Master's thesis

2023

Henrik Rambech Baumann

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Henrik Rambech Baumann

The Mosaic IQ Microarchitecture

A Set-Associative Approach for Efficient Operand Wake-Up in OoO Cores

June 2023

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

The Mosaic IQ Microarchitecture

A Set-Associative Approach for Efficient Operand Wake-Up in OoO Cores

Henrik Rambech Baumann

Electronic Systems Design

Submission date:	June 2023
Supervisor:	Rakesh Kumar
Co-supervisor:	Per Gunnar Kjeldsberg

Norwegian University of Science and Technology
Department of Electronic Systems

Sammendrag

I de senere årene har fokuset på energieffektive prosessorkjerner ført til forskning på metoder for å maksimere ytelsen til mikrobrikker innenfor gitte energibudsjetter. I out-of-order-prosessorer er fleksibel instruksjonsplanlegging avgjørende for å oppnå optimal ytelse. Tradisjonelt har dette blitt oppnådd ved hjelp av en Issue Queue (IQ) basert på et Content-Addressable Memory (CAM), som gir frihet til å velge klare instruksjoner for utførelse. Imidlertid fører denne tilnærmingen til betydelig energiforbruk når operandene i køen vekkes ved å kringkaste destinasjonoperanden til fullførte instruksjoner for å identifisere ventende instruksjoner. Antallet sammenliginger og ledninger som kreves for denne oppgaven øker superlineært med IQ-dybden, noe som resulterer i betydelig energiforbruk.

Denne avhandlingen presenterer Mosaikk IQ-mikroarkitekturen, en ny tilnærming som tar sikte på å forenkle oppvåkningsprosessen i instruksjonsplanleggingen ved å redusere antall sammenligninger som utføres av CAM-strukturen i IQ-en. Mosaikk IQ-mikroarkitekturen oppnår dette ved å redesigne den fullt assosiative IQ-en til en sett-assosiativ struktur, som muliggjør selektiv kringkasting av operandene indeksert gjennom en tabell.

For å evaluere effektiviteten til Mosaikk IQ-mikroarkitekturen, ble det gjennomført en analyse med en tradisjonell referanse-implementering av IQ-en, med fokus på strømforbruk og ytelse. Spesielt ble det lagt vekt på analysen av strømforbruket knyttet til sammenligning av operander, som er en betydelig kilde til strømforbruk i moderne mikroprosessorer. Resultatene viser at Mosaikk IQ-mikroarkitekturen betydelig reduserer antallet unødvendige sammenligninger, og oppnår en imponerende reduksjon på 70% sammenlignet med referanse-implementeringen. Dette resulterer i betydelige strømbesparelser i oppvåkningsprosessen til IQ-en.

Videre undersøker studien Instruksjoner Per Syklus (IPC), som er en viktig indikator for ytelse. Selv om Mosaikk IQ-mikroarkitekturen oppnår en noe lavere IPC sammenlignet med den beste konfigurasjonen av referense-implementasjonen, begrenses reduksjon til 3%. Derimot bruker Mosaikk IQ-mikroarkitekturen det samme antallet ledninger som den enkleste referansekonfigurasjonen, denne implementasjonen opplever derimot en IPC-reduksjon på 42%.

Abstract

In recent years, the shift in processor design focus towards energy-efficient cores has prompted the exploration of methods to maximize chip performance within given power budgets. In out-of-order processors, achieving optimal performance relies on flexible instruction scheduling. Traditionally, this flexibility is attained through the utilization of Content-Addressable Memory (CAM) based Issue Queues (IQs), which offer freedom in selecting ready instructions for execution. However, this approach comes at a notable energy cost when waking up operands inside the queue by broadcasting the destination register of completing instructions to identify which operands are awaiting the result. The number of comparators and wires required for this task grows super-linearly with the IQ depth, leading to significant energy consumption.

This thesis introduces the Mosaic IQ Microarchitecture, a novel approach aimed at simplifying the wake-up process in instruction scheduling by reducing the number of comparisons performed by the CAM structure within the IQ. The Mosaic IQ achieves this objective by redesigning the fully associative IQ into a set-associative structure, enabling selective broadcasting of operands indexed through a map table.

To assess the effectiveness of the Mosaic IQ Microarchitecture, a comparative analysis is conducted against a state-of-the-art baseline IQ implementation, focusing on power consumption and throughput. Notably, power consumption analysis places particular emphasis on operand tag comparisons, a significant contributor to power usage in instruction scheduling. Results demonstrate that the Mosaic IQ Microarchitecture substantially reduces wasted tag comparisons resulting in an impressive 70% decrease in wasteful comparisons when compared to the baseline. Consequently, this reduction translates into substantial power savings in the wake-up process of the IQ.

Moreover, the study examines the Instructions Per Cycle (IPC) metric, a key indicator of throughput. The Mosaic IQ Microarchitecture maintains a comparable number of average broadcast ports to that of the simplest baseline configuration. However, in contrast to this baseline configuration which experiences a substantial 42% reduction in IPC when compared to an aggressive baseline with four broadcast ports, the Mosaic IQ Microarchitecture exhibits only a marginal 3% decrease in performance.

Acknowledgements

I would like to extend my thanks to my supervisor, Rakesh Kumar, for his invaluable insights and guidance.

Thanks to Roman Kaspar Brunner for his technical assistance in getting started setting up the framework.

Lastly, thanks to my co-supervisor Per Gunnar Kjeldsberg for his availability for answering specific questions regarding formal requirements.

Table of Contents

List of Tables

List of Figures

Abbreviations

NTNU Norwegian University of Science and Technology

CPU Central Processing Unit

ISA Instruction Set Architecture

WAR Write-After-Read

WAW Write-After-Write

RAW Read-After-Write

IQ Issue Queue

RS Reservation Station

IW Issue Width

FIFO First-in, First-Out

ROB Reorder Buffer

CAM Content-Addressable Memory

ALU Arithmetic Logic Unit

FPU Floating Point Unit

CPU Central Processing Unit

ISA Instruction Set Architecture

FXA Front-end Execution Architecture

OXU Out-of-Order Execution Unit

IXU In-Order Execution Unit

ILP Instruction-Level Parallelism

MLP Memory-Level Parallelism

DBMA Dependence-Based Microarchitecture

RISC reduced instruction set architectures

CISC Complex Instruction Set Computer

FXA	Front-end Execution Architecture
AGI	Address-Generating Instructions
IBDA	Iterative Backward Dependency Analysis
IST	Instruction Slice Table
RDT	Register Dependency Table
OoO	Out-of-Order
RAM	Random Access Memory
IPC	Instructions Per Cycle
HPC	High Performance Computing
SSH	Secure Socket Shell
SLURM	Simple Linux Utility for Resource Management

Chapter 1

Introduction

Microprocessors have become ubiquitous in modern computing systems, ranging from personal computers and smartphones to embedded devices and large-scale servers. The demand for high-performance microprocessors continues to grow as applications become increasingly complex and require faster and more efficient processing capabilities. Achieving high performance in microprocessor design is a multidimensional challenge involving factors such as clock frequency, instruction throughput, power consumption, and area utilization.

The need for high-performance computing has driven rapid advancements in microprocessor architectures. These tiny powerhouses serve as the brains of modern computing systems, enabling the execution of complex instructions and processing massive amounts of data. To ensure optimal system performance, the effectiveness and performance of microprocessors are of utmost importance.

Microprocessor architectures have experienced significant modifications to reach outstanding performance. These improvements include, among other things, boosting the number of processing cores, adding sophisticated caching systems, and enhancing branch prediction algorithms. Microprocessor advancements have substantially elevated the complexity of processors and individual cores, leading to a notable rise in power consumption. However, with the continued reduction in transistor sizes and the shifting power density, power considerations have become increasingly critical and require greater attention.

It is important to acknowledge the significant role played by Moore's law and Dennard scaling in enhancing processor performance in previous years. Dennard scaling refers to the scaling law governing the reduction of transistor sizes. As transistors become smaller, both current and voltage scale downward with length, resulting in a constant power density (1). Consequently, more logic can be integrated into a single die without substantial drawbacks. Moore's law postulates that the number of transistors on a chip approximately doubles every two years (2). This has enabled computer architects to drastically scale up their design for high area utilization and increased throughput. With this progression increased performance has been achieved at the cost of increased complexity. in most cases, increased complexity leads to higher power consumption.

Now Dennard scaling is no longer applicable, as it was an approximation that no longer holds true, commonly referred to as the end of Dennard scaling (3). Taking the same approach for designing a microprocessor is now not feasible and the focus has shifted towards improving efficiency rather than solely pursuing performance.

Power constraints are now a crucial factor in modern processor design (4), rendering the pursuit of maximum performance without regard for power consumption obsolete. Several factors contribute to this shift. Firstly, higher temperatures impede processor functionality, and increased power consumption leads to elevated power dissipation within the core, resulting in rising temperatures. As temperature rises, failure mechanisms such as silicon interconnections fatigue, electrical-parameter shift, gate dielectric, junction fatigue, thermal runaway, and electro-migration diffusion worsen significantly (5). The processor may eventually sustain irreparable damage. Therefore, power consumption must in many cases be limited to reduce heating, even if it means decreasing core performance. Consequently, running high-performance processors at maximum performance for prolonged periods is often unfeasible. Instead, performance is restricted to intermittent bursts before being reduced, typically achieved by lowering the clock frequency, a practice known as thermal throttling (6)(7). Substantial resources are allocated to cooling systems in terms of cost and space, but there are limits to effective cooling (8).

Power consumption is a significant concern across various systems, from embedded to server systems. Embedded systems, reliant on limited wireless power sources, benefit from energy-efficient designs, sometimes prioritizing power considerations over performance requirements (9). In server applications, enormous power consumption occurs, and reducing power usage would lower costs and improve server sustainability. By eliminating the need for extensive cooling systems and reducing overall energy consumption, servers can become more environmentally friendly and sustainable (10).

To maximize power savings in future systems constrained by energy limitations, efficiency has become synonymous with performance. This has led to the development of energy-efficient cores that significantly reduce architecture power consumption. In this pursuit, it is crucial to identify the power-hungry components, as this knowledge is key to achieving the desired power savings.

In modern high-performance processor cores, known as Out-of-Order (OoO) execution, the continuous instruction stream is buffered for analysis and reordering to achieve optimal execution order. However, this approach introduces additional complexity and latency, necessitating the use of extra structures and mechanisms. These components are employed to track instruction interdependencies and ensure correctness in execution, ultimately facilitating the efficient operation of the processor cores while maintaining high performance.

To maintain high clock frequencies, the core divides its pipeline into distinct stages with different tasks. These stages include:

1. **Instruction Fetch:** This stage retrieves instructions from memory.

2. **Decode:** Complex instructions are decoded, and relevant information is extracted.

3. **Dispatch:** Resources are allocated, and instructions are dispatched to the appropriate structures.

4. **Issue:** This stage handles instruction scheduling by reordering instructions for optimal execution order.

5. **Execution:** Instructions are executed in this stage.

6. **Writeback:** The results of the executed instructions are written back to memory.

By dividing the core into these pipeline stages, the processor can efficiently handle the various tasks involved in executing instructions while maintaining high clock frequencies.

Instruction scheduling carried out in the issue stage of a processor core is one of the more complex mechanisms, thereby one of the most power-consuming possesses. Inside the processor core, instruction scheduling involves rearranging instructions before executing them to best utilize resources and maximize throughput. Having a pool of instructions to select and reorder is critical to get the most benefits from instructions scheduling. Most high-performance cores, therefore, implement a queue to hold instructions before executing them, called the Issue Queue (IQ). This queue is the most power-hungry structure inside the processor core consuming as much as 18-40% of core power(11)(12)(13)(14). This is mostly due to the Content-Addressable Memory (CAM) structure for the purpose of the wake-up logic. The wake-up notifies instructions inside the queue if their operands are produced by broadcasting the result tag (or ID) of an executed instruction to all entries in the IQ. This process is carried out in parallel, checking all operand tags for all entries in each cycle. These comparisons are quite consuming in terms of power consumption and do not scale well with increased IQ sizes as the CAM-based wake-up would need to check more operands. If all operands of an instruction are produced, it could be selected for execution. Reducing the scope of the CAM parallel lookup, and developing a more scalable and efficient wake-up logic would contribute to creating more efficient and therefore performant future processor cores.

