional units and two issue ports. These are what we call homogeneous, symmetric, issue ports, meaning that an issue port can issue to any of the functional units. This doesn't map well to BOOM, which has static heterogeneous issue ports. Each issue port is statically connected to one functional unit. To support four functional units and the possibility of issuing two ALU operations, our implementation has four issue ports. This means that our LSC core could issue up to four instructions if both AQ heads and BQ heads were ready and contained exactly two integer ALU instructions, one memory instruction and one floating point instruction.

Performance Deviation

LSC was reported to outperform in-order by 53 % and cover more than half of the performance gain of a comparable out-of-order [1]. Our results show a 31 % improvement over InO covering 92% of the OoO performance gain. I.e. both InO and LSC performed better, relative to OoO, than expected.

11.2.3 DnB

DnB is a slim, high-performing out-of-order design. Among our implementations, DnB was most suited to be based on an out-of-order core like BOOM. However, there are important deviations in our implementation.

Size Overheads

In terms of size overheads DnB is also affected by the use of static heterogeneous issue ports. The proposed design has four functional units, with two dedicated homogeneous issue ports for the issue queue (IQ) and two dedicated to the DLQ and CRQ. Our implementation has four functional units and four heterogeneous issue ports, divided among the IQ, DLQ and CRQ. Access to the issue ports are controlled by a priority MUX. This means that each issue slot is connected to all four issue ports, rather than just two.

Reducing the issue-width of the issue unit was stated as one of the main achievements in the original DnB paper [8]. After our experience with BOOM and a quick survey of commercial out-of-order designs [48], [68], [69], we concluded that homogeneous execution units are not common in out-of-order processors and thus this claim is questionable.

OoO uses three separate issue queues, each for each type of functional unit. Our DnB implementation has a unified issue queue for all functional units. A unified issue queue design has more complex issue select logic, as it considers all issue slots for all functional units. DnB does not suit separate issue queues well, since a reduced number of issue slots becomes much less flexible when spread across several queues.

BOOMs current implementation of the IQ is based on a design that is no longer used in commercial processors, since it is too expensive for large issue queues. It uses an age prioritized shifting IQ, and while this will give high performance and fair scheduling, it is very expensive [17].

Behavioral Deviation

BOOMs heterogeneous issue ports also has some unfortunate behavioral consequences for DnB. Since we cannot dedicate two issue ports to DLQ and CRQ and two to the issue slots, we use a priority MUX to share them between both DLQ, CRQ and the issue slots. The DLQs two heads are prioritized over the issue slots if they are urgent. In turn issue slots have priority over the two CRQ heads. This means that the issue slots could use all four issue ports if neither of the DLQ heads were ready. It also means that the CRQ has a lower priority than intended in the paper. This could potentially stall the CRQ heads too long and impact performance.

This could be remedied by giving the CRQ and DLQ exclusive access to an integer issue port. All resources in that port would have to be duplicated in the second ALU connected to an other port.

Performance Deviation

In their report, Kumar et al. reports that DnB achieves 95 % of the performance of the Baseline out-of-order [8]. This is aligned with our results which shows DnB getting 96 % of the OoO performance.

One should note that our DnB implementation uses a third of the issue slots that our baseline OoO uses. Kumar et al. [8] let DnB use half the issue slots compared to their Baseline. However, our DnB has the advantage of more flexible issue (up to four issues from issue slots), and a more flexible issue queue.

11.2.4 CASINO

CASINO is another stall-on-use in-order core extended with register renaming, to support limited out-of-order execution. CASINO also implements memory disambiguation to support speculation on memory ordering.

Size Overheads

Like our LSC implementation, the CASINO implementation has significant size overheads. The BOOM frontend is also excessive for CASINO.

CASINO also proposes an improved register renaming scheme which only renames the instructions issued from the SQ. This allows the proposed CASINO implementation to use only 16 additional registers in the physical register file. Our implementation has 48 additional register in the physical register file.

CASINO also proposes a novel LSU design to remove the Load Queue altogether, while still enabling memory ordering speculation. Our implementation uses the BOOM LSU, which should provide the same functionality, albeit at a higher cost.

Behavioral Deviation

CASINO is also affected from the heterogeneous issue ports. The proposed design has six functional units, two of each kind, and only two issue ports. This doesn't map well to BOOM. We decided to use four functional units, with two integer ALU and four issue port to implement a dual-issue-like design. Thus, our implementation could issue up to four instructions. Two from the SQ and two from InQ.

