Fault-tolerant Freeflow using Minimum Additional Overhead

Dual Degree Project Stage 1 Report

Submitted in partial fulfillment of the requirements

for the

Dual Degree Programme

by

Janeel Patel

(Roll No. 180020038)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
August 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic. I would also like to thank Mr. Rajkumar Choudhary for his invaluable time and support.

Janeel Patel
Electrical Engineering
IIT Bombay

Abstract

Transistor count in modern microprocessors continues to rise exponentially thanks to persistent scaling in the CMOS technology. However, such integrated circuits are also more prone to transient faults, making them unreliable as compared to previous chips. Hence, ensuring reliable operation has become very important, especially at a low performance and hardware overhead. This work explores the idea of using a frontend execution engine on top of the Freeflow core [1] which serves two purposes: (1) to early execute ready instructions from the dynamic execution stream and (2) to utilise these frontend ALUs for error detection.

Contents

List of Figures

Chapter 1

Introduction

The goal for September 2022 was to go through the Freeflow core in more depth, particularly focusing on the Early Execution Unit, and to understand implementations proposed in [2] and [3]. The goal was to have a strong fundamental understanding of general fault-tolerant techniques so as to make use of existing ideas for the purpose of detecting errors in execution, if any.

1.1 Background and Motivation

Processors in recent times are designed to either provide significant energy savings (in-order cores) or deliver high performance (superscalar out-of-order cores). At one end of the spectrum, we have in-order (InO) cores that are extremely energy efficient owing to their simple design, while on the other end of the spectrum, we have elaborate superscalar out-of-order (OoO) designs which are extremely power-hungry due to complex performance-boosting hardware structures. An ideal core design should provide high performance at a reasonably low power overhead.

To overcome the slow in-order execution model of InO cores, restriced OoO cores have been proposed to exploit memory hierarchy parallelism (MHP[1]) and to improve instruction level parallelism (ILP) with respect to the standard stall-on-use InO pipeline. A stall-on-use InO pipeline is halted when an instruction that consumes an older load stalls the head of the issue queue, in case of a producer load miss. Now, depending on which cache level this miss is serviced from, the processor stalls for tens to hundreds of cycles. Younger independent instructions are thus disallowed to execute even though they are ready.

[4], [5], [1], all of these papers propose a restricted OoO core microarchitecture with multiple InO queues that dispatch specific kinds of instructions to each queue, and issue instructions from the queue heads. So, instructions are issued in-order from each particular queue, but out-of-order from the heads of various queues as and when instructions at the head of each queue get ready for execution.

The Early Execution Unit (EXU) in Freeflow core [1] executes ready instructions and instructions with short dependency chains in the frontend, reducing pressure on the backend execution engine which has a huge contribution to the overall power consumption and area overhead. This allows us to have a more compact and a more efficient backend.

[1]MHP is defined as the average number of overlapping memory accesses that hit anywhere in the cache hierarchy (including the main memory).

Upon analysing the percentage of instructions executed in the frontend execution engine implemented in [1], we find that the EXU can execute around 32% of the total instructions in the dynamic instruction stream. This also implies that our EXU is idle for most instructions since they do not get executed in the EXU and pass as NOPs through the EXU. So, we can exploit this spatial under-utilisation for redundant execution in order to ensure fault-free operation.

Chapter 2

Literature Survey

In this section, we go through the microarchitecture ideas proposed by [4], [5], [1].

2.1 Load Slice Core (LSC)

LSC builds a second in-order queue on top of the standard stall-on-use in-order pipeline, and separates and forwards load instructions and their address generating instructions (AGIs) to a separate queue called the B-queue. All other instructions (except for the store instruction) are issued from the main queue called the A-queue. Store instructions are broken down into store-data (STD) and store-address (STA) micro-ops. The STA micro-op (along with AGIs for that store instruction) are issued from the B-Q, whereas the STD micro-op is issued from the A-Q. This structure enables LSC to extract more MHP from the dynamic instruction stream as compared to the basic InO pipeline by allowing multiple independent loads to issue from the B-queue, bypassing other load-dependent instructions in the instruction flow.

AGIs leading up to a memory instruction are identified in hardware using a technique called the Iterative Backward Dependency Analysis. These AGIs (commonly known as backward slices for the memory operation) are identified iteratively through multiple runs of the same code using two dedicated hardware structures: the Instruction Slice Table (IST) and the Register Dependency Table (RDT). For each physical register, the RDT stores the instruction address that last wrote to this register. Using this, AGIs are identified in an iterative manner, starting from a memory instruction. Addresses for instructions identified as AGIs are stored in the IST and for each following loop iteration, the hardware looks for producers for known AGIs along with loads and stores.

Experimental analyses conducted show that LSC outperforms the stall-on-use InO core by around 53% on an average (within 25 percentage points of the baseline OoO core) with an approximate area overhead of 15%, and a 22% increase in power consumption with respect to the baseline InO core.