The objective of this thesis is to develop a novel method to simplify the wake-up process by reducing the number of tag comparisons performed by the CAM structure, all while having minimal influence on the performance of the processor core. The design should aim to be customizable, scalable, and agile to adapt to current and especially future architectures.

To that end, this thesis presents The Mosaic IQ Microarchitecture. This design split the fully associative CAM structure into distinct sets, making the IQ set-associative. However, while it still relies on the CAM structure for wake-up, it no longer requires simultaneous operations on the entire queue. Through an instruction steering heuristic instructions are steered into appropriate sets and use a map table to index the sets for the CAM structure to more selectively check operands.

To showcase the effectiveness of The Mosaic IQ Microarchitecture, one promising composition of the design is implemented, explained, discussed, and benchmarked through simulations.

1.1 Contributions

Overall, the contributions of this thesis to the field of microprocessor core microarchitecture can be summarized as follows:

1. **State-of-the-art instructions scheduling:** Examines state-of-the-art instructions scheduling approaches and their implications on the wake-up logic, understanding how these novel approaches try to solve the problem with excessive tag comparisons.

2. **Comprehensive analysis of the wake-up through simulation data:** Provides valuable insight into key metrics inside the issue stage that has significance for the wake-up process.

3. **The Mosaic IQ Microarchitecture:** Presents a new way of designing the IQ structure for increased efficiency.

4. **Implementation and evaluation of a Mosaic IQ:** Presenting a Mosaic IQ implementation, considering key performance metrics and comparing it to a baseline implementation, enabling an understanding of its effectiveness.

1.2 Outline

This thesis is organized into seven chapters. The thesis begins with Chapter 1, the introduction, which provides an overview of the project, the context in which the study takes place, the objective, and the contributions. Chapter 2, the background chapter, lay the foundation required to fully understand the research, implementation, and results. It covers the basic information of important concepts in computer architecture, mostly related to the Issue stage. Chapter 3 focuses on the state of the art, examining existing approaches and techniques developed to address efficiency in instruction scheduling, and how it influenced the wake-up logic. This chapter offers insights into the current state of the field, setting the stage for the introduction of the novel solution proposed in this research. Chapter 4 presents the Mosaic IQ Microarchitecture, which is the central focus of this research. This chapter describes the proposed design, its implementation, and the underlying motivation driving its development. In Chapter 5, the focus shifts to the practical aspects of the research. This chapter provides a description of the experimental setup used for the study and the framework employed to conduct the experiments. It ensures transparency and reproducibility of the research. Chapter 6 presents the results and discussion of the simulations conducted as part of the research project. It showcases the findings and engages in a discussion of their implications. Additionally, this chapter suggests potential avenues for future research. Finally, Chapter 7 serves as the conclusion. It summarizes the research findings, reiterates the significance of the study, and offers recommendations for further research in the field.

Chapter 2

Background

2.1 Computer Architecture and Microprocessors

Computer system design consists of multiple components such as a memory system, interconnects, and the internal processor or microprocessor. The design of such a system is commonly referred to as computer architecture. Computer architecture is a broader term that encompasses fundamental aspects such as Instruction Set Architecture (ISA), hardware, and microarchitecture.

The term *hardware* covers the specifications or specific details of the computer system, such as the packaging technology or the frequency of the microprocessor. The *ISA* of a computer system is the set of instructions that the processor of the computer is designed to process. ISA determines the software that can be executed by the computer and is what bridges the gap between software and hardware.

Microarchitecture defines the design and implementation of the various computer components, such as the processor and memory of the computer system. The organization of the computer components and microarchitecture are used interchangeably.

A microprocessor is the central component of a computer system, functioning as its core control unit and performing essential processing tasks. It serves as the ”brain” of the system, executing software programs or instructions that drive its operations.

As technology advances, microprocessors commonly consist of multiple processors, making the term Central Processing Unit (CPU) gradually diminish. Furthermore, it has given rise to the term *core* to define these processors. As a result, ”multicore microprocessors” has been replaced by the term *multicore*, now widely adopted(15).

2.1.1 x86 Instruction Set Architecture

The x86 architecture is a widely used ISA that has significantly impacted the computer industry. It originated from the Intel 8086 processor (hence the name) introduced in 1978, and its descendants have become the foundation for many personal computers and servers today(16).

The x86 architecture supports a broad range of operating systems and applications, making it a versatile choice for various computing needs. It has been utilized in desktop computers, laptops, servers, and embedded systems. The x86 architecture has also been adapted for use in mobile devices, although reduced instruction set architectures (RISC) such as ARM have gained dominance in that space.

The x86 ISA is defined as a Complex Instruction Set Computer (CISC) ISA. Meaning the x86 ISA are rather complex instructions. x86 supports variable instruction lengths that could involve multiple operations. An example is the enter instruction used to create a stack frame for a procedure. E.g., $\boxed{\boxed{\textit{enter B, N}}}$ where B is the size of the dynamic storage in the stack frame (that is, the number of bytes dynamically allocated on the stack for the procedure), and N gives the lexical nesting level of the procedure. If the lexical nesting level is 0, the instruction would be comprised of the micro-operations $\boxed{\textit{push}}$, $\boxed{\textit{mov}}$, and $\boxed{\textit{sub}}$. The frame pointer is pushed onto the stack, then the current stack is copied from the stack pointer register to the Base pointer register, and then updates the stack pointer with the current stack pointer value minus the value of the first operand (the size operand) (17).

CISC has its advantages and disadvantages. It can simplify the compiler design, and provide high-level abstraction for programmers (as illustrated with the `enter` instruction). It can also reduce code density and improve memory usage. The downside of more complex instructions is that they often also lead to complex microarchitectures. x86 is a prime example of this(18).

Furthermore, x86 also is known for its backward compatibility, allowing newer processors to execute software written for older x86 processors. This compatibility has contributed to the longevity and widespread adoption of the x86 ISA(19). However, as a result, the number of supported instructions has reached an exceptionally large number(20). This further adds to the complexity of the ISA.

Despite competition from alternative architectures, the x86 architecture remains prominent and continues to evolve, with ongoing advancements in performance, power efficiency, and feature sets. Its widespread availability, software compatibility, and continuous innovation contribute to its continued relevance in the modern computing landscape.

2.1.2 Out-of-Order execution

The high performance of modern microprocessors would not be possible if it weren't for OoO execution. Scheduling instructions sequentially result in stalls in the processor core. E.g. when a load instruction is issued to load data from memory, this can be a very tedious process and can take multiple cycles. For in-order execution, all following instructions wait for the load to finish before continuing. This stall limits the performance of the processor.

For OoO cores, the processor would not need to stall, rather it could just execute another instruction instead. This would undoubtedly require that another instruction is, in fact, ready and that the core is able to quickly find and select another instruction without severely increasing the latency. Moreover, in many cases, instructions will need to be executed speculatively not knowing the outcome of branch instructions. If wrong, this has to be detected and fixed in a timely manner. This is a complex task requiring sophisticated mechanisms and designs to ensure precise control and correct program execution.

Nevertheless, OoO executions are undoubtedly one of the key techniques of modern high-performance cores and microprocessors today. It hides the latency of certain instructions, reduces resource stalls, and improves the utilization of the core to such an extent that the throughput is far beyond that of an in-order core.

2.1.3 Instruction scheduling

Instruction scheduling is the process of determining the scheduling of instructions in OoO cores and is a highly complex and power-consuming task. It involves determining the optimal order of executing instructions to maximize performance while considering dependencies and available resources, achieveing high Instruction-Level Parallelism (ILP). The term ILP describes a processor's ability to carry out several instructions simultaneously, exploiting independent operations within a program to enhance performance.

This process requires intricate algorithms and mechanisms to analyze the interdependencies among instructions, balance the workload across functional units, and minimize costly pipeline stalls. The complexity of instruction scheduling arises from the need to efficiently utilize resources, such as registers and execution units while ensuring data dependencies are correctly managed. The power consumption is amplified due to the increased circuitry and control logic required to manage the intricate scheduling decisions, making it a crucial aspect to optimize for power-efficient processor designs.

2.1.4 Dependencies

Dependencies refer to the relationship between instructions that impact the order in which they need to be executed. If dependencies occurring inside the processor core are not respected it would result in hazards, (undesired or incorrect behavior) that would need to be handled as an exception by the core, negatively affecting performance. If the hazards are not handled it would result in erroneous behavior.

1. **Write-After-Write (WAW)** dependency occurs when two instructions write to the same location and the order in which they are executed determines which value that is ultimately saved. It implies the order in which two instructions execute needs to be respected for the correct value to reside in the destination address after the execution is finished.

2. **Write-After-Read (WAR)** dependency arises when an instruction is to read from a location that is subsequently written to by a separate instruction. For the WAR dependency to be respected, the writing instruction needs to wait for the completion of the read instruction.

3. **Read-After-Write (RAW)** dependency is a *data dependency*, where the read instruction is dependent on the data produced by the write instruction.

Data dependencies are the only true dependencies, and the only dependencies it is impossible to avoid (except for some speculation techniques). WAW and WAR are not dependent on the data of the other instruction and are therefore also called *false dependencies* or name dependencies as they only depend on each other due to the use of the same memory address or register. These dependencies are possible to eliminate using techniques such as register renaming.

2.1.5 Register Renaming

Register renaming is a technique used to mitigate false dependencies caused by register usage in a processor. This process only eliminates register dependencies, not memory dependencies. The occurrence of register dependencies is however much more frequent, and therefore also much more critical.

Register renaming is performed at runtime by the processor core. This avoids unnecessary complexity for the programmer or the compiler. Moreover, the processor core will know more about the execution flow of the program compared to the static information obtained by the programmer or compiler. This makes the optimizations done by the core more effective and provides greater flexibility.

For the register rename to work, two separate definitions of registers exist. One is typically referred to as an architectural register which is visible to the programmer, and the other is a physical register. The physical registers are storage locations within the processor.

Register renaming logic maps the architectural register designators into physical designators. This is done for instructions operating on the same architectural register, creating a name dependency. In that case, they will be assigned two separate physical registers, and the named dependency will cease to exist. This is why the core typically requires more physical registers. Renaming will undoubtedly not work for true dependencies, as they are dependent on the actual data(21)(22).

2.1.6 Pipeline Stages

In high-performance microprocessor cores, the execution of instructions typically involves a series of well-defined stages that collectively form the processor's pipeline. These stages, such as instruction fetch, decode, rename/dispatch, issue, execution, and writeback work in tandem to ensure efficient instruction execution and maximize the utilization of available resources. Each stage has important responsibilities, all of which contribute to the overall performance of the microprocessor. This next section will delve into the intricacies of the issue stage, which is highly critical for instruction scheduling. Figure 2.1 (created as part of a project from a previous semester,) draw a simplified and generic block diagram of an OoO pipeline processor core.

2.2 The Issue Stage in Microprocessors

The issue stage is the most complex stage inside a typical microprocessor core. A microprocessor's issue stage is crucial for managing dependencies and scheduling instructions, which significantly impacts how well the processor can use parallelism and execute instructions efficiently, which is essential to the overall execution of instructions. It accepts decoded and renamed instructions and determines their readiness for execution based on

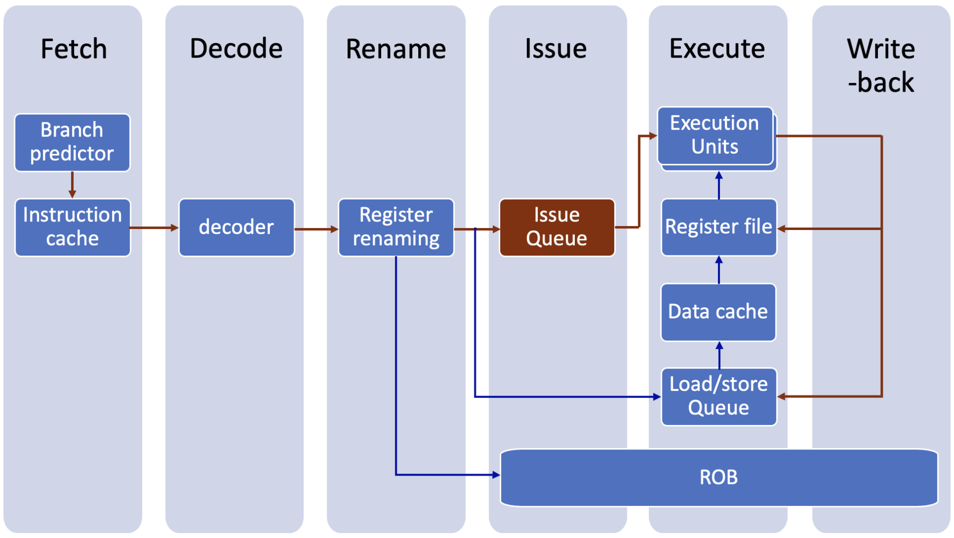

Figure 2.1: Generic pipeline.

various dependencies and resource availability. It acts as a buffer between the frontend of the processor core and the instruction execution.

