ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. Access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.

HW/SW Mechanisms for Instruction Fusion, Issue and Commit in Modern μ-Processors

PhD Thesis

Abhishek Deb

Barcelona, May 2012

HW/SW Mechanisms for Instruction Fusion, Issue and Commit in Modern μ-Processors

Barcelona, May 2012

Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

HW/SW Mechanisms for Instruction Fusion, Issue and Commit in Modern μ-Processors

PhD Thesis

Abhishek Deb

PhD Advisor:

Prof. Dr. Antonio González	Universitat Politècnica de Catalunya and Intel Barcelona Research Center, Spain
Dr. Josep María Codina	Intel Barcelona Research Center, Spain

Contents

Acknowledgments xiii

1 Introduction 1
- 1.1 A Brief Evolution of Uniprocessors . . . 4
  - 1.1.1 Pipelining . . . 4
  - 1.1.2 Pipeline Hazards and Eliminating the Hazards . . . 5
  - 1.1.3 Modern Processors . . . 5
- 1.2 The case for Co-designed Processor . . . 7
  - 1.2.1 The HW/SW Co-designed Virtual Machine . . . 7
  - 1.2.2 The Co-designed Paradigm as an Enabler . . . 8
- 1.3 Instruction Fusion and Accelerators . . . 8
- 1.4 Low-Complexity Issue Logic . . . 9
- 1.5 Bulk Commit Mechanism . . . 10
- 1.6 The Contributions of the Thesis . . . 10
  - 1.6.1 Co-designed Programmable Functional Unit . . . 10
  - 1.6.2 SoftHV . . . 11
  - 1.6.3 Co-designed Out-of-Order Processor . . . 11
- 1.7 Outline . . . 11

2 The Co-designed Virtual Machine 13
- 2.1 Motivation . . . 13
- 2.2 The Co-designed Virtual Machine Architecture . . . 14
  - 2.2.1 Basic Execution Model Overview . . . 15
  - 2.2.2 The Dynamic Optimizer . . . 15
  - 2.2.3 Frontend Support for Native x86 execution . . . 17
  - 2.2.4 Memory System Architecture . . . 18
  - 2.2.5 Code Cache . . . 19
- 2.3 Microarchitectural Support . . . 19
  - 2.3.1 Reducing Overhead in Co-designed Virtual Machine . . . 19
  - 2.3.2 Microarchitectural support to handle Code Optimizations . . . 21
  - 2.3.3 Miss-speculation Recovery and Bulk Commit mechanism . . . 22
  - 2.3.4 Profiling Support . . . 23
- 2.4 Related Work . . . 23

2.4.1 Virtual Machines . . . 23
2.4.2 IBM DAISY . . . 24
2.4.3 Transmeta Crusoe . . . 25
2.4.4 HP Dynamo . . . 25
2.4.5 Reducing Translation Overhead . . . 26
2.4.6 Optimizer and Interpreter . . . 26
2.4.7 Superblock Heuristics . . . 27
2.4.8 Hardware Dynamic Optimizers . . . 27
2.5 Summary of Design Choices . . . 27

3 Experimental Methodology 29
3.1 The Choice of Microarchitecture Simulator . . . 30
3.2 Timing Simulator Enhancements . . . 31
3.2.1 Modifying the Execution Pipeline . . . 31
3.2.2 Simplifying Memory Disambiguation . . . 32
3.2.3 Write Ports . . . 34
3.2.4 Integrating Wattch into PTLsim . . . 34
3.3 Virtual Machine Monitor Implementation . . . 34
3.3.1 VMM Optimizer Overview . . . 35
3.3.2 Fetching μ-ops from Superblocks . . . 36
3.3.3 Code Generation . . . 36
3.3.4 Performance benefits of Code Optimizations . . . 43
3.3.5 Superblock Details . . . 45
3.4 Simplifications to the Implementation . . . 49
3.5 Baselines Used . . . 51
3.5.1 The Baseline Out-of-Order Processor . . . 52
3.5.2 The Baseline Co-designed In-Order Processor . . . 53
3.5.3 Co-designing the baselines . . . 54