Performance Deviation

It is the CASINO implementation that deviates the most in terms of application performance compared to the numbers reported in the paper. CASINO is reported to outperform in-order by 51 % and come within 10 % of the out-of-order performance. CASINO is reported to have 23 % better performance than LSC.

Our results reverse this, with LSC performing 2 % better than CASINO. Both have around 30 % better performance than InO.

A reason for this could be that we use a different number of functional units and issue ports, than CASINO was simulated with in the report. The deviation between LSC's performance gain over in-order reported in CASINO paper and LSC paper is difficult to account for, as the configuration parameters they use for LSC are not reported in the CASINO paper.

11.3 Energy and Area-normalized Performance

LSC, CASINO and DnB all strive to give the highest area and energy normalized performance. Due to our FPGA-targeted synthesis we do not have a singular area measure, but rather a breakdown of the resource utilization of each core. Our power estimates are also unreliable, as they also are targeting an implementation of the core on an FPGA rather than an actual ASIC implementation, and moreover lack simulation based activation vectors. However, in terms of resource utilization there is an ordering of CASINO being the smallest, LSC being slightly bigger and DnB being largest, for all our size metrics. The only exception being LUTRAMs, used as a replacement for BRAMs due to CASINOs shorter SQ.

The size-ordering also matches the application performance ordering. We decided to use the more traditional Single Write IBDA instead of the smaller Fuzzy IBDA since it more closely reflects the original designs. Using Fuzzy IBDA would somewhat reduce LSCs overhead over CASINO. Drawing any further conclusions is difficult, but since CASINO has the greatest potential for further size reductions and is only very slightly behind LSC in terms of performance, we expect it to get most performance per area. Considering the simplicity of its core design its performance is impressive.

LSC is also a close contender and there are potentially larger reductions in the LSU and ROB possible than with CASINO. Combining early execution of load slices from LSC with the cheap memory ordering speculation from CASINO could be a promising way forward. This would reduce the load on the BQ, as store address slices no longer have to be put there.

We are least confident with DnBs ability to significantly improve performance per watt, since it requires essentially all parts of the out-of-order pipeline. The proposed reduction of issue width of DnBs issue unit would be problematic for modern cores with heterogeneous execution units. In our implementation it ends up bigger than the out-of-order reference core.

11.4 Release of BOOM v3

16th of May a major change to the BOOM pipeline was merged into the master branch of the BOOM repository on Github. This is part of the release of SonicBOOM and includes a rework of the frontend and improvement of the branch predictor. This was only a few weeks before the due-date of this thesis. While we are glad we were able to incorporate these changes it presented us with a few issues.

Most importantly, the new changes broke some of our FPGA infrastructure such that we no longer were able to boot Linux. This should be possible to fix, but given the proximity to our deadline we simply did not have time. Also due to the short time frame, we could not run the benchmarks enough times to establish confidence intervals for our results.

On the upside, the new branch-predictor gives us more confidence in our results since it enables wider speculation windows. This was a major limitation of our evaluation methodology using the old frontend. In some benchmark the performance of the new in-order baseline rivals the old out-of-order core.

12 Future Work

The next big step in the comparison of the different techniques would be to synthesize them targeting an ASIC process, resulting in more reliable area estimations. Combined with simulation based switching activity data this would also enable detailed power-analysis of the designs.

One major limitation for academic evaluation of the area of processor designs is the access to memory compilers. Since SRAMs tend to be very efficient they are commonly used for designs aiming to be power efficient. Examples of this are the RDT, IST and FIFO-based queues we explored as part of this work. While there have been attempts to develop accessible memory compilers such as OpenRAM [70] and the Educational Generic Memory Compiler [71], these still impose limits on the types and port-configurations of memories that can be created.

Furthermore, for this work we did not have access to commercial synthesis tools. While open-source synthesis tools such as yosys [72] are in active development, they can currently not fully replace commercial variants. Project like HAMMER [49] aim to make the synthesis process automated and simple. HAMMER also integrates ASAP7 [73], a modern predictive process design kit that emulates a 7 nm process. HAMMER seems to be a good path, since it is integrated into the Chipyard [13] framework we use to generate the SoC surrounding the different core variations. While HAMMER itself is open source, it requires commercial tools from cadence, synopsys and mentor. These tolls are integrated into HAMMER using plugins that are currently not publicly available. However, access to these plugins can be requested for academic use [74].