The issue stage is intended to take advantage of ILP in modern microprocessors' complicated and highly parallel architecture by selecting and allocating independent instructions for concurrent execution. This parallelism enables multiple instructions to run concurrently, boosting throughput and performance overall.

Examining interdependencies between instructions and allocating the resources required for their execution are the primary duties of the issue stage. Dependencies develop when one instruction depends on the outcomes of another, resulting in a data flow dependency. Control dependencies can also happen when the execution of an instruction depends on how a branch instruction turns out. By allowing independent instructions to be scheduled for concurrent execution, this method decreases dependencies and boosts parallelism.

Resource usage is a crucial part of the issue stage. Resources include memory units like load/store units and cache hierarchies and functional units like Arithmetic Logic Unit (ALU)s and Floating Point Unit (FPU)s. The issue stage ensures that instructions that need particular resources are given access to the functional units and memory locations required to run effectively.

Maximizing the available resources and attaining high-performance execution depend on effective management of the issue stage. Because it directly affects the processor's capacity to take advantage of parallelism and carry out instructions in a timely and effective manner, the design and optimization of the issue stage significantly impact the overall performance of a microprocessor.

2.2.1 The Issue Queue

The issue stage frequently uses a queue-based structure known as the IQ which acts as a temporary buffer for instructions waiting to be executed, making it possible to schedule tasks effectively, keeping track of dependencies while hiding stalls and enabling OoO executions of instructions for the most efficient program order. Typically, instructions are kept in the IQ along with pertinent metadata, such as resource requirements and dependency information.

The primary function of the IQ is to provide instructions with dependencies a place to reside until their dependent operands become available. Tracking the operand availability makes the IQ one of the most power-hungry structures within a microprocessor core(23)(24).

The IQ consumes significant power due to its size and the activities it performs. It requires a substantial amount of storage resources to hold multiple instructions and their associated data. Additionally, the operations carried out by the IQ, such as tag comparisons, further contribute to its power consumption.

As most high-performance processors operate OoO, the issue stage directs instructions for execution as soon as their source operands become available, optimizing processor efficiency. The IQ maintains a record of instructions awaiting their operands by implementing an entry in the IQ buffer for each pending instruction. Figure 2.2 (created as part of a project from a previous semester) illustrates a standard entry in the IQ for instructions with a maximum of two source operands.

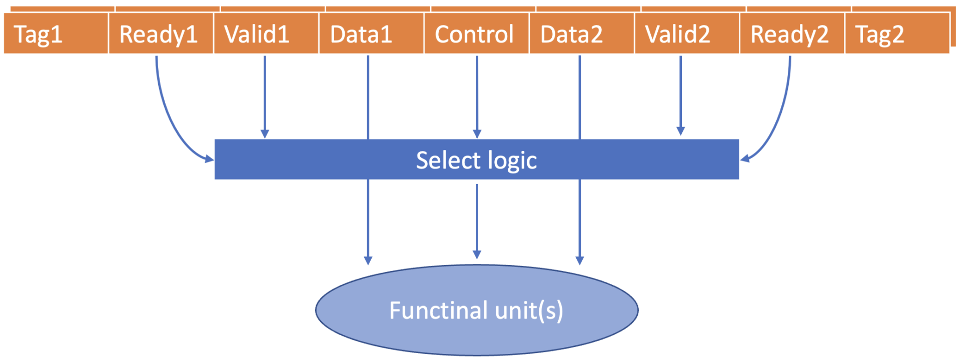

Figure 2.2: IQ entry.

The IQ contains essential decoded information about instructions and their operands. This includes the operand tag, the valid bit, and the ready bit for each operand. The tag uniquely identifies each operand, while the valid and ready bits indicate the status of the operand. The valid bit determines whether the instruction utilizes the operand, as not all instructions require one or two source operands. The ready bit signifies if the operand possesses the necessary data and is prepared for issuance. This data is stored in the data field of the IQ entry for each operand.

Besides storing operand information, the IQ typically retains control signals associated with the corresponding instruction. These control signals aid in managing the instruction's execution by specifying the operation to be performed and the destination for the result. Control signals may encompass details about the operation (e.g., opcode or mnemonic) and other relevant control information.

All of the eligible entries in the IQ are in each cycle considered for issue by the select logic. The select logic's primary function is to determine which instructions from the issue queue will be executed in a particular cycle. To be eligible for selection, an instruction must meet at least two criteria: its source operands must be ready, and the necessary execution resources must be available. For instance, if a processor has only one divider, it cannot simultaneously execute two divide instructions even if it has two divide instructions ready.

2.2.2 Wake-Up Logic

One of the main tasks of the IQ is to wake up instructions when their dependencies are resolved. To achieve this, the unique tag serves as an identifier for tracking dependencies. The tags are typically based on the register renaming ID. When an instruction enters the IQ, it is accompanied by the tags of the operands it depends on. These tags are compared against the destination tags of completing instructions. If the tags match, indicating that the operands are now available, the instruction is woken up and can proceed to the next stage of execution.

Figure 2.3 (created as part of a project from a previous semester inspired by (21)) visualizes the wake-up logic and its functions as follows:

- When an instruction's result is generated, the associated tag (the ID of that instruction's destination operand) is broadcasted to all waiting source operands residing in the IQ.

- The number of unique tags broadcasted could, at most, correspond to the maximum number of instructions issued per cycle, also known as the Issue Width (IW).

- The Wake-up logic broadcasts a tag only when an operand value is produced. Consequently, the number of tags broadcasted varies dynamically, and on average, it is less than the IW as not all instructions produce a value for the destination operand (e.g. branch and store). Furthermore, the core often encounters limitations in consistently issuing the maximum number of instructions each cycle (IW).

- Each instruction within the IQ compares its source operand tags, *Opd tag L* and *Opd tag R*, with the broadcasted tags. If a match is found, the operand is marked as available by setting the ready flag, *rdyL* or *rdyR*.

The IQ is typically implemented as a CAM array. Each entry in the IQ holds one instruction and contains comparators to match the result tags against the operand tags of the instruction in that entry. The outputs of the comparators are combined using OR logic to set the flags accordingly. Once an instruction's operands are all available, it becomes ready for execution and can be considered by the selection logic.

The register file, the IQ, and other elements in the issue stage all function in combination with the wake-up logic. Typically non-speculative values of architectural registers are

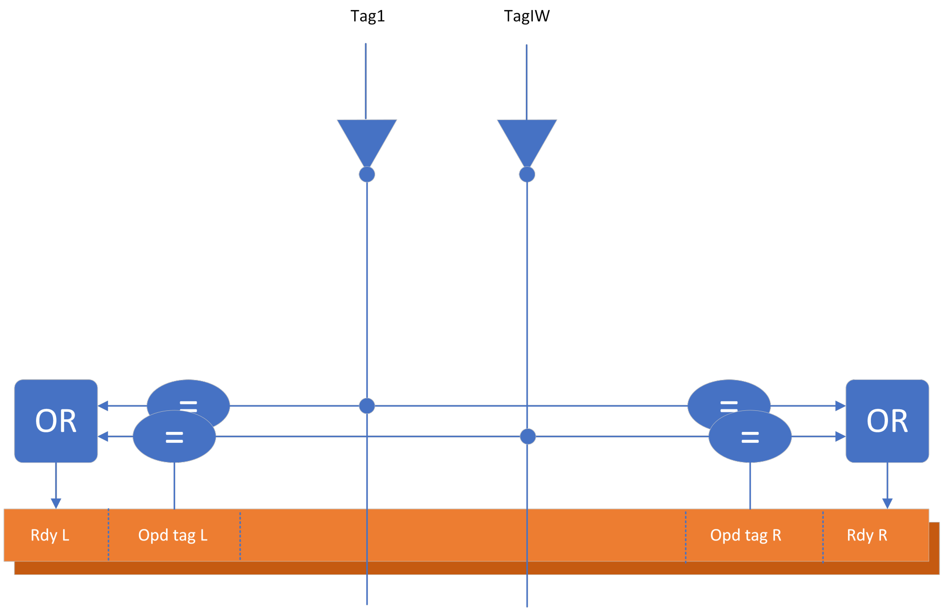

Figure 2.3: Wake-up logic.

kept in the register file, while the instructions and the operands they are connected with are held in the IQ until all dependencies are met. The Issue stage ensures these parts are coordinated and synchronized, enabling effective instruction execution.

Minimizing the processing overhead involved with tag comparisons is one of the main difficulties in wake-up logic. The number of instructions in the IQ increase along with the complexity and depth of the microprocessors instruction window, resulting in a substantial increase in the number of tag comparisons during the wake-up process. These tag comparisons may affect the performance of the entire microprocessor.

Moreover, developing microprocessor designs and technologies is intimately related to optimizing wake-up logic. The necessity for effective wake-up techniques intensifies as microprocessors continue to develop, getting a higher IW, bigger IQs, and greater complexity. Wake-up logic is vital for advancement since high-performance microprocessors sometimes struggle with issues, including lengthy critical pathways, resource competition, and dependencies between instructions.

Chapter 3

State of the Art

The state-of-the-art methods for addressing energy concerns in instruction scheduling are closely connected to the costly CAM-based IQ. This section investigates different approaches and their impact on the IQ. Initially, it explores techniques that eliminate the CAM structures from the IQ. Next, it examines methods to decrease the number of entries required for the CAM-based wake-up mechanism. Finally, it summarizes the main findings derived from the analysis.

3.1 Strategies for Eliminating Wake-Up Logic

The following approaches address the power-expensive wake-up by creating designs that eliminate the need for it entirely.

3.1.1 Complexity-Effective Superscalar Processors

The paper titled ”Complexity-Effective Superscalar Processors” by Palacharla et al.(21) represents an early exploration into the challenges arising from growing complexity in high-performance microprocessor core architectures. The research presented in this paper is motivated by the recognition of increased latency resulting from heightened complexity. The study’s primary objective aligns with this thesis’s goal, which is to mitigate complexity. Specifically, the authors aim to address the complexity associated with the critical issue stage by streamlining the wake-up process. To achieve this, they propose a novel microarchitecture termed ”*Dependence-Based Microarchitecture (DBMA)*,” which simplifies the wake-up logic by employing First-in, First-Out (FIFO) structures to group dependent instructions.

Key insight

The discovery that instructions inside a program frequently display interdependencies, resulting in the construction of dependency chains, serves as the foundation for the DBMA.

The execution of each instruction in a dependency chain depends on its predecessor. The output of one instruction in a dependency chain acts as the input for the subsequent instruction in the chain. Due to the nature of dependency chains, a chain of multiple instructions needs to be executed in-order, such that each instruction can produce the source for its descendant instructions. Having this knowledge, many instructions can be excluded from the wake-up logic, as we know only the head of a dependency chain can awaken. All subsequent instructions in the same chain could then be excluded.

FIFO IQ

Having established that only the head of a dependency chain needs to be considered, it now becomes a question of how to utilize this information best for tracking the chains and effectively mitigate tag comparisons.

By placing the different dependency chains in separate FIFOs, the nature of how FIFOs function naturally accommodates the tracking of a chain, always keeping them in order by using minimal resources. Consequently, only the head of each FIFO must be examined for wake-up purposes. Furthermore, the select logic is simplified as it just needs to consider the FIFOs head for selection.