4 Co-designed Programmable Functional Unit 57
4.1 Introduction . . . 57
4.2 The Co-designed PFU proposal . . . 59
4.2.1 Split-Mop Execution Model . . . 59
4.2.2 Alternate Split-Mop Execution Model . . . 61
4.2.3 PFU Microarchitecture . . . 62
4.3 The Co-designed Out-of-Order Processor . . . 67
4.3.1 Bulk Commit of Atomic Superblocks . . . 68
4.3.2 Bulk Commit using a Speculative Map Table . . . 70
4.4 Code Generation . . . 71
4.4.1 Pre-Scheduling . . . 72
4.4.2 Macro-op Fusion . . . 73
4.4.3 Final Code Generation . . . 76
4.5 Performance Evaluation . . . 77

4.5.1 Impact of Microarchitectural Constraints 78
4.5.2 Impact of Fusion Heuristics 81
4.5.3 Impact of Mov-set 82
4.5.4 Comparison with alternate designs 82
4.5.5 Qualitative Discussion on PFU 85
4.6 Related Work 86
4.7 Conclusions 88

5 SoftHV 89
5.1 Introduction 90
5.2 Motivation for Horizontal and Vertical Fusion 91
5.3 SoftHV Architecture Overview 92
5.3.1 Microarchitecture Overview 92
5.3.2 HW Accelerators : ICALU and VLDU 93
5.3.3 Interlock Collapsing ALU 94
5.3.4 Vector Load Units 95
5.3.5 Instruction Encoding 97
5.4 SoftHV Binary Optimizations 98
5.4.1 μ-op Fusion 98
5.5 Performance Evaluation 100
5.5.1 Code Coverage of Fused Instructions 100
5.5.2 Performance benefit due to horizontal and vertical fusion 101
5.5.3 Comparison with an Out-Of-Order processor 102
5.6 Related Work 103
5.7 Conclusions 104

6 A Power-efficient Co-designed Out-of-Order Processor 107
6.1 Introduction 108
6.2 Overview of the Proposed Microarchitecture 110
6.3 Out-of-Order Logic 112
6.3.1 Dependence-based Steering Heuristic 113
6.3.2 Enhanced Steering Heuristic 114
6.3.3 Early Release 118
6.3.4 FIFO start policy 119
6.3.5 Memory Disambiguation Logic 119
6.4 Co-designing the Commit Logic 120
6.4.1 Bulk Commit Problem 3 121
6.4.2 A ROB-free Bulk Commit mechanism 121
6.4.3 Register State 121
6.4.4 Superblock Ordering Buffer 123
6.4.5 Physical Register Recycling 125
6.4.6 Memory State 125
6.4.7 Handling Precise Exceptions 125

6.5 Summary of Changes . . . 126
6.6 Evaluation . . . 127
6.6.1 Performance of new Steering Policy . . . 128
6.6.2 Bulk Commit Mechanism Study . . . 136
6.6.3 Dynamic Power and Energy Results . . . 137
6.6.4 SOB+FIFO Vs ROB+CAM processor . . . 142
6.7 Related Work . . . 152
6.8 Conclusion . . . 154

7 Conclusions 155
7.1 Future Work . . . 156
7.1.1 Larger Regions . . . 156
7.1.2 Coarse-grained Accelerators . . . 157
7.1.3 Alternate Issue Logic . . . 157
7.1.4 Co-designing the Steering Heuristic . . . 158
7.1.5 Speculative Caches . . . 158
7.1.6 More accurate Cd-VM modeling . . . 158
7.1.7 Comparison with other Fine-grain Accelerators . . . 159
7.2 List of Publications . . . 159