Another area that could improve is the selection of core parameters. We did not have enough time to do extensive parameter sweeps in order to find configurations that offer high performance at low implementation cost. BOOMs nature as a processor generator is well suited for this kind of automated exploration.

Furthermore, it would be beneficial to be able to evaluate the performance of BOOMv3 when running applications under Linux. While the proxy kernel eliminates operating system interference it also does not realistically represent all parts of the workload, since system calls are offloaded to the frontend server. This would require finding and eliminating the boot problems we currently experience. Linux testing could also be enhanced by integrating custom performance counters into the measurements we perform using perf.

The slightly lengthened critical path, in order to retain a 4-cycle load use in our designs, is also something that should be addressed. Reducing the overheads in the ROB and LSU for the Load Slice Core would provide an interesting continuation of our analysis of the overheads of BOOM over smaller in-order cores.

13 Conclusion

In this work we presented the RTL implementation and in-depth evaluation of LSC, DnB and CASINO, based on the RISC-V core BOOM. The focus of the work is on the instruction scheduling techniques employed by these three. Our results show InO (in-order baseline), CASINO, LSC and DnB with 68 %, 90 %, 92 % and 96 % of the performance of the out-of-order baseline. To estimate area we analyze the resource usage of the implementations when synthesized to an FPGA. It is not possible to represent the resource utilization with a single number, but our results clearly show CASINO needing less resources than LSC. Our DnB implementation needs even more resources than the out-of-order baseline. We also analyzed different instruction queue implementation, needed by the different cores, in terms of area efficiency.

Finally, our work also includes an in-depth analysis of Iterative Backwards Dependency Analysis (IBDA), which is used by both LSC and DnB. We propose Single Write IBDA and Fuzzy IBDA, which greatly reduces the required resources, with negligible loss in performance.

Bibliography

[1] Carlson, T. E., Heirman, W., Allam, O., Kaxiras, S., & Eeckhout, L. 2015. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 272–284. doi:10.1145/2749469.2750407.

[2] *BOOM Documentation*. URL: https://docs.boom-core.org/_/downloads/en/latest/pdf/.

[3] Dennard, R., Gaensslen, F., Yu, H.-N., Rideovt, V., Bassous, E., & Leblanc, A. 02 2007. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits Newsletter, IEEE*, 12, 38 – 50. doi:10.1109/N-SSC.2007.4785543.

[4] Hennessy, J. L. & Patterson, D. A. January 2019. A new golden age for computer architecture. *Commun. ACM*, 62(2), 48–60. doi:10.1145/3282307.

[5] Suggs, D., Bouvier, D., Clark, M., Lepak, K., & Subramony, M. 2019. AMD "zen 2". In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*, 1–24. IEEE. doi:10.1109/HOTCHIPS.2019.8875673.

[6] Taylor, M. B. 2012. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, Groeneveld, P., Sciuto, D., & Hassoun, S., eds, 1131–1136. ACM. doi:10.1145/2228360.2228567.

[7] Stamelakos, I., Xydis, S., Palermo, G., & Silvano, C. 01 2014. Variation-aware voltage island formation for power efficient near-threshold manycore architectures. 304–310. doi:10.1109/ASPDAC.2014.6742907.

[8] Alipour, M., Kaxiras, S., Black-Schaffer, D., & Kumar, R. 2020. Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, 424–434. IEEE. doi:10.1109/HPCA47549.2020.00042.

[9] Kumar, R., Alipour, M., & Black-Schaffer, D. 2019. Freeway: Maximizing MLP for slice-out-of-order execution. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, 558–569. IEEE. doi:10.1109/HPCA.2019.00009.

[10] Jeong, I., Park, S., Lee, C., & Ro, W. W. 2020. Casino core microarchitecture: Generating out-of-order schedules using cascaded in-order scheduling windows. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 383–396. doi:10.1109/HPCA47549.2020.00039.

[11] Zhao, J., Korpan, B., Gonzalez, A., & Asanovic, K. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. URL: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf.

[12] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., & Asanović, K. June 2012. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 1212–1221. doi:10.1145/2228360.2228584.

[13] Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y., Asanovic, K., & Nikolic, B. 05 2020. Chipyard: Integrated design,simulation, and implementation framework for custom socs. *IEEE Micro*, PP, 1–1. doi:10.1109/MM.2020.2996616.