Instruction Steering

A table similar to the map table used for register renaming is accessed to steer dispatched instructions to FIFOs within the same dependency chain. It is indexed using the local register identifier of the source operand(s) where the entry accessed holds the appropriate FIFO. If the value is already produced, the entry is deemed invalid. If an instruction dispatched is not part of any dependency chain or is the head of a new chain, it is steered into an empty FIFO. If multiple source operands remain to be produced, the instruction is steered into the first FIFO found in the table from indexing with the leftmost source operand.

DBMA Limitations

Although the use of dependencies information to guide the wake-up process is an effective approach, using a collection FIFOs to act as the IQ introduces performance limitations to DBMA. To extract a considerable amount of ILP, an increasing number of instructions must be examined for issue (25). As only the head of the FIFOs could be considered by the select logic, an increasing number of FIFOs would be required, making DBMA redundant. The steering mechanism also relies heavily on the number of FIFOs, limiting it in parallelism detection. When no FIFOs are available, the mechanism experiences stalling until some FIFOs become vacant, restricting the exploitation of ILP to its full potential and prohibiting DBMA from reaching high levels of performance.

3.1.2 The Load Slice Core Microarchitecture

The Load Slice Core microarchitecture, introduced by Carlson et al.(26), offers an innovative solution to address the pressing issue of energy efficiency in contemporary processor cores. This approach entails identifying and tracking Memory-Level Parallelism (MLP) critical instructions Address-Generating Instructions (AGI)s and executing them OoO relative to the remaining instructions, which continue to execute in-order. As a result of the limited implementation of OoO execution, this microarchitecture incurs only a fraction of the energy cost associated with full-fledged OoO execution, providing a more efficient alternative to OoO cores.

Address Generation for Load Prioritization

In-order cores are known for their energy efficiency from not relying on costly power-hungry mechanisms such as the Wake-up logic and select logic. Nevertheless, they often fall short in terms of performance compared to OoO cores. One significant bottleneck in in-order cores is the waiting time for long latency load instructions to complete, leading to stalls that can last for multiple cycles. To maximize MLP, it is crucial to issue load instructions ahead of others, allowing them to execute while waiting for the loads to finish. However, this requires the readiness of the memory address from which to load. This address is produced by preceding instructions in the same dependency chain, known as AGIs. Prioritizing AGIs and loads while maintaining the in-order execution paradigm is achieved by utilizing two separate queues: a bypass queue for executing AGIs and loads and a main queue for other instructions.

To ensure memory dependency correctness, load instructions that precede store instructions must check the store address for potential conflicts. Therefore, the AGIs of store instructions are also included in the bypass queue to facilitate this dependency tracking.

By prioritizing AGIs and loads through separate queues and carefully managing memory dependencies, the microarchitecture of the partially in-order core aims to strike a balance between MLP extraction and energy efficiency. Only the head of the queues needs to be considered for wake-up and issue, therefore, eliminating the need for complex wake-up and select logic while still gaining favorable properties of a fully OoO core.

AGI Detection

The Load Slice Core detects AGIs by leveraging the inherent loop behavior in software using Iterative Backward Dependency Analysis (IBDA). It does this by analyzing loops in a program and gradually finding the backward slice of producer instructions for AGI chain in each loop iteration. This information is stored in two hardware structures: the Instruction Slice Table (IST) and the Register Dependency Table (RDT). The IST keeps track of address-generating instructions, while the RDT maps registers to the instructions that last wrote to them, enabling IBDA to locate one producer slice further back in the chain for each iteration. IBDA works on the in-order instruction stream and does not require recompilation or software modifications, providing benefits without altering the original application code. In case of branch mispredictions or exceptions, the Load Slice Core utilizes a recovery log to rewind and recover register mappings.

Limitations

The Load Slice Core suffers from the same limitations as the DBMA due to the use of FIFOs, severely restricting the select logic's ability to extract high levels of ILP. It also struggles with extracting sufficient MLP in certain memory-heavy workloads. If a load depends on another load, the dependent load blocks the bypass queue which hurts performance.

3.1.3 Freeway: Maximizing MLP for Slice-Out-of-Order Execution

Freeway by Kumar et al.(27) builds on the microarchitecture from the Load Slice Core. It further improves the MLP capabilities of The Load Slice Core microarchitecture by identifying interdependencies between loads. Moving them to an additional FIFO lets the independent loads execute without stalling. This does improve the design, but still, it is not able to reach performance equal to that of a fully OoO core.

3.2 Effective Strategies for Shrinking the IQ

The following design addresses the wake-up complexity by shrinking the IQ having the wake-up logic operated on fewer operands, effectively mitigating some of the overhead from instruction wake-up.

3.2.1 A Front-end Execution Architecture for High Energy Efficiency

The Front-end Execution Architecture (FXA) by Shioya et al.(28) is designed to enhance energy efficiency and performance for single-threaded tasks. It introduces a novel approach by employing two execution units: the Out-of-Order Execution Unit (OXU) and the In-Order Execution Unit (IXU). The FXA leverages both the energy-efficient properties of an in-order core and the performance properties of an OoO core. The front-end IXU tries to execute as many instructions as possible effectively, while the backend OXU ensures that the performance does not drop significantly when the IXU otherwise would stall.

Architecture Overview

The FXA follows a specific execution flow where all instructions are initially fed to the IXU for execution due to its lower power consumption. Instructions that are not ready within the resolution of their dependencies are sent through theIXU as a NOP (No-Operation) and scheduled for execution by the OXU. The IXU effectively filters instructions for the OXU and removes executed instructions from the pipeline. The IXU executes instructions without the need for wake-up and select logic, and in addition, it reduces the stress on the OXU, allowing the OXU to reduce the complexity and size of structures as the IQ.

FXA Limitations

Despite its advantages, the FXA may have limitations that warrant further investigation. These include potential scalability challenges when handling highly parallel workloads or complex instructions. The FXA relies on a bypass network for the IXU to forward data and execute ready instructions in-order. However, the bypass network may become a bottleneck as the number of functional units or pipeline stages increases, making it less adaptable due to increased wire delay and energy consumption.

The FXA requires additional hardware components to support the two execution units. These components may increase the area and complexity of the processor design and introduce additional challenges for verification and testing.

3.2.2 Long term parking: criticality-aware resource allocation in OOO processors

Long Term Parking by Sembrant et al.(29) make the observation that some instructions reside in the IQ for extended periods at a time. These non-critical instructions usually rely on time-consuming operations and tend to spend a significant amount of time in the IQ before becoming ready, thereby increasing the pressure on the IQ. By delaying the inclusion of non-critical slow instructions in the IQ the pressure is reduced, and it becomes possible to reduce IQ dept. The non-critical instructions are delayed by placing them in a cheaper in-order FIFO.

3.2.3 Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors

Alipour et al.(30) first proposed an architecture that leveraged ready instructions to simplify the instruction scheduling. Delay and Bypass by Alipour et al.(31) build on this, using both the observations made by the FXA and the Long Term Parking microarchitecture. FXA exploit instructions readiness by immediate execution in the IXU, conversely the Long Term Parking microarchitecture park non-ready instructions. As these approaches are somewhat orthogonal they can be combined. The Delay and Bypass microarchitecture delays non-ready and non-critical instructions while it let critical ready instructions bypass the IQ for immediate execution.

3.3 Key takeaway

The use of FIFOs, such as in the DBMA and the Load Slice Core, can lead to significant energy savings. However, relying solely on FIFOs tends to result in greater performance loss. On the other hand, reducing the size of the IQ and retaining the CAM structure offers relatively less energy savings but incurs minimal performance degradation. The trade-off between energy efficiency and performance should be carefully considered when designing microarchitectures.

Chapter 4

Mosaic IQ Microarchitecture

The term ”mosaic” captures the concept of various elements harmoniously combining to create a cohesive entity. In the context of the Mosaic IQ Microarchitecture, this concept is applied to the division of the IQ into distinct sets, forming a mosaic-like structure where these sets collaborate to handle instruction scheduling. This chapter provides insights into the rationale behind the development of this innovative design. Subsequently, a comprehensive description of the microarchitecture and its operational principles is presented. Finally, an in-depth exploration of the microarchitecture's configuration and implementation is provided.

4.1 Motivation

The motivation for the proposed design in this thesis stems from three key findings observed in the context of microprocessor architecture. Firstly, by reducing the number of broadcast ports, it has been observed that the impact of excessive tag broadcasts on performance manifests in a significant drop in throughput only when the number of ports is limited to one. Secondly, an analysis of tag comparisons reveals that the number of matches represents a mere 5% of the total comparisons performed, illustrating a substantial gap between actual matches and the overall number of comparisons. Lastly, a notable proportion of instructions exhibit a remarkably limited number of dependencies, with over one-third of instructions having zero dependencies upon dispatch. These findings provide a strong rationale for exploring novel solutions to optimize the wake-up logic in the issue stage of high-performance microprocessors.

4.1.1 Broadcast Port Reduction

Varying the number of broadcast ports available for wake-up each cycle shed light on the impact of excessive broadcasts on performance. Tag broadcasts play a crucial role in coordinating the wake-up of dependent instructions and ensuring the timely availability of operands. However, an excessive number of broadcasts can lead to increased power consumption, decreased bandwidth utilization, and a potential increase in latency.

Figure 4.1 illustrates the change in Instructions Per Cycle (IPC) as the number of ports available for broadcasting the tag changes. The IW is constant for all configurations. Instructions that cannot broadcast due to the limited amount of broadcast ports are placed in an in-order buffer and allowed to broadcast as soon as a port becomes available. There is no limit on the buffer. The bottleneck then only becomes the number of broadcast ports and how this limits the ability to extract ILP and finding ready instructions to execute.

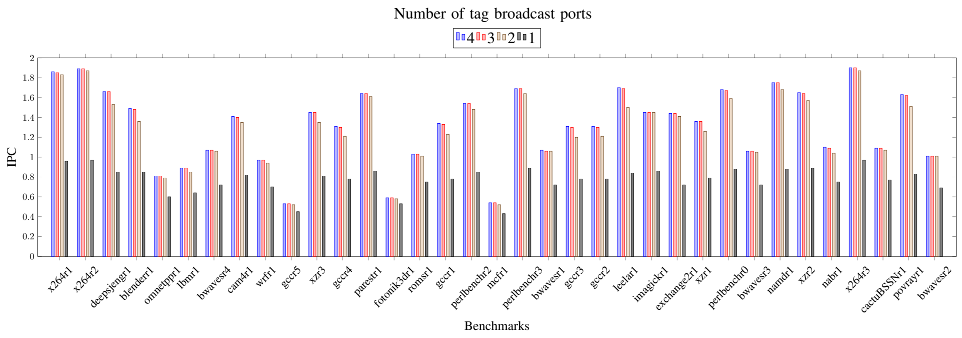

Figure 4.1: Broadcast ports configurations IPC across all workloads.

Figure 4.2 plot the average IPC across all workloads. From 4 to 3 ports there is virtually no IPC drop. Then drop to 2 ports are noticeable but still not significant. The reduction in IPC is suddenly very apparent when reducing the ports to 1, only allowing one instruction to broadcast the tag of its destination operand each cycle. This is likely the threshold for which the benefits from reduced overhead from the reduction in ports arguably no longer could outweigh the reductions in performance, as a performance drop of 42 percent compared to 4 ports are not acceptable when on the contrary, 2 ports only drop less than 5% in IPC.

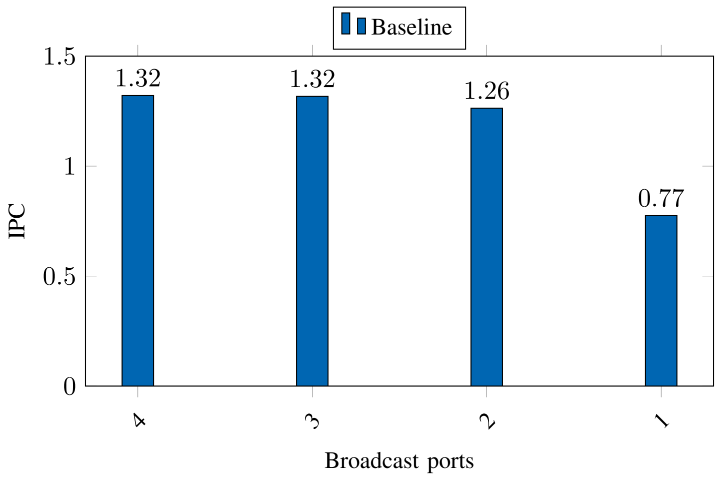

Figure 4.2: Broadcast ports configurations average IPC.