Acronyms 161

Index 163

Bibliography 165

List of Figures

2.1 The Co-designed Virtual Machine Architecture Overview 14
2.2 Atomic Superblock 16
2.3 Dual Mode x86 decoder Frontend pipeline 17
2.4 Memory System Architecture 18
2.5 Dual Mode x86 Decoder 20
2.6 The Jump Translated Lookaside Buffer (JTLB) 21
2.7 General Recovery Mechanism 22
2.8 Basic Block and Edge profile of a Control Flow Graph 23

3.1 Block diagram of Microarchitecture that PTLsim models 30
3.2 Illustration of Load hit Speculation. 32
3.3 Store to load forwarding restrictions. 33
3.4 Profiling and Invoking Optimizer Flowchart. 35
3.5 Fetch Flowchart. 37
3.6 Code Generation Flow Chart 38
3.7 Superblock Heuristic 39
3.8 Load Store Telescoping 41
3.9 The Effect of Code Optimizations 43
3.10 Generated Data Flow Graph 48
3.11 Illustration of RRT 53
3.12 The Co-designed In-order Processor Overview 55
3.13 μ-op/macro-op pipeline 55

4.1 Split-MOP Model 60
4.2 Execution Pipeline 61
4.3 Modified Microarchitectural Block Diagram 62
4.4 Programmable Functional Unit 63
4.5 Processing Element 64
4.6 Configuration Line 65
4.7 Hierarchical Issue Queue Model 67
4.8 Bulk Commit and size of Superblocks 68
4.9 Bulk Commit and Frontend Stalls 69
4.10 Bulk Commit with SpecRRT. 71
4.11 Code Generation Flow Chart 72

4.12 Macro-op Fusion Illustration . . . 74
4.13 Dataflow graph showing the fused Macro-op . . . 76
4.14 Schedule before and after the fusion . . . 77
4.15 Impact of grid size . . . 79
4.16 Impact of Write Ports . . . 80
4.17 Impact of PEs connected to input bypass . . . 80
4.18 Impact of PFU Latency . . . 81
4.19 Impact of Fusion Heuristics . . . 81
4.20 Impact of removing mov-set. . . . 83
4.21 Comparison with SIMD and 8-way out-of-order . . . 84
4.22 Comparison 8-way ROB 256 out-of-order . . . 85

5.1 The Co-designed Processor Overview . . . 92
5.2 Code before and after fusion . . . 94
5.3 Interlock Collapsing ALU . . . 95
5.4 Illustration of VLD execution . . . 96
5.5 VLD Miss Handling . . . 97
5.6 MOP/VLD instruction encoding . . . 98
5.7 Code Generation Flow Chart . . . 99
5.8 μ-op Code Coverage . . . 100
5.9 ICALU VLDU Isolation . . . 101
5.10 ICALU VLDU Combination . . . 102
5.11 Comparison with out-of-order . . . 103

6.1 Performance of Steering Heuristics . . . 108
6.2 Block diagram of Microarchitecture of SOB based processor . . . 111
6.3 FIFO based OoO Logic . . . 113
6.4 Dispatch Stalling Conditions . . . 115
6.5 Dispatch Stall Condition Distribution . . . 115
6.6 Steering Logic . . . 116
6.7 Illustration of head and tail check . . . 117
6.8 Illustration of early release . . . 118
6.9 Register State Rename and Commit. . . . 122
6.10 Superblock Ordering Buffer . . . 124
6.11 Performance of Enhanced Steering Heuristic . . . 128
6.12 Dispatch Stalls . . . 129
6.13 Dispatch Stall Condition . . . 129
6.14 Early Release . . . 131
6.15 Enhanced Steering Heuristic . . . 131
6.16 Early Release and Steering Heuristics . . . 132
6.17 The Effect of Per FIFO Issue Width . . . 133
6.18 The Effect of Issue Queue Start Policy . . . 134
6.19 Comparison with Lat_FIFO scheme . . . 134