[14] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., & Waterman, A. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[15] Zilles, C. & Sohi, G. 06 2000. Understanding the backward slices of performance degrading instructions. *ACM SIGARCH Computer Architecture News*, 28. doi:10.1145/339647.339676.

[16] Smith, J. E. April 1982. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3), 112–119. doi:10.1145/1067649.801719.

[17] Ando, H. 2018. Performance improvement by prioritizing the issue of the instructions in unconfident branch slices. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, 82–94. IEEE Press. doi:10.1109/MICRO.2018.00016.

[18] Crago, N. C. & Patel, S. J. 2011. Outrider: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 117–128, New York, NY, USA. Association for Computing Machinery. doi:10.1145/2000064.2000079.

[19] Moshovos, A., Pnevmatikatos, D. N., & Baniasadi, A. 2001. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 15th International Conference on*

Supercomputing, ICS '01, 321–334, New York, NY, USA. Association for Computing Machinery. doi:10.1145/377792.377856.

[20] Mitzenmacher, M. & Vadhan, S. P. 2008. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, Teng, S., ed, 746–755. SIAM. URL: http://dl.acm.org/citation.cfm?id=1347082.1347164.

[21] Shannon, C. E. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.

[22] Adams, R. D. *Multi-Port Memories*, 47–56. Springer US, Boston, MA, 2003. doi:10.1007/0-306-47972-9_3.

[23] Xilinx, Inc. *7 Series FPGAs Memory Resources*. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[24] Xilinx, Inc. *7 Series FPGAs Configurable Logic Block*. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[25] Laforest, C. E., Li, Z., O'rourke, T., Liu, M. G., & Steffan, J. G. September 2014. Composing multi-ported memories on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3). doi:10.1145/2629629.

[26] Patterson, D. A. & Hennessy, J. L. 2007. *Computer organization and design - the hardware / software interface (3. ed.)*. Morgan Kaufmann.

[27] Solihin, Y. 11 2015. *Fundamentals of Parallel Multicore Architecture*. doi:10.1201/b20200.

[28] Celio, C. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html.

[29] Liu, Y., Zhao, X., Jahre, M., Wang, Z., Wang, X., Luo, Y., & Eeckhout, L. 2018. Get out of the valley: Power-efficient address mapping for gpus. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, 166–179. IEEE Press. doi:10.1109/ISCA.2018.00024.

[30] Augot, D., Finiasz, M., & Sendrier, N. 01 2003. A fast provably secure cryptographic hash function. *IACR Cryptology ePrint Archive*, 2003, 230. URL: http://eprint.iacr.org/2003/230.

[31] Almeida, P. S., Baquero, C., Preguiça, N., & Hutchison, D. 2007. Scalable bloom filters. *Information Processing Letters*, 101(6), 255 – 261. doi:10.1016/j.ipl.2006.10.007.

[32] Kirsch, A. & Mitzenmacher, M. September 2008. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2), 187–218. doi:10.1002/rsa.20208.

[33] Snyder, W. Verilator, the fastest free verilog hdl simulator. (online). URL: https://www.veripool.org/wiki/verilator.

[34] Wulf, W. A. & McKee, S. A. March 1995. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1), 20–24. doi:10.1145/216585.216588.

[35] Hennessy, J. & Patterson, D. 2017. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann 8. Textbook.

[36] Kora, Y., Yamaguchi, K., & Ando, H. 2013. Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 37–48, New York, NY, USA. Association for Computing Machinery. doi:10.1145/2540708.2540713.

[37] Sembrant, A., Carlson, T., Hagersten, E., Black-Shaffer, D., Perais, A., Seznec, A., & Michaud, P. 2015. Long term parking (ltp): Criticality-aware resource allocation in ooo processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 334–346, New York, NY, USA. Association for Computing Machinery. doi:10.1145/2830772.2830815.

[38] Farrell, J. A. & Fischer, T. C. May 1998. Issue logic for a 600-MHz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5), 707–712. doi:10.1109/4.668985.

[39] Waterman, A. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

[40] Adve, S. V. & Gharachorloo, K. 1995. Shared memory consistency models: A tutorial. *IEEE Computer*, 29, 66–76. doi:10.1109/2.546611.

[41] Celio, C., Chiu, P.-F., Nikolic, B., Patterson, D. A., & Asanović, K. Sep 2017. Boom v2: an open-source out-of-order risc-v core. Number UCB/EECS-2017-157. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[42] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., & Wood, D. A. August 2011. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 1–7. doi:10.1145/2024716.2024718.