4.1.2 Tag Comparisons

The normalized distribution of instructions broadcasting their result tag by different tag comparisons frequency averaged across all workloads is plotted in figure 4.3. This plot excludes all store and branch operations as they do not produce any result for a destination operand. Hence, they have no result tag to broadcast. The comparisons are counted for every valid and unready source operand for each instruction inside the IQ. If any operand is ready or not valid (as not all operations require the same amount of source register), the operand is considered gated off such that it does not waste a comparison, and it is therefore also not counted in the experimental data used to make the plot.

The plot 4.3 visualize how the number of comparisons varies vastly. With the high number of instructions simulated, the plot exhibits traits of that of a unimodal probability distribution. It rises quickly to a peak of 9 tag comparisons, being the mode at about 5 percent of all instructions. This is followed by an exponential decay until the highest recorded value of 197 tag comparisons.

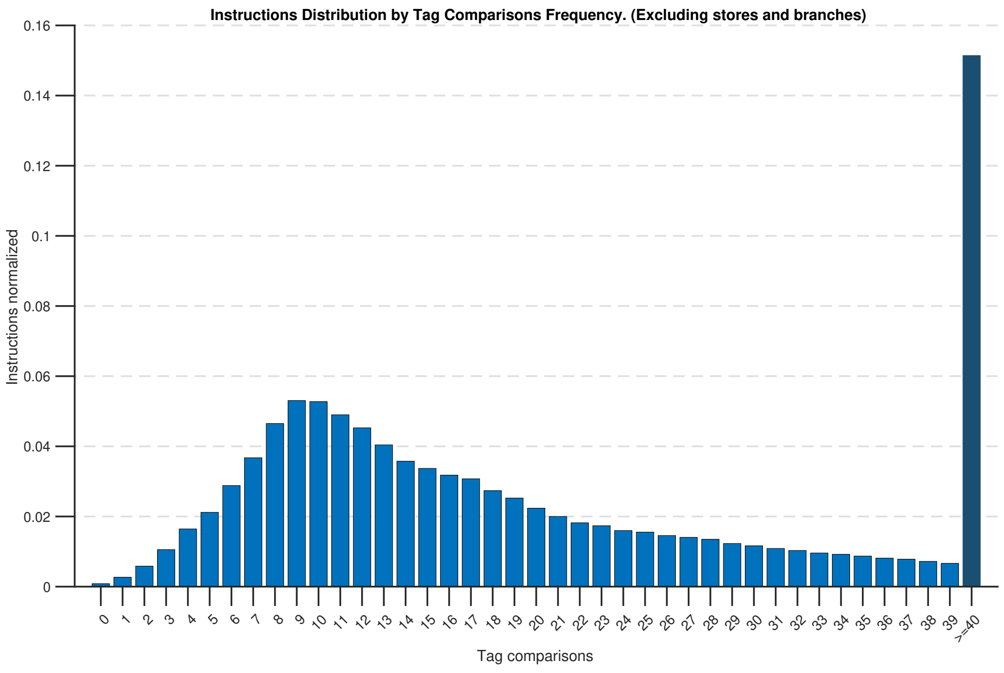

Figure 4.3: Instructions per Tag comparisons

The number of useful tag comparisons resulting in a tag match and an awakened operand is plotted in figure 4.4. The number of operands that are woken up each cycle also varies greatly; as many as 67 tag matches are observed during simulations. The distribution of instructions by tag matches follows similar traits as that of the plot 4.3. However, it peaks at 1 tag comparison per instruction, accounting for more than 70 percent of all instructions, before decaying. The sum of instructions for all different values of the number of tag matches of 4 or above is well below 5%. This indicates that when instructions broadcasting their tag have a high number of comparisons, usually, most of them are mismatched.

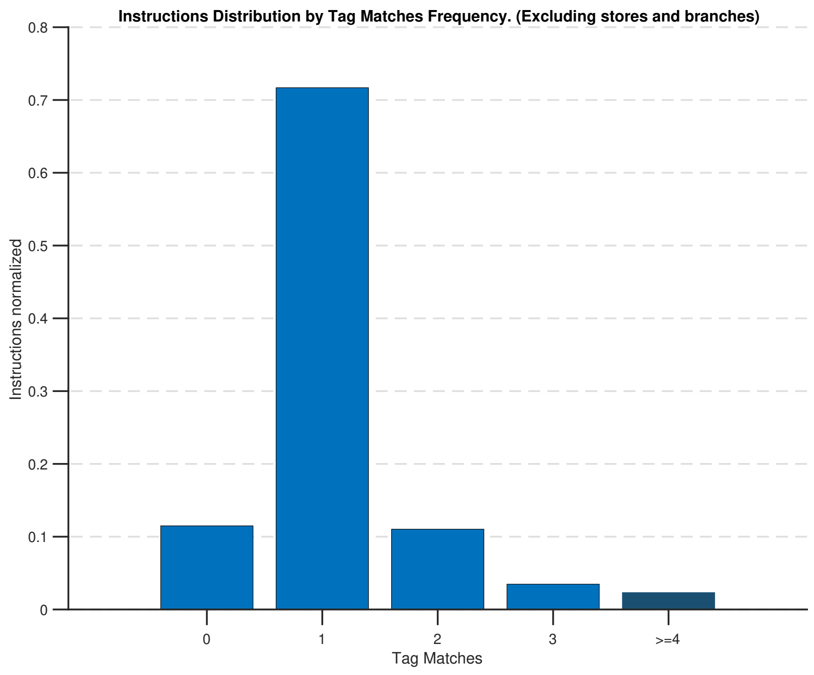

Figure 4.4: Awakened operands

To further establish the potential for tag comparison reduction and get more precise values, the average number of tags comparisons per instruction broadcasting its result tag can be calculated using the formula 4.1.

$$\text{Average} = \frac{\text{Total Tag Comparisons}}{\text{Total Producer Instructions}} \tag{4.1}$$

From this, it can be expanded to formula 4.2, whereas ”*T*” is a set of distinct tag comparison values, each value representing a unique number of tag comparisons. The symbol ”*t*” represents each element or tag comparisons value from the set *T*.

$$\text{Average} = \sum_{t \in T} (\text{Tag Comparisons}_t \times \text{Normalized Instructions}_t) \tag{4.2}$$

Using the data from figure 4.3 and 4.4, the average number of tag comparisons and matches performed per instruction broadcasting its result tag could be derived by adding the values of each bar multiplied by its corresponding number of tag matches for each of the two averages. Lastly, the average for wasted or mismatched tag comparisons could be calculated with the formula 4.3, giving the results listed in table 4.1.

$$\text{Average}_{\text{Mismatches}} = \text{Average}_{\text{Comparisons}} - \text{Average}_{\text{Matches}} \tag{4.3}$$

Table 4.1: Average tag comparisons performed per instruction broadcasting its result tag.

Average:	**Tag Comparisons per Instruction**
Comparisons	23.035
Matches (Useful)	1.161
Mismatches (Wasted)	21.874

Based on the values in 4.1, most tag comparisons are futile and prove that there is considerable room for improvement. Out of all comparisons, 95 percent are mismatches, draining a considerable amount of current. Since high-performance processors today can typically perform in the range of billions to tens of billions of instructions per second, these wasted mismatches are quite substantial.

4.1.3 Dependencies

Each complex x86 instruction is decoded into separate simpler instructions, commonly referred to as micro-operations. The instructions in the Reorder Buffer (ROB) and the IQ are only micro-operations. This reduces some complexity and makes it easier to manage dependencies. After the instructions are decoded, the instruction residing in the IQ has at most 3 source operands. This would infer that micro-instructions could, at most have three dependencies. However, there is also one other origin of source operands. These are the control flag registers. These registers contain important information, such as whether or not a branch should be taken based on the result of a previous instruction executed. These rarely produce dependencies, but nevertheless, they still need to be considered. At most, these control flags comprise 11 possible source operands and, therefore, 11 additional potential dependencies. The total then adds up to a total of 14 possible dependencies for a micro-operation in the IQ.

A dependency analysis of micro-instructions entering the IQ challenges the assumption that a substantial number of instructions possess numerous dependencies. By scrutinizing figure 4.5 it becomes apparent that a significant proportion of instructions dispatched actually exhibit a limited number of dependencies.

Figure 4.6 plots the average across all the workloads, and it becomes visible that nearly half of all micro-operations entering the IQ demonstrate only one dependency, indicating a relatively low interdependence among instructions. Furthermore, more than a third of dispatched micro-operations do not have any dependencies whatsoever. This finding remains consistent across multiple workloads, where more than five dependencies are virtually non-existent. Similarly, the presence of three or more dependencies is relatively rare.

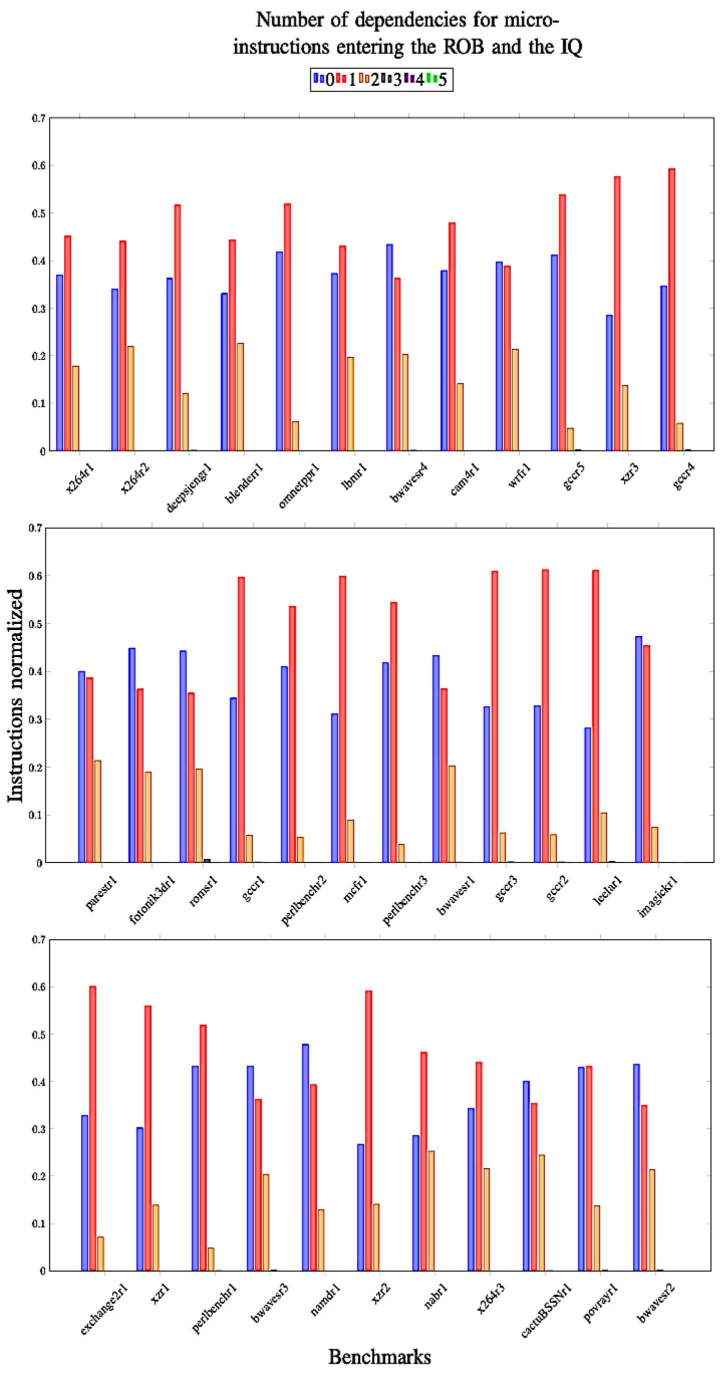

Figure 4.5: Dependencies across all workloads

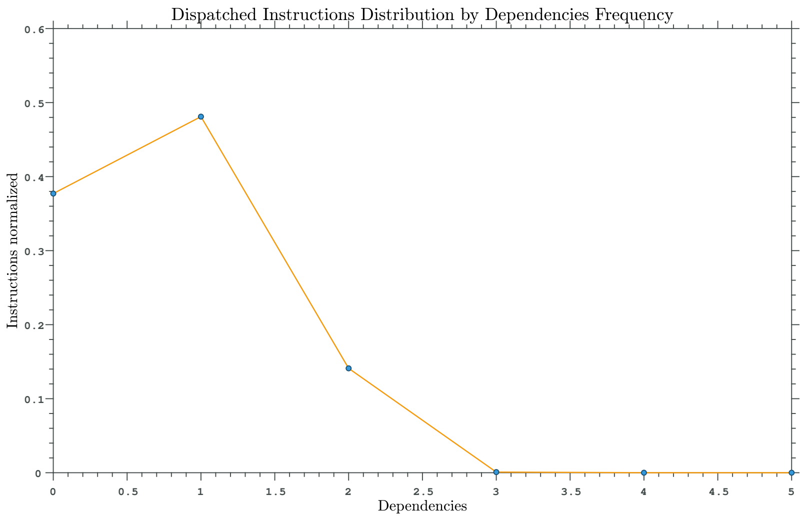

Figure 4.6: Dependencies averaged across all workloads

The identification of a limited number of dependencies presents a promising prospect for mitigating the complexity and overhead associated with wake-up logic. By adopting a strategic resource allocation approach that prioritizes instructions with a higher dependency count, the elimination of redundant tag comparisons becomes feasible for instructions with fewer dependencies.