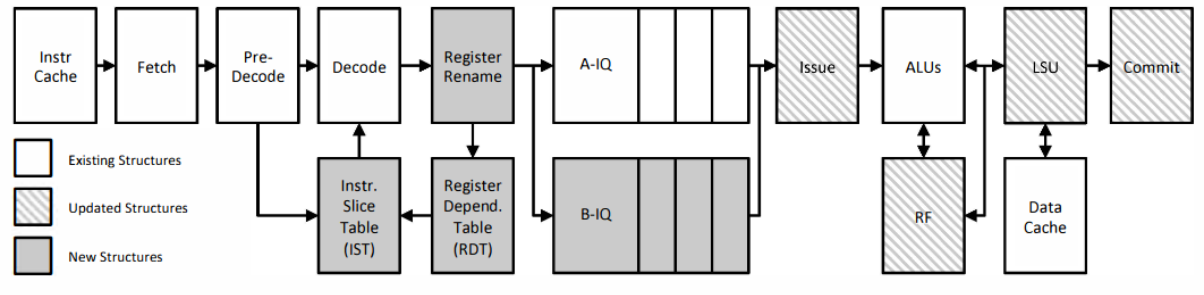

Figure 2.1: Schematic of the LSC microarchitecture. New structures added to the baseline InO pipeline are shaded gray, while those extended over the baseline InO core are dashed.

2.2 Freeflow Core (FC)

One of the major limitations of the LSC microarchitecture is that the head of the queues of either of the two pipelines may be blocked due to a dependency on a long-latency miss, which stalls the entire processor core. In the LSC microarchitecture, producers of a load in the A-queue may halt execution of younger, ready instructions in the queue depending on where the load is serviced from. Blocking of the A-queue by such instructions keeps execution units idle until their dependencies are resolved. FC identifies these instructions and forwards them to a third in-order queue called the C-queue, which is an alternate execution path for high latency instructions.

FC implements simple 3-bit latency counters, the set counter and the reset counter at the head of queues A and C respectively, in order to keep track of the number of cycles an instruction spends at the head of the queues. In case an instruction stalls the A-queue for more than a fixed number of cycles (chosen as six in the proposed implementation and measured using the set counter), its address is stored in the Freeflow Instruction Slice Table (FIST) and it is forwarded to the C-queue in subsequent executions of the same code. Such instructions are termed high latency instructions. There is also the potentiality that the producer for an instruction that previously stalled the head of the A-queue no longer experiences a long-latency cache miss, and the consuming instruction corresponding to this load now becomes a normal instruction. Hence, if an instruction waits at the head of C-queue for less than a fixed number of cycles (chosen as six in the proposed implementation and measured using the reset counter), the high latency instruction is re-marked as a normal instruction, and its address is invalidated in the FIST.

Experimental analyses conducted show that FC incurs a performance hit of 14% while being 100% more power efficient[2] as compared to the OoO design. It performs roughly 89% better than the InO baseline (as indicated the IPC metric) while being 46% more power efficient. It outperforms the then state-of-the-art LSC design by 11 percentage points (in terms of IPC).

[2]Power efficiency is measured in terms of the average number of instructions executed per watt of power spent.

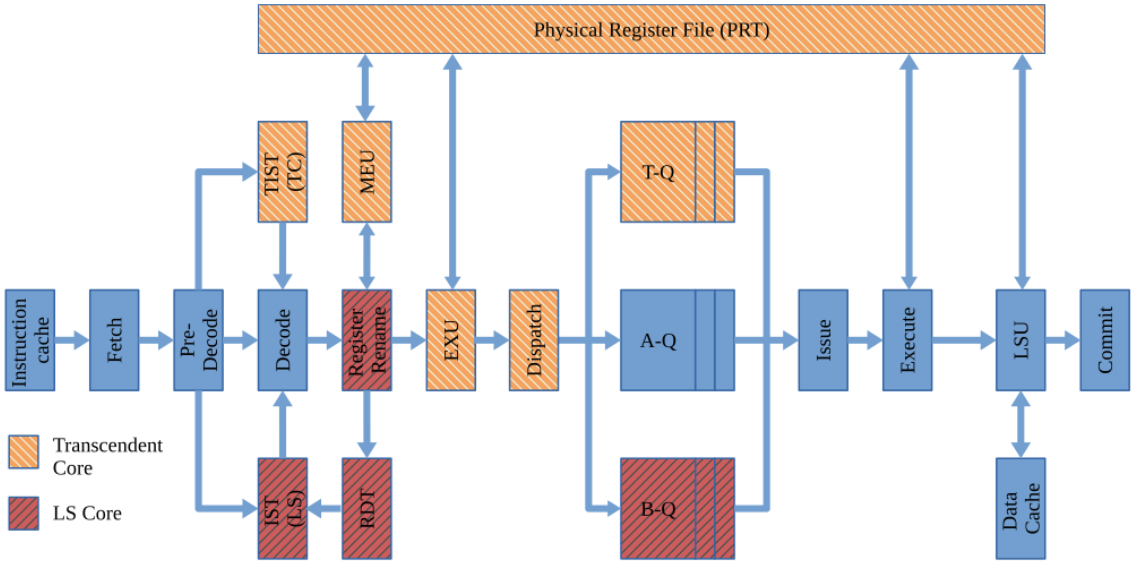

Figure 2.2: Schematic of the FC microarchitecture.