6.20 SOB vs ROB 136
6.21 The Effect of SRRT 137
6.22 Normalized Dynamic Power FIFOs 138
6.23 Normalized Dynamic Power and Performance Results. 139
6.24 Normalized Energy FIFOs 140
6.25 Dynamic Power Consumption SOB + SRRT vs ROB + SpecRRT 141
6.26 Performance SOB vs ROB without memory disambiguation 142
6.27 Performance SOB vs ROB with memory disambiguation 143
6.28 Dynamic Power Consumption SOB vs ROB 144
6.29 Dynamic Power Consumption Distribution SOB vs ROB 145
6.30 Energy consumption SOB vs ROB 146
6.31 Energy-Delay SOB vs ROB 147
6.32 Energy-Delay^2 SOB vs ROB 147
6.33 Performance SOB vs ROB + CAM with memory disambiguation 148
6.34 Energy SOB vs ROB + CAM with memory disambiguation 149
6.35 Dynamic Power Consumption Distribution ROB with superblocks 150
6.36 Energy-delay product SOB vs ROB + CAM with memory disambiguation 150
6.37 Energy delay^2 product SOB vs ROB + CAM with memory disambiguation 151

List of Tables

2.1 Summary of Design Choices. 27

3.1 Superblock Size in μ-ops 44
3.2 Overhead in terms of x86 instructions 46
3.3 Baseline 4-wide Out-of-Order Processor Configuration 51
3.4 Baseline 8-wide Out-of-Order Processor Configuration 52
3.5 Baseline In-order processor configuration 54

4.1 Proposed Processor Configuration 78

5.1 Breakdown of Speedup 102

6.1 Qualitative Comparison 126

List of Algorithms

1 List Scheduling Algorithm . . . 42
2 Pre-Scheduling Algorithm . . . 73
3 Macro-op Fusion Algorithm . . . 75
4 Fusion Algorithm . . . 100

Acknowledgments

First of all I would like to thank my Advisors Prof. Antonio González and Dr. Josep Maria Codina for their guidance and vision. I would like to thank Josep Maria for being generous enough to accommodate time for meetings even at a short notice. We had numerous technical discussions, which lead to the key contributions of this thesis.

I would like to thank Generalitat de Catalunya and the Spanish Ministry of Education and Science. This thesis was partially supported by the Generalitat de Catalunya under grant 2009SGR1250, the Spanish Ministry of Education and Science under contracts TIN2007-61763 and TIN2010-18368, and Intel Corporation. I was supported by a FI grant of Generalitat de Catalunya.

I would like to thank all the administrative staff at UPC for being very kind and helpful. They provided the much needed logistic support for me to complete my PhD. They had also shown extra-ordinary professionalism, yet warmth, whenever a task was required to be done.

I would like to thank many PhD students at UPC, with whom I have had several technical discussions. I learned a lot just by interacting with other senior and junior students. I would like to thank Paul Carpenter for his support throughout my PhD. I would like to thank him for spending time reviewing this thesis and my papers emphasizing on minute details.

I would like to thank all my family and friends for their support. Specially to my parents, my sister and Arijit, they had provided me the much needed motivation. My parents have always valued education, and they are the one who had sowed the desire in me to be a researcher.

Finally, I would like to thank Manjusree for being very patient and supportive. She had motivated me at times when I was down and always inspired me to do better. She had always lent her ears, listening to me, my issues and let me make correct decisions.

Chapter 1

Introduction

Modern microprocessor designs have shown significant performance improvements over the last decades by making advances in semiconductor process and by exploiting instruction level parallelism (ILP). Out-of-order processors were introduced in order to execute independent instructions in parallel and in an order determined by the dataflow order instead of the program order.

Their unique ability to execute instructions in this dataflow order, enables them to exploit the dynamic behavior of the application, which for a compiler is hard to determine. More so for applications that are irregular, such as SPECINT[3]. The irregularities arises due to unbiased branches. Moreover, memory disambiguation plays a big role in letting the loads execute early; thereby reducing the critical path.

