Optimization of RISC-V BOOM Core for General Applications

Aniket Adhikari
Department of Computer Science
Virginia Tech
Virginia, U.S.

***Abstract*—Given the varying needs for instruction sets in different industries, there are a number of ways in which their implementations need to be parameterized. Using the the Berkley-Out-of-Order Machine (BOOM) core, an implementation of the RISC-V ISA, a number of alterations were made on its parameters to create an ISA configuration for general purpose applications. This done by using various benchmarks including Dhrystone and matrix operations and measuring cycles per instruction and instructions per cycle. Decode and fetch width were found to be a very important factor in performance and throughput as well as the type of branch predictor used.**

***Index Terms*—RISC-V, Computer Organization, Out-of-Order Execution**

I. Introduction

Global demand for microprocessors continues to increase, likely as a result of growing acceptance of smartphones and tablets as well as implementation of cloud computing. Additionally, certain industries such as automotive, banking, aerospace and defense, medicine, and others are contributing to this growing demand [8].

New instruction set architectures (ISAs) are created and implemented, such as RISC-V, to fit the requirements of various industries, whether it be cost, energy, or performance. The Berkeley Out-of-Order Machine (BOOM) is a high-performance, synthesizable superscalar core implementation of the RISC-V ISA. The core offers a number of different parameters that can be manipulated, such as branch prediction, number of instructions fetched (or decoded) per cycle, number of registers for different operations, and more [6] [7], giving it strong flexibility for optimization. The parameters of the BOOM core would be altered for reasons such as improved performance, efficient resource usage, improved scalability, cost reduction, or reduced energy consumption. However, the requirements for optimization generally need to be holistic, meaning it has a mix of improvements.

Overall, It is critical to modify configurations of ISA cores, such as the BOOM core, in order to meet specific requirements of diverse computing applications and environments. For my project, I am proposing to create an optimized BOOM core configuration for general applications through fine-tuning of parameters and settings.

ECE 5504: Computer Architecture

II. Related Work

A. A Highly Productive Implementation of an Out-of-Order Processor Generator

The technical report for the BOOM core [4] explains that the out-of-order machine is required to overcome Moore's Law, which has made it increasingly challenging for the computing industry to improve performance. Additionally, there is an increasing need for Warehouse Scale Computing (WSC), which is relevant in compute-intensive areas such as search, media delivery, gaming, shopping, and more, as well as Internet of Things (IoT), which is meant to be more cost efficient. The purpose of the BOOM core is primarily for the advancement in computer architecture research as well as education, research, and industry, but it is to also show how superscalar machines can be parameterized to fit industry-specific needs. Through use of the RISC-V ISA and Chisel hardware construction language, the BOOM core is industry-competitive in terms of branch prediction and can be parameterized for different workflows. By leveraging open-source artifacts like the RISC-V ISA, the core was implemented using only 16,000 lines of code. The Towards the end of the report, there is a section about future work that talks about users being able to configure BOOM in millions of unique ways. However, it is important to be careful when actually deciding on what design configurations to pursue because of how expensive it can be to determine configurations for end-users. They also mention how performance improvements in BOOM can be explored, such as data-prefetching or more aggressive instruction fetch (IF) unit.

B. BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation

This paper from Celio et al. [3] presents the Berkeley resilient out-of-order machine, which is a resilient wide-voltage-range implementation of the BOOM Core. The BROOM Core is designed in 28nm technology and enhances BOOM with architectural techniques to enable operation at low voltages. BROOM consists of the out-of-order BOOM core along with a 1MB L2 cache, each in separate voltage/clock domains. The paper discusses optimizations made to the initial BOOM core design to improve performance when implemented in a 28nm application-specific integrated circuit (ASIC) flow, rather than simply as synthesizable register-transfer level (RTL). This

included redesigning structures like the register file, branch predictor, and instruction issue queues identified as critical paths. Additional reliability techniques were also implemented in the L2 cache to enable low voltage operation. The resulting test chip with the evolved BOOM core tapes out at 1 GHz in 28nm technology, and can operate down to 0.47V. The productivity benefits of using open-source tools like the Chisel hardware construction language and Rocket Chip SoC generator are also discussed. Christopher Celio, one of the authors of this paper, is also a contributor to the creation of the BOOM Core, making this paper additionally relevant. This paper did not really play a role in the optimization of my BOOM Core instance but it was interesting to read.