[43] Butko, A., Garibotti, R., Ost, L., & Sassatelli, G. 07 2012. Accuracy evaluation of gem5 simulator system. 1–7. doi:10.1109/ReCoSoC.2012.6322869.

[44] Carlson, T., Heirman, W., & Eeckhout, L. 11 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. 52. doi:10.1145/2063384.2063454.

[45] Genbrugge, D., Eyerman, S., & Eeckhout, L. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 1–12. IEEE. doi:10.1109/HPCA.2010.5416636.

[46] Hill, M. D., Christie, D., Patterson, D., Yi, J. J., Chiou, D., & Sendag, R. July 2016. Proprietary versus open instruction sets. *IEEE Micro*, 36(4), 58–68. doi:10.1109/MM.2016.61.

[47] Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., & Bachrach, J. 2017. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 209–216. doi:10.1109/ICCAD.2017.8203780.

[48] Greenhalgh, P. big.little processing with arm cortex-a15 & cortex-a7. Technical report, 2012. URL: https://web.archive.org/web/20131017064722/http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf.

[49] Wang, E., Izraelevitz, A., Schmidt, C., Nikolic, B., Alon, E., & Bachrach, J. 2018. Hammer: Enabling reusable physical design. In *Workshop on Open-Source EDA Technology (WOSET)*. URL: https://woset-workshop.github.io/PDFs/a27.pdf.

[50] Shioya, R., Goshima, M., & Ando, H. 2014. A front-end execution architecture for high energy efficiency. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 419–431. doi:10.1109/MICRO.2014.35.

[51] Ros, A. & Kaxiras, S. 10 2018. The superfluous load queue. 95–107. doi:10.1109/MICRO.2018.00017.

[52] Xilinx, Inc. *Vivado Design Suite User Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf.

[53] Xilinx, Inc. *Vivado Design Suite User Guide - Power Analysis and Optimization*, ug907 (v2018.2) edition, June 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug907-vivado-power-analysis-optimization.pdf.

[54] Xilinx, Inc. *Zynq-7000 SoC Technical Reference Manual*. URL: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[55] *Chipyard Documentation*. URL: https://readthedocs.org/projects/chipyard/downloads/pdf/latest/.

[56] Beamer, S., Karandikar, S., & Mao, H. Rocket chip on zynq fpgas. URL: https://github.com/riscv-boom/fpga-zynq.

[57] Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R., Bachrach, J., & Asanovi´ c

, K. 2018. 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, 29–42, Piscataway, NJ, USA. IEEE Press. doi:10.1109/ISCA.2018.00014.

[58] Xilinx, Inc. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf.

[59] Biancolin, D., Karandikar, S., Kim, D., Koenig, J., Waterman, A., Bachrach, J., & Asanovic, K. 2019. Fased: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, 330–339, New York, NY, USA. Association for Computing Machinery. doi:10.1145/3289602.3293894.

[60] Magyar, A., Biancolin, D., Koenig, J., Seshia, S., Bachrach, J., & Asanovic, K. 2019. Golden gate: Bridging the resource-efficiency gap between asics and FPGA prototypes. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, Pan, D. Z., ed, 1–8. ACM. doi:10.1109/ICCAD45719.2019.8942087.

[61] Waterman, A., Asanović, K., & others. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, June 8, 2019. URL: https://riscv.org/specifications/privileged-isa/.

[62] The peformance problem of boom (online). URL: https://groups.google.com/forum/#!topic/riscv-boom/LVhm8vPZbW0.

[63] Waterman, A., Lee, Y., & Terpstra, W. W. Risc-v proxy kernel and boot loader. URL: https://github.com/riscv/riscv-pk.

[64] Waterman, A. & others. Risc-v frontend server (online). URL: https://github.com/riscv/riscv-fesvr.

[65] Karandikar, S., Mao, H., Mao, H., & Gonzalez, A. Firemarshal. URL: https://github.com/firesim/FireMarshal.

[66] Celio, C., Dabbelt, P., & Waterman, A. Speckle: A wrapper for the spec cpu2006 benchmark suite. URL: https://github.com/ccelio/Speckle.

[67] Henry Cook, W. T. & Lee, Y. Oct 2017. Diplomatic design patterns: A tilelink case study. In *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*. URL: https://carrv.github.io/2017/papers/cook-diplomacy-carrv2017.pdf.