In particular, the broadcasting process, which entails propagating the results of an instruction to its dependent instructions, can be optimized based on the dependency characteristics. Considering that instructions with a high number of dependencies necessitate more frequent broadcasting, they should be assigned more broadcast ports. Meanwhile, reducing the ports for low-dependencies instructions would offer energy saving while having a minimal impact on performance.

4.2 Design

The proposed design centers around the optimization of wake-up logic within the issue stage of the microprocessor core. This optimization is achieved by dividing the IQ into a set-associative structure. This is mabe possible by implementing an instruction steering heuristic and a map table. The microarchitecture aims to reduce comparisons and the amount of wiring (broadcast ports) while maintaining high performance. This section offers deeper insights into its mechanics and functionality.

4.2.1 Dependency Tracking for Minimizing Wasteful Tag Comparisons

A crucial goal is to minimize the number of tag matches that do not contribute to meaningful operations and, as a result, consume power unnecessarily. Various approaches have been proposed to address this issue. One approach involves tracking all dependents' placement in the IQ and their entire dependency chains. This method has proven to be highly effective in reducing the number of futile tag comparisons. Ideally, if we had complete knowledge of the placement of all dependencies, we could selectively access operands, making the CAM-based wake-up redundant. However, achieving such a level of certainty is not a straightforward task. It would require constant tracking and updating of all these dependencies, which introduces additional complexity and increases power consumption. Otherwise, FIFO implementations for tracking are shown to be possible, but these would never be able to reach substantial performance.

The primary objective is, therefore, to minimize the amount of effort expended and the number of tag comparisons required while ensuring that any potential decrease in performance remains minimal or non-existent. This objective can be effectively achieved by adopting a strategy of broadcasting to a narrower subset of instructions within the IQ. However, it is crucial to ensure that each entry in the IQ contains all the relevant tags that would potentially match the broadcasted tag. Failure to include all matching operands within the broadcasted tag could lead to the occurrence of exceptions, potentially compromising system performance or even introducing the possibility of errors. To mitigate this risk, it becomes imperative to employ a tracking mechanism that prevents such exceptions from arising. Consequently, a trade-off emerges between the intricacy of the tracking mechanism and the benefits derived from the resultant reduction in tag comparisons. The ultimate aim lies in identifying an efficient methodology for tracking dependencies that effectively optimizes the reduction of tag comparisons without compromising system performance.

4.2.2 Set-assosiative IQ

The wake-up process utilizes the operand's tags rather than their addresses by incorporating CAM within the IQ. This approach offers significant advantages by enabling high-speed associative searches without the need for explicit address knowledge. However, as previously established, the CAM structure accesses vast amounts of entries leading to excessive power consumption.

To address this challenge, an alternative approach is proposed in this research. Instead of relying on the conventional fully associative tag lookup, the proposal suggests employing a set-associative tag lookup mechanism with a dedicated CAM for each set. This approach involves dividing the IQ into distinct sets, allowing the provision of addresses for tag comparisons without requiring greater precision than, at most, the number of sets.

Adopting a set-associative design avoids the complexities associated with intricate tracking mechanisms that determine the exact address of each operand while limiting the scope of CAM accesses to specific sets rather than the entire IQ.

4.2.3 Table-Based Broadcasting

After the register renaming process, instruction operands are assigned a physical register that differs from the architectural registers visible to programmers. This ensures that only true data dependencies remain, meaning only one instruction can write to a particular register (as there are no WAW or WAR dependencies). The instructions with their destination operand mapped to the same physical register as the source operands of the upcoming instruction are known as producer instructions or dependencies. Once an instruction is dispatched into the IQ, it is placed in a set, its dependencies need to know which set to broadcast their result tag. Some instructions will have to broadcast to multiple sets, as they are to wake up more operands. Others may not need to broadcast at all.

To effectively keep track of which set to broadcast, this thesis proposes the inclusion of a map table, similar to that used for register renaming. The map table is to be indexed using the local register designator and contains a bit for each set. A value of 1 is assigned to sets intended to be broadcasted, while 0 is assigned to those not. During dispatch, instructions access the map table and enable the bit of the set to which they are dispatched. When instructions leave the IQ and broadcast its tag, the map table is accessed to determine which sets to broadcast. The map table can be implemented as a Random Access Memory (RAM) scheme, where the map table is a register file where the physical registers designator directly accesses an entry that contains the set bits. The number of entries in the map table is equal to the number of physical registers.

4.2.4 Instructions Steering Heuristic and Broadcast Ports

Multiple heuristics can classify and steer instructions into different sets within the IQ. This thesis proposes a simple yet efficient heuristic that employs the count of dependencies. The primary purpose of this categorization is to exploit the fact that a substantial proportion of the instructions that enter the IQ possess a small number of dependencies. Consequently, they have fewer operands to be transmitted to. Any instruction that lacks dependencies will not receive the tag broadcast, whereas those with multiple dependencies will require more broadcasting.

This steering mechanism exploits this observation to adapt the number of ports to different sets. Figure 4.2 illustrated the ports needed to maintain different levels of IPC. When categorizing instructions based on the number of non-ready operands, it is possible to focus the ports on the instructions that require them while reducing ports for instructions with fewer dependencies and removing them entirely from instructions with no dependencies.

4.3 Implementation

This thesis implements a specific design to showcase the potential of a set-associative IQ with dependency-based steering. The IQ is divided into three sets, each accommodating instructions with a different number of dependencies. The proposed design is depicted in figure 4.7. Instructions are steered into the appropriate set based on the number of non-ready operands. The number of ports allocated to each set is based on the number of non-ready operands (dependencies) of the instructions residing in the set. The first set is assigned zero ports and holds instructions with zero dependencies. The second set is allocated one port and holds instructions with one dependency. Lastly, the third set is assigned two ports and holds all instructions with more than one dependency.

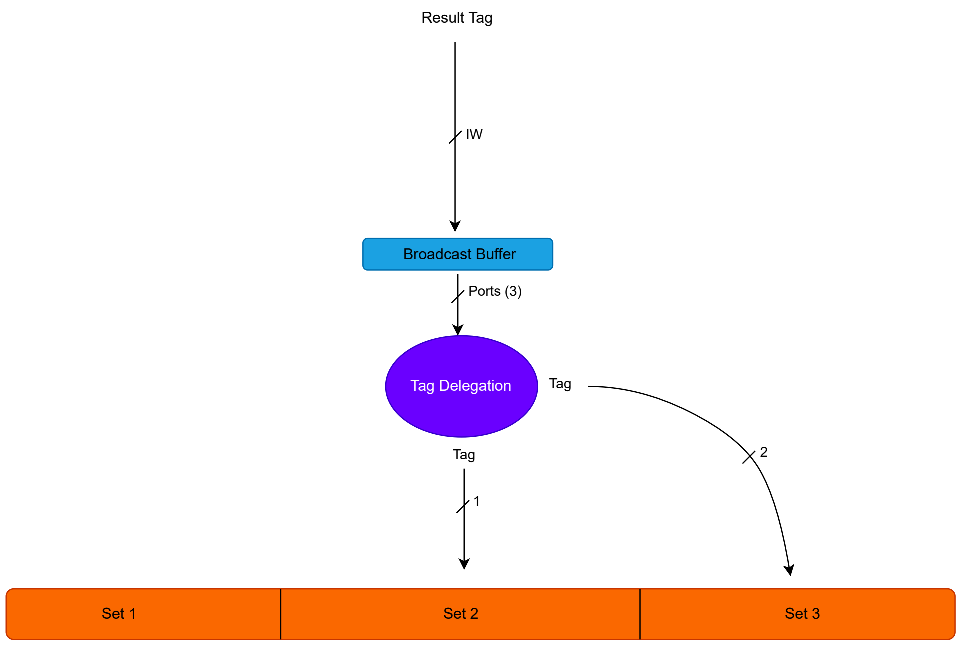

Figure 4.7: Broadcasting tags with 3-set associative IQ.

4.3.1 Broadcast Buffer

The Broadcast Buffer holds tags that are not broadcastable due to limited broadcast ports. If the buffer fills up, the processor would need to stall. The buffer receives at most as many tags as the IW of the processor each cycle. The rate at which the buffer fills up then depends on the processor's IW, while the rate at which the buffer empties depends on the number of ports available for broadcast. The IW of the processor will be higher than the number of ports for tag broadcasting. Otherwise, there would be no need for a broadcast buffer.

The buffer does not need to store many entries. If the buffer fills with many pending tags, instructions in the IQ would not be awakened, quickly influencing the processor's performance. If the broadcast buffer is unreasonably large and never stalls, the processor would naturally come to a halt anyways, as instructions will not be awakened, and the processor would have to wait for operands to become ready and available for issue. One could then easily argue that there is a threshold for which the size increase would not significantly influence the performance. This implementation uses a 32 entries FIFO, storing up to 32 instruction result tags. This is a sufficient size such that the processor never stalls due to the buffer being full, exceeding the suggested threshold.

4.3.2 Tag Delegation

The tag delegation functions as follows.

1. *Check the FIFO for what tag is next.*
2. *Look up the tag-to-set mapped table.*
3. *Check if the required port is available.*
4. *If available, broadcast tag and advance the FIFO.*
5. *Repeat until 3. fails.*

4.3.3 Set 1: Zero Dependency Instructions

The first section of the IQ is allocated zero broadcast ports. This section accommodates instructions with zero dependencies when entering the IQ. These instructions are typically ready for execution and do not require to be awakened. Placing them in a separate section eliminates the need for tag comparisons and broadcasts to this section, reducing some overhead.

4.3.4 Set 2: Single Dependency Instructions

The second section of the IQ is equipped with a single broadcast port for tag broadcasting. Instructions in this section possess only one dependency when entering the IQ. Based on observations, it has been noted that a substantial portion of instructions fall into this category. Placing them in a separate section with a dedicated broadcast port allows for efficient wake-up by broadcasting the tag to the specific dependent instruction. This approach reduces the number of tag comparisons and minimizes unnecessary broadcasts for instructions with only one dependency.

4.3.5 Set 3: Multiple Dependency Instructions

The third and final section of the IQ is designed to accommodate instructions having two or more dependencies when dispatched. This section is assigned two broadcast ports for tag broadcasting. Instructions in this category typically require more extensive wake-up coordination due to their multiple dependencies. By providing two dedicated broadcast ports, the design facilitates efficient wake-up by simultaneously broadcasting the tag of two issued instructions, minimizing delays and ensuring timely availability of operands.