However, the benefits in performance due to out-of-order execution came along with the price of complex hardware structures, which lead to higher power consumption [46] and design complexity [76]. Moreover, these hardware structures tend to become the critical path of the design, leading to diminishing returns.

Furthermore, as the chips are getting smaller with each generation, the power density is increasing sharply. As a result the power consumption is becoming a key deciding factor on the performance that can be achieved [75].

As a result of which, the semiconductor industry has embraced multi-cores, i.e. using multiple simpler cores in a single chip. This helps in achieving a good power-performance trade-off. However, providing high performance now depends a lot on the software; how well it can utilize multiple cores. This is a challenging problem and advances are being made in this area [67, 65, 90, 36, 50, 44, 13].

However, there will always be a need for high-performance but simple uniprocessor cores that satisfies the power budget and performance need of system designers. In fact when higher performance is desired, it may be more energy-efficient to execute applications on

an out-of-order processor than on an in-order processor [8].

An out-of-order processor can be seen as a hardware dynamic optimizer. Dynamic scheduling enables in re-ordering instructions on-the-fly based on their data dependencies. However, instructions have false dependencies between them, which makes hard for the out-of-order engine to fully utilize the functional units. False dependencies exists between two instructions writing to the same destination register, or between an older instruction that consumes from a register and a younger instruction that writes to the same register.

As a result, if these instructions complete out-of-order then it would lead to an unintended program behavior. Register renaming enables in getting rid of these false data dependencies. The benefit of performing all the optimizations on-the-fly enables in achieving a better performance for irregular applications. However, as mentioned earlier addition of these hardware structures lead to higher complexity and higher power consumption.

A software approach to perform out-of-order execution can be implemented with the use of static compilers. Since compiler optimizations such as scheduling re-orders the instructions, a new program order is generated that is different from the programmer's intended program order. However, the static compiler optimizations have their own limitations. For instance, code optimizations are applied at the granularity of Basic Blocks, which limits the scope of the optimization. Moreover, they have to be conservative, for instance, speculative load hoisting cannot be performed.

The dynamic optimizer, on the other hand, can enjoy the benefits of speculative optimizations by exploiting the runtime behavior of the application. Execution is rolled-back to unoptimized code versions in case of a misspeculation. Hardware structures can be added to enable some of these speculative optimizations.

Moreover, since most of the optimizations can be done by the dynamic optimizer, the underlying microarchitecture can be simpler. The use of simpler microarchitecture helps in cutting down power consumption and complexity. Moreover, simpler hardware structures enable higher clock frequency, resulting in reduced execution time.

As a result, the Instruction Set Architecture (ISA) to which the Operating System (OS) and the application is compiled to *–the source ISA–* can be implemented partly in the software and partly in the hardware. Such an approach to co-designing the *source ISA* is referred to as the HW/SW Co-designed Virtual Machine (Cd-VM) in the literature [88].

Most prior work on Cd-VM considered a VLIW microarchitecture. VLIW offers simple hardware with low power consumption. Moreover, it can also provide better performance with the right use of code generation techniques by the dynamic binary optimizer.

However, the use of a VLIW microarchitecture requires, in most of the cases, the need to emulate the cold code in software[1]. As a result of which, the overhead incurred can be

[1] Cold code has to be interpreted. For instance, its hard to decode source X86 instructions to VLIW instructions using a hardware decoder

significant; more so for codes that last for short period of time.

Hence in order to overcome the overhead due to cold code emulation, in this thesis, we consider a Cd-VM that emulates only the hot code. Hardware support is used to execute cold code natively by decoding them into μ-ops. As a consequence of this support, we propose co-designed processors using in-order and out-of-order microarchitecture.