[68] Grayson, B. e. a. 2020. Evolution of the samsung exynos cpu microarchitecture. ISCA. URL: https://www.iscaconf.org/isca2020/papers/466100a040.pdf.

[69] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[70] Guthaus , M. R., Stine , J. E., Ataei , S., Brian Chen , Bin Wu , & Sarwar , M. Nov 2016. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1–6. doi:10.1145/2966986.2980098.

[71] Goldman, R., Bartleson, K., Wood, T., Melikyan, V., & Babayan, E. 2014. Synopsys' educational generic memory compiler. In *10th European Workshop on Microelectronics Education (EWME)*, 89–92. doi:10.1109/EWME.2014.6877402.

[72] Shah, D., Hung, E., Wolf, C., Bazanski, S., Gisselquist, D., & Milanovic, M. 2019. Yosys+nextpnr: an open source framework from verilog to bitstream for commercial fpgas. *CoRR*, abs/1903.10407. URL: http://arxiv.org/abs/1903.10407, arXiv:1903.10407.

[73] Clark, L. T., Vashishtha, V., Shifren, L., Gujja, A., Sinha, S., Cline, B., Ramamurthy, C., & Yeric, G. 2016. Asap7: A 7-nm finfet predictive process design kit. *Microelectronics Journal*, 53, 105 – 115. doi:https://doi.org/10.1016/j.mejo.2016.04.006.

[74] *Hammer Documentation*. URL: https://hammer-vlsi.readthedocs.io/_/downloads/en/latest/pdf/.

A Manual: Running RISC-V SoCs on a Zynq FPGA

This manual describes how to generate FPGA bitstreams based on Chipyard designs Our implementation is based on riscv-boom/fpga-zynq, a fork of the deprecated ucb-bar/fpga-zynq. It utilizes the Chipyard infrastructure to generate Verilog and synthesizes it targeting the Xilinx Zynq ZC706 board.

A.1 Installation

The installation consists of Xilinx Vivado 2018.2, a customized Chipyard version and the fpqa-zynq fork.

A.1.1 Vivado

Vivado 2018.2 can be found here. It requires a Xilinx account to download and install.

Installation

In order for the installation to finish under Ubuntu 18.04 `libtinfo5` needs to be installed as described here. During the installation the Vivado HL System Edition and support for at least 7000 series FPGAs should be chosen. The free WebPACK edition is not compatible with larger FPGAs such as the XC7Z045e we use. In order to be able to target it the free 30-day evaluation license should be chosen during installation. This evaluation license does not support bitstream generation and can only be used for testing. This manual assumes that Vivado is installed under `~/Xilinx/Vivado/2018.2`. In order to be able to flash bitstreams from Vivado and debug using waveforms collected on the FPGA, drivers need to be installed. This can be done by executing:

```
cd ~/Xilinx/Vivado/2018.2/data/xicom/
sudo ./install_drivers
```

Once Vivado is installed licensing has to be set up.

License

Connecting to the NTNU license server requires either being in the NTNU network (e.g. via eduroam) or using a VPN. In order to connect to the NTNU license server
`export XILINXD_LICENSE_FILE=27000@xilinx.lisens.ntnu.no` needs to be added to the bashrc (`~/.bashrc`). This sets the environment variable `XILINXD_LICENSE_FILE` for bash sessions. A side effect of this is that Vivado will not find a license when started from the application launcher and has to be started from a terminal. In order to troubleshoot issues it is possible to manually check if the connection to the license server works using lmutil. Lmutil is located in
`~/Xilinx/Vivado/2018.2/bin/unwrapped/lnx64.o`.
`./lmutil lmstat -a -c 27000@xilinx.lisens.ntnu.no` can be used to check if the connection

to the license server works and lists available licenses. `./lmutil lmstat -a` additionally checks if `XILINXD_LICENSE_FILE` is correctly set. If the commands complain that the file lmutil can't be found even though ls lists it a symlink for a shared object needs to be created first using `ln -s /lib64/ld-linux-x86-64.so.2 /lib64/ld-lsb-x86-64.so.3`.