Chapter 5

Framework and Experimental Setup

5.1 SniperSim

To evaluate and develop the Mosaic IQ architecture, the microarchitectural simulator SniperSim was used(32)(33).

SniperSim is a cycle-level simulator designed to accurately model the behavior and performance of modern microarchitectures. It provides a flexible and extensible framework for studying various microarchitectural designs and exploring their impact on system performance.

At its core, SniperSim simulates the execution of instructions, memory accesses, and interactions among various microarchitectural components, such as the core, caches, memory subsystem, and interconnects. It takes into account architectural details, timing constraints, and dependencies to provide an accurate representation of the underlying hardware behavior.

One of the notable features of SniperSim is its ability to simulate OoO execution, allowing for realistic modeling of modern processor architectures. It supports a wide range of configuration options, enabling researchers to explore different architectural parameters and trade-offs(34)(33).

5.1.1 Core configuration

Table 5.1 lists the most critical configuration settings for SniperSim. These settings define various parameters that influence the behavior and performance of the simulated system.

The core is configured with the Intel Nehalem architecture(35). The Nehalem microarchitecture, introduced by Intel, is known for its advanced features and performance optimizations. With SniperSim, the core is set to operate OoO, enhancing instruction scheduling and allowing for better utilization of available resources.

The `commit_width` and `dispatch_width` parameters are both set to 4, enabling the core to handle up to four instructions per cycle. The IW (`issue_width`) of the core is limited by the number of functional units, generic ports, and load/store ports. It

Table 5.1: SniperSim Configuration

Parameter	**Value**
core_model	Nehalem
in_order	false
commit_width	4
dispatch_width	4
issue_width	4
issue_contention	true
rs_entries	180
issue_memops_at_issue	true
address_disambiguation	true
outstanding_loads	48
outstanding_stores	32
store_to_load_forwarding	true

is not limited by the configurations parameter; rather, it ends up not being able to issue more than 4 instructions each cycle at most. This is due to the configuration of the `issue_contention`, respecting the architectural limitations for issue, such as the number of ports.

The size of the IQ is configured to 180 entries set by the `rs_entries`. Reservation Station (RS) is just another word used for the IQ.

Additional parameters such as `issue_memops_at_issue`, `address_disambiguation`, `outstanding_loads`, `outstanding_stores`, and `store_to_load_forwarding` are set to appropriate values to optimize memory accesses realistically.

Note: The provided configuration is a subset of the full SniperSim configuration file and focuses on the most relevant parameters for this research.

5.2 SPEC CPU 2017

Trace files are primarily used for simulation and analysis purposes. Simulators like SniperSim use trace files as inputs to recreate the execution behavior of the benchmark applications. By replaying the recorded events and instructions, simulators can emulate the program's execution and analyze various aspects of system performance.

Tracefiles typically include information such as the instructions executed, memory accesses, branches, or other additional events or statistics.

SPEC CPU 2017 is a benchmark suite developed by the Standard Performance Evaluation Corporation (SPEC) to assess the performance of computer systems, specifically the central processing unit (CPU) and memory subsystem. It consists of a collection of real-world applications compiled with standardized settings to ensure fair and comparable measurements across different systems.

The benchmark suite is designed to simulate a diverse range of computing workloads that represent typical usage scenarios. It includes a variety of applications from different domains such as scientific computing, media processing, financial analysis, and more.

Some of the included benchmarks are popular programs like GCC, Python, Perl, and MATLAB(36).

5.3 IDUN

The SniperSim is a relatively accurate simulator that can run cycle-level simulations. As each workload comprises about 1 billion instructions and there are well over 30 workloads, the computing workloads are very heavy. To efficiently simulate these workloads, IDUN was used to reduce the computation time of the simulations significantly.

IDUN is a High Performance Computing (HPC) cluster of multiple high-performance computer nodes. The IDUN cluster is a collaborative project between various faculties and the IT division at Norwegian University of Science and Technology (NTNU). Its primary goal is to create a professionally managed and highly available computing platform for the university. It facilitates efficient benchmarking and development, making it possible to schedule multiple jobs in parallel with high simulation speeds relative to regular personal computers. It can be easily accessed remotely through an Secure Socket Shell (SSH) connection. IDUN provides a framework for users to submit their computational jobs to the cluster, and it allocates resources such as CPU cores, memory, and GPUs to these jobs based on specified requirements and availability(37). This framework is open-source software called SLURM.

5.3.1 SLURM

SLURM is a highly popular workload manager and job scheduler utilized within HPC clusters. Its primary function is to distribute computational tasks across the nodes of an HPC cluster, ensuring optimal resource utilization and job execution. SLURM is widely used in academic, research, and scientific computing centers, as well as in commercial and industrial settings that require efficient resource management and job scheduling(38).

With its powerful script-based capabilities, SLURM enables users to effectively manage a large number of workloads within an HPC environment. In this study, jobs were submitted per benchmark for simulation purposes. By utilizing multiple SLURM scripts, jobs were easily submitted, monitored, and data was managed for each individual workload simulation.

Chapter 6

Results and Discussion

6.1 Power consumption

6.1.1 Tag Matches

Tag Comparisons are highly correlated with the power consumption in microprocessors. This is especially true for caches and the IQ. The Wake-up process can be quite power-hungry, accounting for up to 63% of the IQ's power consumption in certain architectures(39). This is mostly due to the high number of unnecessary checking of entries from the CAM structure. As the wake-up almost exclusively consumes power due to the tag comparisons, reducing the number of comparisons would inevitably reduce the power consumed by the wake-up activity, ultimately decreasing the total power consumption of the IQ.

Figure 6.1 illustrates the decrease in tag comparisons achieved by the wake-up logic in the Mosaic IQ Microarchitecture compared to the baseline. It plots the average total tag comparisons across all workloads.

The average comparisons per producer instruction are listed in the table 6.1, comparing the baseline to the Mosaic IQ microarchitecture. The matched tag comparisons are assumed to be practically identical. The wasted comparisons, on the other hand, are far from it. An astonishing decrease from around 22 mismatches per instruction down to less than 7, showing that the Mosaic architecture provides around 70% reduction in wasted tag comparisons compared to the baseline. This will certainly significantly reduce the power consumed by both the wake-up process as well as the IQ and issue stage as a whole.

Figure 6.2 plots the tag comparison distribution in the Mosaic IQ microarchitecture. The plot is visibly different from the baseline presented in chapter 4. The mode is now at one comparison with far fewer instructions checking more than 40 operand tags.

6.2 Throughput

To determine the throughput of a microprocessor, two key metrics are considered. This would be the number of instructions that can be executed and committed each cycle; the

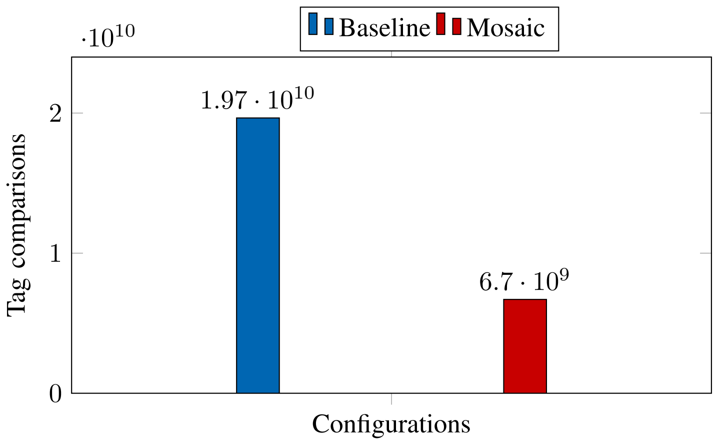

Figure 6.1: Tag comparisons in Baseline vs. Mosaic

Table 6.1: Average tag comparisons performed per instruction broadcasting its result tag, Mosaic vs Baseline.

Average	**Tag Comparisons per Instruction**
	Mosaic
Comparisons	7.851
Mismatches	6.69
	Baseline
Comparisons	23.035
Mismatches	21.874

IPC , and how many cycles the processor performs each second; the processor's clock frequency. By multiplying the processor's clock frequency with the IPC, we derive the number of instructions that a particular core can perform each second.

This work uses the cycle-accurate SniperSim simulator, so the latency of the different pipeline stages is very hard to estimate. This ultimately means that it is difficult to estimate the frequency of the processor core. However, the goal of this thesis is not to improve the latency, and it is estimated to be more or less equivalent to that of the baseline implementation. The basis for this claim comes from the fact that an increase in complexity and wiring generally contributes to increased latency. As this work focuses on decreasing the complexity, the latency is rather likely to decrease with it.

6.2.1 IPC

With the assumption that the clock frequency at which both processors operate is equal, the IPC becomes the determining factor for the throughput of the processor core. Figure 6.3 show the IPC across all workloads, followed by figure 6.4 plotting the average across all

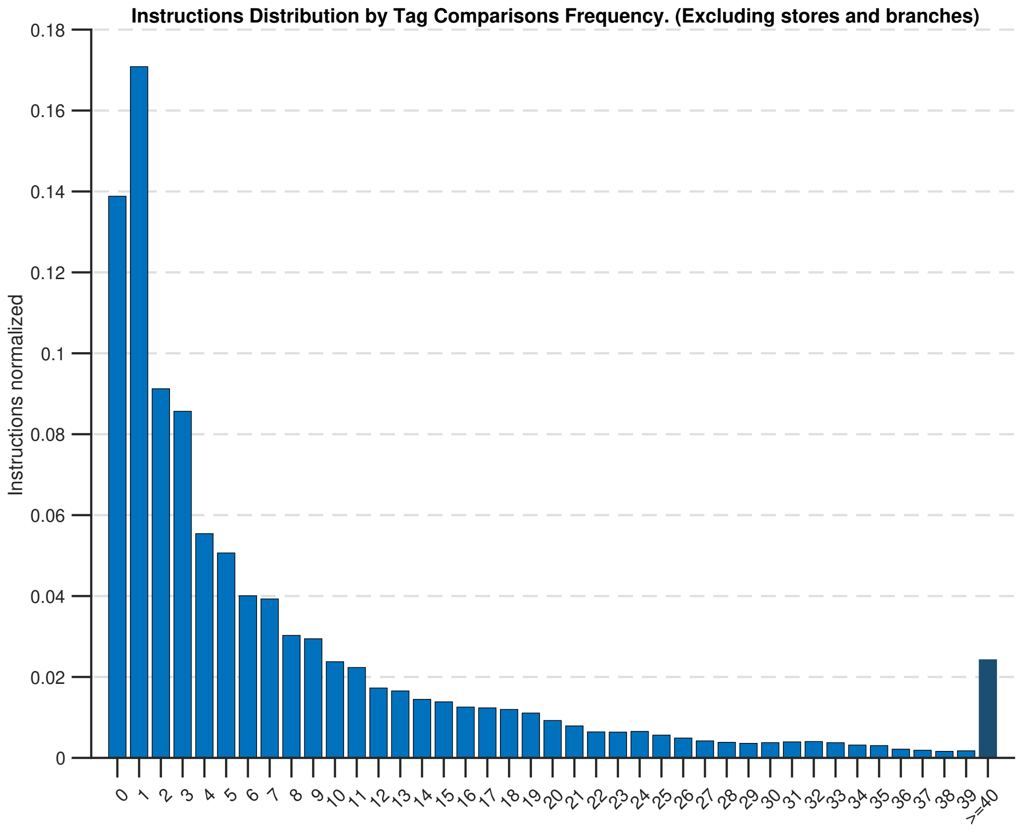

Figure 6.2: Tag comparisons distribution

the workloads. These plots show the IPC of the Mosaic IQ architecture compared to four baseline configurations. These configurations are the number of available ports to the IQ on which the wake-up can operate. The plot shows that the Mosaic IQ Microarchitecture has a slightly lower IPC than the best-performing baseline configuration. However, the throughput drop is not high at around 3 %. On the other hand, the number of broadcast ports and comparators it requires is close to the amount used for the baseline configuration with one port for each entry in the IQ. The baseline configuration experience an IPC drop of above 40%. The Mosaic IQ architecture improves the IPC by 66% compared to the baseline model with a similar amount of wiring and comparators.