The key objective of this thesis is to revisit the HW/SW co-design paradigm and to propose novel HW/SW techniques to improve performance and/or energy-efficiency. These can be achieved by using new accelerators, by combining existing accelerators, and by considering improved design for the key stages of the processor pipeline as issue and commit logic. With such goals in mind we have proposed techniques in the following three areas:

- **Instruction Fusion :** We propose two mechanisms for instruction fusion using specialized functional units. The fused instructions when executed on these specialized functional units accelerates parts of applications. These functional units exploit two properties : 1) executing collapsed instructions with low latency, 2) exploit ILP by bundling the execution of parallel instructions together. The associated code generation heuristics are proposed as well.
- **FIFO based Out-of-Order Issue :** The existing co-designed processors are primarily VLIW based. Using VLIW cuts down the processor complexity and power consumption. High-performance can be obtained by using advanced code generation techniques. However, even with dynamic optimizer the lack of out-of-order execution can be a performance handicap –for the cases where its hard to know the instructions readiness times. A FIFO based out-of-order issue provides a middle-ground solution with limited out-of-order execution at low-complexity and low-power consumption.
- **Bulk commit :** We propose two bulk commit mechanisms for atomic superblocks in the context of co-designed out-of-order microarchitecture. These bulk commit enables larger superblocks to execute without being constrained by out-of-order resources such as ROB. Moreover, these bulk commit mechanisms also reduce resource related stalls at frontend.

In this chapter first we will briefly discuss the evolution of uniprocessors in Section 1.1. Next, in Section 1.2 we will motivate the need for co-designed processors. We will briefly describe a HW/SW co-designed Virtual Machine and how it acts as an enabler. Next, in Section 1.3 we will discuss instruction fusion and accelerators in the context of co-designed Virtual Machines.

In Section 1.4 we will briefly mention low-complexity issue logic with limited out-of-order capability. In Section 1.5 we will describe bulk commit mechanism required in order to commit the program state corresponding to a region atomically. We will also show how

the existing solution is limited to in-order microarchitectures and motivate the need for newer solutions for out-of-order microarchitectures. Next, in Section 1.6 we will highlight the contributions of this thesis. In the end, in Section 1.7 we will present the outline of the rest of the thesis.

1.1 A Brief Evolution of Uniprocessors

Although, general purpose the uniprocessors are, their design is guided by the applications that would execute on it. Microprocessors for embedded systems, desktops and servers differ from each other in numerous obvious and subtle ways. They are classical examples of being application-specific yet being general purpose. This section discusses how the uniprocessors have evolved into their modern day avatars.

There is on going research both in industry and academia to strive for better performance and energy-efficiency for uniprocessors. Decades of research have resulted in breaking various kinds of dependencies to achieve higher performance. The research was driven in order to increase the performance by concurrently executing instructions.

1.1.1 Pipelining

Pipelining is at the very inception of concurrent instruction execution. An instruction spends several cycles inside the processor moving from one pipeline stage to another. As a result, multiple instructions could be present inside the processor at any given point of time. This might result in longer latency, but a higher throughput[2] is achieved. As one can imagine pipelining is a sort of concurrency that is quite prevalent in our day-to-day lives and is analogous to assembly lines in the Manufacturing Sector.

However, the concept of pipelining in the context of processors differs from that in the manufacturing world. This is simply because instructions are different, such as arithmetic instructions, memory instructions, control transfer instructions etc. Moreover, the execution latency of instructions are different as well, and in some cases non-deterministic[3]. As a result, the continuous flow of instructions in the processor pipeline is interrupted. Furthermore, instructions are dependent, unlike objects in assembly lines. This causes the instructions to stall until its producer has executed.

[2] as measured in instructions per processor cycle

[3] a load that misses in a cache

1.1.2 Pipeline Hazards and Eliminating the Hazards

These interruptions are widely known as bubbles in the pipeline, and are caused by various kinds of hazards. First kind of these hazards are the data hazards that are caused due to dependencies among instructions. True dependencies need to respected in order to allow correct flow of data from the producing instruction to the consuming one. False dependencies, on the other hand, are caused due to limited architectural registers.