A.1.2 Chipyard

The installation of Chipyard mostly matches the instructions found here but uses our fork that adds the zynq generator. The dependencies Chipyard requires on Ubuntu 18.04 can be installed using `chipyard-deps.sh`. This manual assumes that Chipyard is installed to `~/git/chipyard` as follows:

```
cd ~/git
git clone https://github.com/EECS-NTNU/chipyard.git
cd chipyard
# check out right branch
git checkout thesis-final
# initialize submodules
./scripts/init-submodules-no-riscv-tools.sh
# build toolchains
./scripts/build-toolchains.sh
```

Afterwards the installation can be tested:

```
# go to the verilator simulation folder
cd ~/git/chipyard/sims/verilator/
# source the chipyard environment (RISC-V toolchain)
source ~/git/chipyard/env.sh
# create a verilator emulator for small boom
make -j CONFIG=SmallBoomConfig
# run the test and benchmark suite
make -j run-fast CONFIG=SmallBoomConfig
```

`source ~/git/chipyard/env.sh` needs to be executed whenever the RISC-V toolchain is needed.

A.1.3 Fpga-zynq

Our fork of fpga-zynq can be installed as follows:

```
cd ~/git
git clone https://github.com/davidmetz/fpga-zynq.git
cd fpga-zynq/
# check out right branch
git checkout thesis-final
git submodule init
git submodule update
```

To use it a terminal with a VPN connection is required.

```
cd ~/git/fpga-zynq/zc706_MIG/
source ~/Xilinx/Vivado/2018.2/settings64.sh
source ~/Xilinx/SDK/2018.2/settings64.sh
source ~/git/chipyard/env.sh
# make uboot once to get the tools needed to close the ramdisk
# needs libssl-dev
make arm-uboot
make boot-bin CONFIG=WithL2TLB_WithL2Cache_With1GbRam_SmallBoomZynqConfig
```

`make arm-uboot` only needs to be executed once in order to generate the tools required to modify the ramdisk for the ARM core.

FireMarshal

FireMarshal is the tool used by FireSim to generate RISC-V Linux images.

```
cd ~/git
git clone https://github.com/firesim/FireMarshal.git
cd FireMarshal
# install python setuptools & wheel
sudo apt-get install python3-setuptools
pip3 install wheel
# install the list of requirements
pip3 install -r python-requirements.txt
# initalize submodules
./init-submodules.sh
# get RISC-V toolchain provided by chipyard
source ~/git/chipyard/env.sh
# build basic linux image
./marshal -v build workloads/br-base.json
```

The linux image generated can be found in the folder `images`. To run it on the quemu jit-recompiler run `./marshal -v launch workloads/br-base.json`. The username is `root` and the password is `firesim`.

A.1.4 Zynq Board

The Zynq board boots from an SD card. It boots into Linux using a ramdisk. This means that changes made to the filesystem are not persistent after a reboot. In order to retain files they need to be stored to the sdcard that can be mounted at `/sdcard` by running the script `~/init-sd-ssh.sh` on ARM Linux.

SD Card

The SD card needs to be FAT-32 formatted and contain the following files:

- `BOOT.bin` - Xilinx specific boot binary containing the FSBL, FPGA bitstream and U-Boot.
- `devicetree.dtb` - Device tree binary describing the layout of the Zynq SoC and connected peripherals.
- `uImage` - Linux Kernel image in U-Boot specific format.
- `uramdisk.image.gz` - Filesystem image used for the ramdisk of ARM Linux.

The files are the same for both zc706 and zc706_MIG and can be found in

`~/git/fpga-zynq/zc/06/fpga-images-zc706`.

Ethernet Connection

The board needs to be connected directly to a PC via Ethernet. It uses the static IP `192.168.1.5`. In order to connect to it the Ethernet port it is connected to needs to use the following settings:

- address 192.168.1.1
- netmask 255.255.255.0

This can be done for the network adapter `enx00050210001c` using the command
`sudo ifconfig enx00050210001c 192.168.1.1 netmask 255.255.255.0 up`. The network settings of the board can be changed by modifying the file `etc/network/interfaces` in the ramdisk. The Zynq board should *not* be connected to a larger network or the internet since it uses an outdated kernel and very poor login credentials.