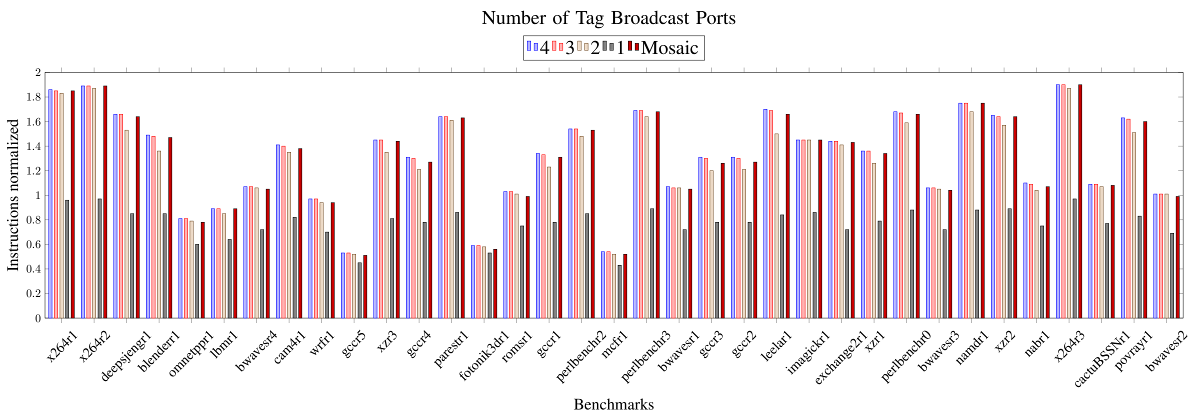

Figure 6.3: IPC across all workloads

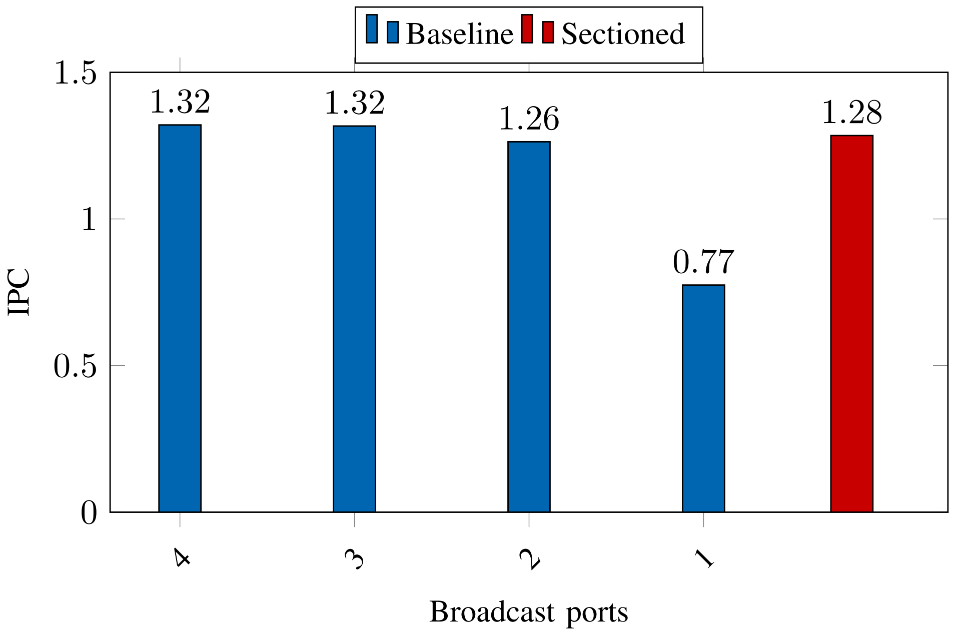

Figure 6.4: Average IPC

The number of ports used for broadcast does not necessarily influence the total number of tag comparisons. More wires do, however, increase complexity and power consumption. More broadcast wires would also implicate more OR gates for tag comparisons, increasing power consumption and an increased area. Furthermore, wires do not scale well, as broadcast wires are long wires that can be critical for the latency of the issue stage. Reducing the amount of wiring could also prove critical for future technologies.

6.3 Future Work

The Mosaic IQ Microarchitecture requires further investigation to explore alternative configurations. Several aspects can be explored, such as:

- **Varying the number of broadcast ports to different sets:** Experimenting with different numbers of broadcast ports allocated to various instruction sets can provide insights into the impact on overall performance. By adjusting the allocation, it may be possible to optimize the ports usage further.
- **Changing the number of sets:** Examining the effects of altering the number of instruction sets within the Mosaic IQ Microarchitecture can help determine the optimal configuration. Increasing the number of sets could potentially reduce tag comparisons and contention, leading to improved performance. However, careful consideration should be given to the associated overhead that may arise from the increased complexity.
- **Varying the size of the broadcast buffer:** Investigating the impact of different sizes for the broadcast buffer can shed light on its relationship with throughput.
- **Trying out different instructions steering heuristics:** Exploring alternative heuristics for assigning instructions to different sets can provide valuable insights. One simple approach could involve implementing a counter to randomly and fairly distribute instructions across sets. This counter could increment from 1 to 6 (one for each set), for example, to determine the set placement of instructions.

To further enhance the evaluation of the proposed design, the following steps can be taken:

- **RTL design implementation:** Creating a Register Transfer Level (RTL) design of the Mosaic IQ Microarchitecture can enable more accurate measurements and analysis. This detailed implementation will allow for a better understanding of the design's performance characteristics, power consumption, latency, and IPC.
- **Investigation of map table and tag delegation overhead:** Examining the overhead introduced by the map table and tag delegation is crucial to understand their impact on overall performance. By quantifying this overhead, it will be possible to assess the trade-offs associated with these components and make any necessary adjustments or optimizations.

Chapter 7

Conclusion

In conclusion, the Mosaic IQ Microarchitecture presents a promising approach to address the power consumption challenges associated with instruction scheduling in OoO processors cores. By introducing a set-associative structure in the IQ and implementing selective broadcasting of operand tag information, the Mosaic IQ successfully reduces the number of wasteful tag comparisons, resulting in substantial power savings during the wake-up process.

The evaluation of the Mosaic IQ against a baseline IQ implementation demonstrates its efficacy in terms of power consumption and throughput. The significant reduction of wasted tag comparisons by 70% showcases the potential for power optimization in microprocessors. Additionally, while the Mosaic IQ achieves a slightly lower IPC compared to the best-performing baseline configuration, it maintains a minor 3% drop. This performance retention, coupled with the preservation of the same number of ports as the simplest baseline implementation highlights the Mosaic IQ's ability to achieve competitive throughput without sacrificing efficiency in the instruction scheduling. The findings of this study emphasize the effectiveness of the Mosaic IQ Microarchitecture in striking a balance between power consumption and performance in instruction scheduling.

To further advance the Mosaic IQ Microarchitecture, future investigation into alternative configurations is crucial. Several aspects can be explored, such as varying the number of broadcast ports allocated to different sets, changing the number of instruction sets, adjusting the size of the broadcast buffer, and experimenting with different instruction steering heuristics. These explorations can provide valuable insights into optimizing performance and resource utilization within the Mosaic IQ.

Ultimately, the Mosaic IQ Microarchitecture contributes to the ongoing efforts in developing energy-efficient processor cores while maintaining satisfactory performance levels. The insights gained from this study pave the way for future research and advancements in microarchitecture design, fostering the development of more power-efficient and high-performance computing systems.

Bibliography

[1] R. Dennard, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, pp. 256–268, 1974.

[2] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, pp. 114–117, 1965.

[3] C.-L. Hu, "The end of dennard scaling," *Communications of the ACM*, pp. 67–77, 2012.

[4] I. Takouna, W. Dawoud, and C. Meinel, "Accurate multicore processor power models for power-aware resource management," 12 2011.

[5] C. H. Small, "Shrinking devices put the squeeze on system packaging," *EDN*, vol. 39, no. 4, pp. 41–54, 1994.

[6] D. R. Sulaiman, "Microprocessors thermal challenges for portable and embedded systems using thermal throttling technique," *Procedia Computer Science*, vol. 3, pp. 1023–1032, 2011, world Conference on Information Technology. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050910005430

[7] T.-H. Tsai and Y.-S. Chen, "Thermal-throttling server: A thermal-aware real-time task scheduling framework for three-dimensional multicore chips," *Journal of Systems and Software*, vol. 112, pp. 11–25, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002319

[8] T. N. Mudge, "Power: A first class design constraint for future architecture and automation," in *Proceedings of the 7th International Conference on High Performance Computing*, ser. HiPC '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 215–224.

[9] I. J. of Computer Applications, "Power Management for Energy Efficiency in Embedded Systems," vol. 145, no. 5, pp. 24–32, 2016.

[10] A. M. Research, "The Global Data Center Power Market Forecast 2019-2025," 2019. [Online]. Available: https://www.alliedmarketresearch.com/data-center-power-market

[11] Y. Kora, K. Yamaguchi, and H. Ando, “Mip-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 37–48.

[12] M. Gowan, L. Biro, and D. Jackson, “Power considerations in the design of the alpha 21264 microprocessor,” in *Proceedings of the 35th Annual Design Automation Conference*, ser. DAC ’98, 1998, pp. 726–731.

[13] K.-S. Hsiao and C.-H. Chen, “An efficient wakeup design for energy reduction in high-performance superscalar processors,” in *Proceedings of the 2Nd Conference on Computing Frontiers*, ser. CF ’05, 2005, pp. 353–360.

[14] S. Manne, A. Klauser, and D. Grunwald, “Pipeline gating: Speculation control for energy reduction,” in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA ’98, 1998, pp. 132–141.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2017.

[16] PhoenixNAP. (2021) x64 vs x86: What’s the difference? [Online]. Available: https://phoenixnap.com/kb/x64-vs-x86

[17] F. Cloutier, “Enter — make stack frame for procedure parameters,” https://www.felixcloutier.com/x86/enter, accessed: 29.05.2023.

[18] Javatpoint. (2021) Risc vs cisc. [Online]. Available: https://www.javatpoint.com/risc-vs-cisc

[19] Microsoft. (2020) x86 architecture. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

[20] W. Mahoney and J. T. McDonald, “Enumerating x86-64-it’s not as easy as counting,” 2021.

[21] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 206–218.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2017.

[23] D. Folegnani and A. Gonzalez, “Energy-effective issue logic,” in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001, pp. 230–239.

[24] Y. Kora, K. Yamaguchi, and H. Ando, “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*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 37–48. [Online]. Available: https://doi.org/10.1145/2540708.2540713

[25] S. Onder and R. Gupta, “Superscalar execution with dynamic data forwarding,” in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, 1998, pp. 130–135.

[26] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, “The load slice core microarchitecture,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 272–284. [Online]. Available: https://doi.org/10.1145/2749469.2750407

[27] R. Kumar, M. Alipour, and D. Black-Schaffer, “Freeway: Maximizing mlp for slice-out-of-order execution,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 558–569.

[28] R. Shioya, M. Goshima, and H. Ando, “A front-end execution architecture for high energy efficiency,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 419–431.

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

[30] M. Alipour, R. Kumar, S. Kaxiras, and D. Black-Schaffer, “Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[31] M. Alipour, S. Kaxiras, D. Black-Schaffer, and R. Kumar, “Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 424–434.

[32] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, “An evaluation of high-level mechanistic core models,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, aug 2014. [Online]. Available: https://doi.org/10.1145/2629677

[33] SniperSim, “Sniper multi-core simulator,” https://github.com/snipersim/snipersim, 2021.

[34] “The sniper multi-core simulator,” http://snipersim.org//w/The_Sniper_Multi-Core_Simulator, 2021.

[35] Intel, “Intel next generation microarchitecture (nehalem),” https://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf, 2008.

[36] “SPEC CPU® 2017,” https://www.spec.org/cpu2017/, accessed: 15-06-2023.

[37] HPC @ NTNU, “Idun - HPC @ NTNU.” [Online]. Available: https://www.hpc.ntnu.no/idun/

[38] “Slurm workload manager,” https://slurm.schedmd.com/overview.html, accessed: 15-06-2023.

[39] D. Folegnani and A. González, “Energy-effective issue logic,” *SIGARCH Comput. Archit. News*, vol. 29, no. 2, p. 230–239, may 2001. [Online]. Available: https://doi.org/10.1145/384285.379266