C. Machine Learning Power Models for CPU Microarchitecture Design

The paper from Kumar et al. [5] presents a machine learning based approach for microarchitecture-level power modeling of complex CPUs. The goal is to develop models that can accurately estimate power consumption at a fine-grained spatial (sub-component) and temporal (cycle-level) granularity. The approach extracts features from high-level microarchitecture simulation activity traces and trains machine learning regressors to correlate features to power consumption for different sub-blocks. Sub-block models are then composed hierarchically into full core models. A systematic methodology is introduced for feature selection and engineering to model common CPU structures based on categorization of signals into data, control, mixed, and advanced signals. Techniques are developed specifically for modeling aspects like pipelining, gating, and glue logic. The approach is demonstrated by developing power models for an in-order RISC-V core called RI5CY) and the BOOM Core. Cross-validation shows the models can predict cycle-level power with 2.2% (RI5CY) and 2.9% (BOOM) mean absolute error compared to gate-level estimation. BOOM core models trained only on micro-benchmarks are also shown to predict cycle power for real applications like CoreMark and FFT with less than 3.6% error. The models can also estimate average power and peak power accurately. In summary, the paper shows machine learning based power modeling at microarchitecture level can achieve accuracy comparable to lower level techniques but with significantly lower modeling effort.

III. Method

This sections discusses what parameters of the BOOM core were altered and why they were selected for configuration.

In my initial proposal, there were a number of parameters I wanted to alter for my configuration. This included the microarchitecture parameters, memory hierarchy, and branch predictor. While it is possible to alter these parts of the BOOM core, I found that only some of these parameters were able to be altered without sacrificing dependent parameters. I was able to modify the following parameters:

- `decodeWidth`: The number of instructions that can be decoded in a single clock cycle. Essentially, it determines how many instructions can be taken from the instruction queue and prepared for execution. Increasing this allows for more instructions to be be decoded per cycle, thus improving performance. However, it requires larger and more complex decode and rename logic. There are also diminishing returns as overhead starts dominating.
- `fetchWidth`: The number of instructions that can be fetched in a single clock cycle. Essentially, it determines how many instructions can be taken from the instruction cache or memory. Increasing this allows for more instructions to be fed into the pipeline, but there are costs for a larger instruction cache and BTB needed to support wider fetches. It also needs to be accompanied by a relatively large `decodeWidth`.
- `numOfROBEntries`: The number of entries in the reorder buffer (ROB), which is an essential to performing out-of-order execution. It stores instructions in the order they are fetched and reorders them for efficient execution. Increasing this enables more instructions to execute out-of-order leading to higher performance, but it requires larger ROB storage and increases latency for mispredict recovery. More entries also increase power consumption.
- `numPhysRegisters`: The number of physical integer and floating-point registers available for register renaming. These registers store data and intermediate results during computation. Additional registers reduce stalls and increase ILP. But they require larger register files and longer read/write ports, consuming more power and chip area.
- `numLdqEntries`: The number of entries available in the load queue, which manages loads from memory. It tracks pending loads and ensures correct ordering and dependencies. Increasing load queue size reduces load queue full stalls to improve performance. But queuing more loads requires larger load queue buffers and control logic.
- `numStqEntries`: The number of entries available in the store queue, which manages stores or writes to memory. It tracks pending stores and ensures proper memory ordering. More store queue entries reduces blocking between stores and later loads. However, stire queue size can't be very large as it is accessed associatively. Larger queues also increase power.

The emerging trend here is that increasing the value for any one of these parameters leads to improved performance with trade-offs in increased hardware cost and increased power consumption. Recall that the aim of this project is to optimize for general-purpose applications. As a result, we want to make sure the BOOM Core is fine-tuned so that it doesn't consume too much power or hardware space while simultaneously keeping performance and throughput in mind.

Additionally, I looked at different branch predictor options:

- **Tagged Geometric Length Predictor (TAGE)**: Employs multiple tables with different history lengths. It's designed to improve prediction accuracy by utilizing

different tables based on the patterns in the branch history. It dynamically selects the best predictor based on the history of branches, using a tagged mechanism to identify which tables are likely to make accurate predictions for different branches [6].

- **Alpha**: Utilizes a combination of several predictors, including local, global, and tournament predictors. The local predictor uses a history table to predict branches based on the history of a specific branch instruction. The global predictor uses a history register to predict branches based on global history patterns. The tournament predictor selects between the local and global predictors based on their performance on previous predictions [6].
- **Gshare**: Utilizes a hash of the global branch history and the branch program counter (PC). It XORs the global history with the lower bits of the PC to index into a table, which stores the prediction for a particular branch. Relatively simple, making it efficient in terms of hardware implementation while offering pretty accurate branch predicion [6].

IV. Evaluation Plan

Below I have included the benchmarks and metrics that I have used as well as their overall purpose in the context of the project.