Second kind of the hazards are the control hazards that are caused by a control-transfer instruction such as a branch, jump etc. These instruction affect the seamless flow of instructions in the pipeline, because the instruction that needs to be executed immediately after the branch need not be the one that is placed immediately after it in the compiled code.

Third are the structural hazards, which are caused due to several instructions needing to use the same resource[4]. This causes a resource conflict and leads to bubbles in the pipeline.

Researchers have come up with numerous solutions to break these hazards. Structural hazards are the easiest one to break, simply by replicating the resources. Control hazards were tackled by predicting the outcome of a branch at fetch itself. This enabled fetching of instructions from the predicted path instead of stalling. Several branch prediction schemes [97, 87, 70] were invented to predict the outcome of the branch.

Among data hazards false dependencies were broken with the help of register renaming [91]. Researchers have even proposed breaking true data dependencies using Value prediction [62, 84].

1.1.3 Modern Processors

In-Order Processors

Over the period of time some of these ideas have made their place into commercial microarchitectures. General purpose microarchitectures can be broadly classified into in-order and out-of-order. In-order processors executes instructions in their original program order. These processors are pipelined and could also issue multiple independent instructions in a single cycle, which is widely known as superscalar issue.

The modern in-order processors, however , are not simple pipelined processors that they used to be anymore[5]. Instead they have decoupled microarchitectures, such as ARM Cortex-A8[94] and Intel Atom. Buffers are introduced between fetch, decode and execu-

[4] Examples of shared resource are Register ports, Memory ports, Functional Units etc.

[5] Though in really low-end domain simple pipelined in-order processors are used.

tion backend in order to decouple each of these parts of the processor. This decoupling avoids in letting the bubbles in the later stages to ripple up to the earlier stages of the pipeline.

Out-of-Order Processors

By issuing the instructions in an order that is determined by the readiness of instructions instead of its age; the sequential program order is broken. In other words out-of-order program execution is achieved. Instructions are issued as soon as their operands are ready and all the resources related to execution are available. A Reorder Buffer (ROB) is present in the backend to ensure the program state i.e. the register state and the memory state is updated in the original program order.

The first out-of-order processor was proposed by Tomasulo for IBM 360/91 floating-point unit [91]. However, the modern day out-of-order processors differs significantly from the original Tomasulo's processor. For instance, in modern out-of-order processor exceptions are supported precisely using ROB.

Moreover, in Tomasulo's algorithm, the *Reservation stations* provide operands, where as in modern out-of-order processors operands are provided either by the ROB or the physical register File. Furthermore, branches are predicted to better utilize the functional units. This implies instructions from predicted path execute speculatively; the use of ROB ensure update of program state in the program order.

Following are some of the widely know commercial out-of-order microarchitectures DEC Alpha21264[58], Intel Netburst[51], AMD K8[56], IBM PowerPC based[85], ARM Cortex-A15[2].

The benefit due to out-of-order execution is highly dependent on the hazard removal mechanisms described above. For instance, in absence of register renaming and branch prediction mechanism there would be fewer instructions available to be woken up. In such a scenario, its performance would be equivalent to that of an in-order processor.

Moreover, in out-of-order microarchitectures various hardware structures such as issue queues, reorder buffers, load store queues are required for successful out-of-order execution. Both issue queues and load store queues are CAM based structures. The multiple tag comparison required every cycle are the sources of complexity [76] and power consumption [46]. As a result, scaling such structures adds further to complexity and power consumption.

Recently, the shift in microprocessor industry has been towards simpler low power and lower-complexity processors such as the introduction of Intel Atom. However, even for a low-end domain the need for high-performance is always growing and new mechanisms must be found in order to improve performance at low-power and low-complexity.