2.3 Forward Slice Core (FSC)

The LSC microarchitecture offers limited MHP because of two reasons: (1) the IBDA mechanism needs few iterations to generate backward slices and due to this, during the initial iterations, AGIs are forwarded to the A-queue negatively impacting the exploitable MHP; (2) if the number of AGIs for a particular memory instruction exceeds the finite size of the IST, some of these AGIs won't be accomodated in the IST causing them to be forwarded to the A-queue incorrectly. The LSC microarchitecture also requires additional hardware structures such as the RDT and the IST.

The FSC microarchitecture builds on the basic stall-on-use in-order pipeline. It identifies forward slices corresponding to a load instruction, defined as the sequence of instructions that depend (directly or indirectly) on a previous load instruction. Such forward slice instructions are sent to dedicated in-order queues: execute-type instructions are dispatched to the Dependent Execute Lane (DEL) whereas load-dependent loads are dispatched to the Dependent Load Lane (DLL). Instructions stuck at the head of the DEL for more than a fixed number of cycles (chosen as four in the proposed implementation) are forwarded to the Holding Lane (HL), to pave way for younger ready instructions in the DEL. FSC identifies forward slices dynamically in hardware using the Steering Bit Vector (SBV). SBV has a bit entry corresponding to each physical register. The SBV bit corresponding to the destination physical register of the load instruction encountered first is set. Any younger instruction that consumes a physical register for which the SBV bit is set, the SBV bit corresponding to that instruction's destination register is set. An SBV bit is cleared upon successful computation of the destination physical register corresponding to the instruction that set this bit. This enables FSC to propagate an instruction's dependence on a load forward. Apart from this, for store instructions, the STA micro-op is replicated across all four lanes in order to deal with the memory disambiguation problem.

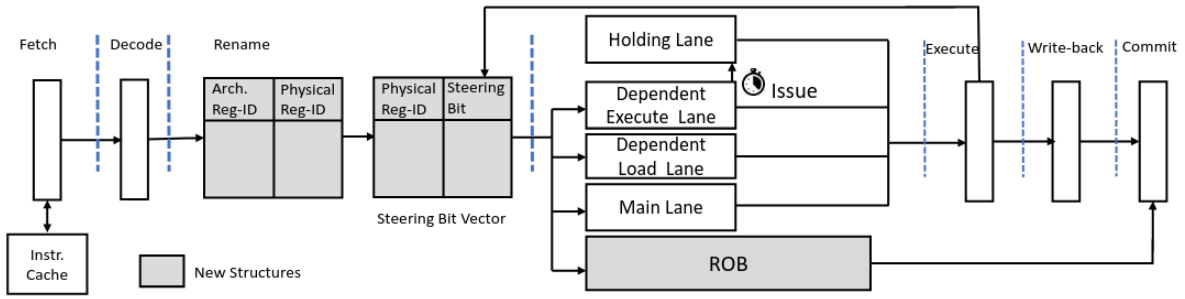

Figure 2.3: Schematic of the FSC microarchitecture. New structures (apart from the three additional in-order queues) added to the baseline stall-on-use InO pipeline are shaded gray.

Experimental analyses conducted show that the FSC microarchitecture outperforms the InO baseline owing to a 64% increase in IPC (a gain of 14 percentage points over the then state-of-the-art Freeway[6] core), performing within 6.9% of the OoO core. The area overhead is just 1% of the baseline InO core, occupying 37% less chip area compared to the baseline OoO core.

Chapter 3

Progress and Future Work

3.1 Completed Work

- Conducted an extensive literature review for restricted OoO microarchitecture proposals.
- Installed Sniper multi-core simulator v6.1 and ran multiple experiments and simulations on implementations for [4], [1] and [5].
- Analysed ideas proposed in [7], [2] and [3] in sufficient detail.

3.2 Plans for October

- Development of a control sub-unit to make use of under-utilised ALUs in the EXU.
- Introducing an FPU in the EXU, and observing impact on performance.
- Performing sensitivity analyses for the different queues in the Freeflow implementation.

References

[1] R. K. Choudhary, N. Singh, H. Nair, R. Rawat, and V. Singh, "Freeflow core: Enhancing performance of in-order cores with energy efficiency," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 702–705, 2019.

[2] S. Gopalakrishnan and V. Singh, "Remo: Redundant execution with minimum area, power, performance overhead fault tolerant architecture," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 109–114, 2016.

[3] S. Gopalakrishnan and V. Singh, "Remora: A hybrid low-cost soft-error reliable fault tolerant architecture," in *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2017.

[4] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," *SIGARCH Comput. Archit. News*, vol. 43, p. 272–284, jun 2015.

[5] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout, "The forward slice core microarchitecture," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, (New York, NY, USA), p. 361–372, Association for Computing Machinery, 2020.

[6] 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)*, pp. 558–569, 2019.

[7] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (USA), p. 419–431, IEEE Computer Society, 2014.