U-Boot

The Zynq SoC boots Linux by first running an initialization program from an internal ROM. It then executes the First Stage Boot Loader (FSBL) contained in `BOOT.bin` and flashes a bitstream to the FPGA. The FSBL then starts U-Boot. U-Boot reads a configuration from the QSPI flash on the Zynq board. This can be used to configure how much RAM Linux uses in order to reserve some when using zc706 instead of zc706_MIG. A wrong configuration in the QSPI flash can also prevent Linux from booting. In order to diagnose U-Boot or change the configuration a mini USB cable needs to be connected to the UART port of the board. `sudo screen /dev/ttyUSB0 115200` can then be used observe the U-Boot output. U-Boot waits for a user input that interrupts the normal boot progress and enters the U-Boot shell. The following commands configure uboot to boot from the files on the SD card using 1 GB of RAM:

```
setenv bootargs "console=ttyPS0,115200␣earlyprintk␣mem=1024M"
setenv sdboot "echo␣Copying␣Linux␣from␣SD␣to␣RAM...␣&&␣fatload␣mmc␣0␣${
    ↪ kernel_load_address}␣uImage␣&&␣fatload␣mmc␣0␣${devicetree_load_address}␣
    ↪ devicetree.dtb␣&&␣echo␣Copying␣ramdisk...␣&&␣fatload␣mmc␣0␣${
    ↪ ramdisk_load_address}␣${ramdisk_image}␣&&␣bootm␣${kernel_load_address}␣${
    ↪ ramdisk_load_address}␣${devicetree_load_address}"
saveenv
```

Cooling

The default fan used by the Zynq board runs at a very high speed due to its small size, generating a lot of high-pitched noise. The fan speed can be reduced using PWM, however the cooling provided by the small fan at lower speeds might not be sufficient. In order to keep both the heat and noise low it is advisable to connect a 120 mm PC fan to the Zynq board instead and direct its airflow to the SoC. Its speed can be reduced by configuring the fan speed using fesvr-zynq.

Passwordless SSH

Passwordless SSH enables easier transfers using scp. In order to enable it a key pair needs to be generated and added to the authorized keys in the ARM ramdisk image:

```
# create ssh keypair:
ssh-keygen -t rsa
# use default options
cd ~/git/fpga-zynq/zc706
make ramdisk-open
# append ssh key to authorized keys
cat ~/.ssh/id_rsa.pub >> ramdisk/home/root/.ssh/authorized_keys
make ramdisk-close
# remove ramdisk mount folder
sudo rm -rf ramdisk
```

The updated ramdisk can then be changed either by copying it directly to the SD card or by transferring it via scp. The latter requires that `./init-sd-ssh.sh` was run on ARM linux. It can be transferred via scp using the command
`scp fpga-images-zc706/uramdisk.image.gz root@192.168.1.5:/sdcard/.`

A.2 Running Applications

To generate the bitstreams for BOOM v3 used in this thesis run the following commands:

```
cd ~/git/fpga-zynq/zc706_MIG/
source ~/Xilinx/Vivado/2018.2/settings64.sh
source ~/Xilinx/SDK/2018.2/settings64.sh
source ~/git/chipyard/env.sh
# generate bitstreams
make boot-bin CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithMediumBooms_zynq.MediumBoomZynqConfig
make boot-bin CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithMediumInoBooms_zynq.MediumBoomZynqConfig
make boot-bin CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumSliceBooms_zynq.
    ↪ MediumBoomZynqConfig
make boot-bin CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumCasBooms_zynq.
    ↪ MediumBoomZynqConfig
make boot-bin CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumDnbBooms_zynq.
    ↪ MediumBoomZynqConfig
```

After `./init-sd-ssh.sh` was run on ARM linux, these can be transferred:

```
make transfer CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithMediumBooms_zynq.MediumBoomZynqConfig
make transfer CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithMediumInoBooms_zynq.MediumBoomZynqConfig
make transfer CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumSliceBooms_zynq.
    ↪ MediumBoomZynqConfig
```

```
make transfer CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumCasBooms_zynq.
    ↪ MediumBoomZynqConfig
make transfer CONFIG=WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithQueuePerfCounters_boom.common.WithMediumDnbBooms_zynq.
    ↪ MediumBoomZynqConfig
```

The bitstreams are transferred into the home directory of the ZC706 ARM Linux. There they can be flashed onto the FPGA using the script `~/flash-bitstream.sh`:

```
~/flash-bitstream.sh WithBenchmarkConfig_zynq.WithZynqConfig_boom.common.
    ↪ WithMediumBooms_zynq.MediumBoomZynqConfig.bit
```

Bare metal applications can be executed directly:

```
~/fesvr-zynq hello
```

Executes the application `hello`. This test is also performed by `~/flash-bitstream.sh`. To run an application requiring an operating system, the proxy-kernel is used:

```
~/fesvr-zynq ~/pk -s application -args
```

The `-s` option of the proxy-kernel enables collection of performance counter data.

Erling Rennemo Jellum, David Metz

NTNU
Norwegian University of
Science and Technology