A. Benchmarks

There are 8 benchmarks used for my project. While this is certainly a lot, I believe it paints a clear picture of general performance:

1) **Dhrystone**: Measures the efficiency and performance of computer systems, specifically on integer computing capabilities [9]. This benchmark has a place in this project since it is readily available to us as students. However, it being almost 40 years old forced me to consider other benchmarks in addition to Dhrystone.
2) **Median**: Performs a 1D three element median filter. It assesses how quickly a system or algorithm can calculate the median value within a dataset. It's a measure of sorting and algorithm efficiency. It primarily measures general CPU performance, especially integer arithmetic and string handling operations. It's an older benchmark and might not fully represent modern computing workloads, but it's useful for a broad evaluation of processor performance.
3) **QSort/RSort**: Uses quicksort to sort an array of integers. The difference is that QSort sorts in ascending order and RSort sorts in descending order. Both are good for measuring recursion performance.
4) **Towers of Hanoi**: Towers of Hanoi is a classic puzzle problem. It evaluates recursion performance and the ability of a system to handle recursive operations.
5) **Vector-vector Add Benchmark**: Measures the speed of adding two vectors together element-wise. This is relevant for assessing the performance of vector processing or SIMD (Single Instruction, Multiple Data) capabilities in hardware.
6) **Memory Block Copy** Assesses memory copying speed. It's about measuring how fast data can be transferred from one location to another within memory. This benchmark is crucial for evaluating memory performance.
7) **Sparse Matrix-Vector Multiplication** Measures how efficiently a system can perform operations involving sparse matrices and vectors. It's relevant in scientific computing and certain types of data analysis where sparse matrices are common.

B. Metrics

- **Cycles per Instruction (CPI)**: CPI is a crucial performance indicator to evaluate the efficiency of a processor in executing instructions. Low CPI generally equates to better performance. I compared the CPI of different configurations of the BOOM core when they were run on the previously mentioned benchmarks
- **Instructions per Cycle (IPC)**: IPC is the inverse cycles per instruction and measures the throughput for a processor. High CPI generally equates to better performance as it indicates more instructions are being executed per cycle. This is especially important for the BOOM core since it is superscalar, meaning it has the ability to pipeline more than one instruction per cycle. I compared the IPC of different configurations of the BOOM core when they were run on the previously mentioned benchmarks.

V. Evaluation Results

Testing was performed over the course of two weeks for various configurations of the BOOM Core using the benchmarks that were previously mentioned. Finding the benchmarks was fairly challenging as I had originally wanted to use the SPEC or CoreMark benchmarks for RISC-V but found it difficult to port into my instance. I got close to using CoreMark but was hindred by the lack of a high-performance monitor (HPM). I even had troubles with the benchmarks I ended up using because I wanted to include a HPM to measure cache performance but could not figure out how to port it or create one from scratch. After settling on benchmarks, I began using them to test the `SmallBoomConfig`, `MediumBoomConfig`, and `LargeBoomConfig`, which were all provided by the Chipyard framework from Berkeley [2]. For each of these configurations, I measured the performance in CPI and throughput in IPC and found that the `LargeBoomConfig` performed the best. The next step was to look at the branch predictor options of TAGE, Alpha, and Gshare. This was done by creating new configurations that combined elements of the `LargeBoomConfig` and the respective branch predictor. This allowed for uniformity in the results because the underlying parameters remained the same and the only part of the configuration that is changed is the branch predictor. The results showed that the TAGE predictor performed vastly

better than the Gshare and Alpha predictors in terms of performance and throughput. These configurations were fairly easy to run on each of the benchmarks. The only problem with running these configurations was how time-consuming issuing a `make` command for each configuration was. It was not something I had originally factored into my timeline and became increasingly problematic when I began creating my own configurations.

To create my own configurations, I took the `LargeBOOMConfig` and made changes to the parameters mentioned in Section III. I came up with 2 different configurations using the TAGE predictor which I labeled as `AniketBoomConfigLargeReduced` and `AniketBoomConfigLargeIncreased` as shown in Fig. 1 and Fig. 2. With `AniketBoomConfigLargeReduced`, or reduced configuration, I reduced the number of ROB entries, load queue entries, and store queue entries but also increased the number of integer and floating point registers. The performance for both seemed almost exactly the same, down to the number of cycles. As a result, it was necessary to create another configuration, `AniketBoomConfigLargeIncreased`. The biggest change made to this configuration when compared to the `LargeBoomConfig` and the reduced configuration is decode width being changed from 3 to 4. This greatly increased performance and throughput when measured against the benchmarks because more instructions were able to be decoded.

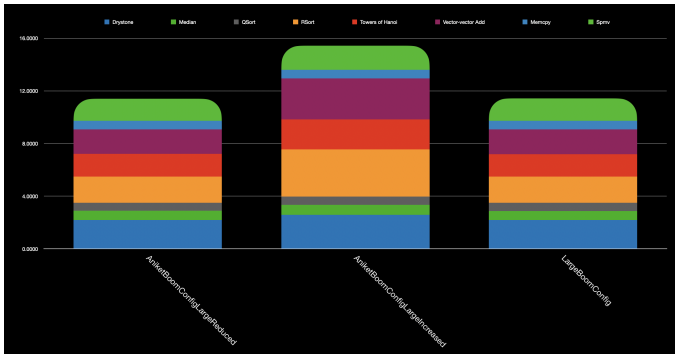

Fig. 1. Throughput Results for Different BOOM Configurations Using Instructions Per Cycle

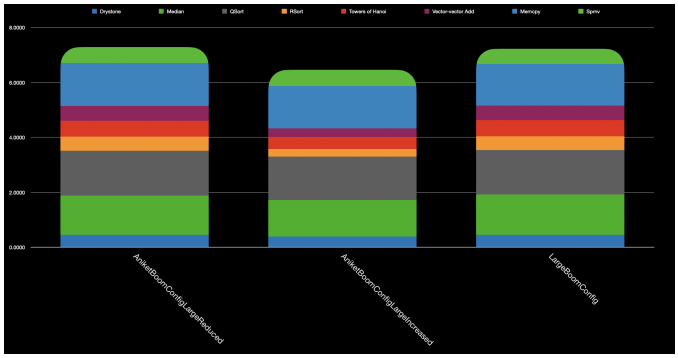

Fig. 2. Performance Results for Different BOOM Configurations Using Cycles Per Instruction

The drawback I experienced with both of these configurations was how long it took to build them using `make`. This is the result of increased overhead. That being said, it was interesting to see how important decode width is when measuring performance and throughput. Also, I did not expect reducing physical registers to have such a little effect on CPI and IPC. When factoring in the initial tests on the other BOOM configurations with varying sizes of fetch width, it's clear that both the fetch width and decode width play a pivotal role in performance. They also negatively influence the amount of overhead. It also seems like the number of physical registers and ROB entries begin to have diminishing returns at some point since they continue to take up physical space upon hardware with little improvements in performance.

VI. Timeline

My original timeline involved:

- 1 week of evaluating feedback from instructors about proposal and research about parameters of ISAs in general that might be important
- 2 weeks of stress-testing configurations
- 1 week of running configurations against the benchmarks
- 2 weeks of writing the final paper and presentation

Ultimately, the timeline I set out for myself ended up being pretty inline with the work that was performed. There was not that much feedback in regards to my paper proposal other than to look into differentiating my project from work we've performed in class. There were far too many similarities between what I was proposing and what was done in class, and it was basically redoing it with minor changes. In addition to researching about parameters of ISA, I felt like it was necessary to research ways to make my project different from others. Originally, I spent time looking into how to use different benchmarks like SPEC or CoreMark. Unfortunately, there were issues in porting both of them which led me to use a repository of RISC-V benchmarks for computing performance in different areas like integer computing capabilities [1]. I also spent a great deal of time attempting to create my own BOOM configuration based on pre-existing BOOM configurations. The trouble was that altering parameters often introduced dependency issues. For example, making alterations to the number of instructions fetched requires alterations to the number of instructions decoded. This would have been easier if the exceptions thrown were more descriptive of the problems. Despite this, I was able to resolve the issues and end with a finished and working OPT project.

References

[1] ucb-bar/riscv-benchmarks, Nov. 2023. original-date: 2016-03-16T23:08:36Z.

[2] B. A. R. . Welcome to Chipyard's documentation (version "1.10.0")! — Chipyard 1.10.0 documentation, 2019.

[3] C. Celio, P.-F. Chiu, K. Asanovic, B. Nikolic, and D. Patterson. BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS. *IEEE Micro*, 39(2):52–60, Mar. 2019.
[4] C. P. Celio. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec. 2018.
[5] A. K. A. Kumar, S. Al-Salamin, H. Amrouch, and A. Gerstlauer. Machine Learning-Based Microarchitecture- Level Power Modeling of CPUs. *IEEE Transactions on Computers*, 72(4):941–956, Apr. 2023.
[6] R. of University of California. Welcome to RISCV-BOOM's documentation! — RISCV-BOOM documentation, 2019.
[7] R. of University of California. RISC-V BOOM, 2020.
[8] Z. M. Research. Microprocessor Market Size, Share, Growth and Forecast 2023-2030.
[9] A. R. Weiss. Dhrystone Benchmark: History, Analysis, "Scores" and Recommendations, Oct. 2002.