VLIW Processors

As described above, in-order processors and out-of-order processors are the two extremes of uniprocessor design space. In-order processors have low design complexity and can be clocked faster. Out-of-order processors, on the other hand, provides significant performance benefit in applications with high ILP. Moreover, they could also be energy-efficient when high performance is desired.

Ideally, one would like the power consumption and complexity of in-order processors and the performance of out-of-order processors. From one vantage point out-of-order processors seems to be performing on the fly code optimizations in hardware. For instance, registers are renamed similar to Static Single Assignment (SSA) to eliminate false dependencies. Instructions are issued when their operands are ready similar to code scheduling techniques applied by a compiler.

A trade-off could be reached by performing the code optimizations using the compiler accompanied with a simpler microarchitecture, in other words a HW/SW co-designed approach. VLIW [43] was thus invented; VLIW processors have the ability of issue multiple independent instructions. Compiler packs a group of independent instruction into a Very Long Instruction Word. As as result the hardware is simplified by getting rid of expensive issue and dispatch logic such as scoreboarding, issue queues etc.

Advanced scheduling techniques such as software pipelining [80, 63] could be applied to better form the VLIW instructions. They work very well for kernel oriented applications but not so for irregular application such as SPECINT [3].

1.2 The case for Co-designed Processor

As described above VLIW is a HW/SW co-designed approach toward better performance at lower power consumption and processor complexity. However, the use of static compiler limits the ability of code optimizations that could be applied. As mentioned earlier, the static code optimizations have to be conservative.

1.2.1 The HW/SW Co-designed Virtual Machine

A simpler microarchitecture when accompanied with a dynamic optimizer overcomes some of the above-mentioned shortcomings. Aggressive code optimizations could be applied by speculating on dependencies. Detection and correction mechanism can be implemented in the hardware to deal with misspeculation. As a result performance benefit is obtained by making the common case fast.

This approach when applied on a full-system level lead to the invention of the Cd-VMs [88]. In such a scheme, a processor is a co-designed effort between hardware and software designers. In other words the ISA is partly implemented in the software and partly in the hardware. The software layer performs dynamic binary translation and optimization on the source code, in order to adapt it to better exploit the capabilities of the underlying microarchitecture.

As mentioned earlier the co-designed paradigm enables in migrating to a simpler microarchitecture, because code optimizations are done in the software. Aggressive code optimizations enable in achieving high performance. The optimizations are performed on a code region once[6] and stored for later reuse. The optimization and translation costs are amortized by repetitive use of the optimized code region. A detailed description on Cd-VM architecture and related work is presented in Chapter 2.

1.2.2 The Co-designed Paradigm as an Enabler

Primarily, a Cd-VM enables microarchitects to migrate to a simpler core, thereby cutting down power consumption and complexity. It also enables microarchitects to add features that are not visible to the application or the OS, such as accelerators in order to speedup parts of applications. The accompanying software layer selects and fuses groups of instructions to be executed in the accelerator. In this thesis we will show the use of **accelerators** in a co-designed environment.

Moreover, simpler core need not be a very simple microarchitecture, as is the case with the existing designs. Microarchitects can choose from a wide spectrum of processor design space, ranging from low-end in-order processors to aggressive out-of-order processors. Past designs such as Transmeta Crusoe [60] and IBM Daisy [39] have just focused on VLIW based HW/SW Co-designed processors. In this thesis we will show that **FIFO based out-of-order logic** when combined with HW/SW co-designed processor leads to an energy-efficient design.

Furthermore, it requires the **commit mechanism** to be co-designed in the context of both the in-order and out-of-order processors. This requirement actually turns into a boon and further cuts power consumption by getting rid of some structures. Moreover, this co-designed commit also plays a key role in enabling aggressive code optimizations.

1.3 Instruction Fusion and Accelerators

Due to recent advancement in process technology, transistors in a die are fairly abundant. In this scenario, we argue that specialized hardware accelerators are a promising

[6] several levels of optimizations could be applied incrementally as code gets hotter

alternative to harness both the abundance of transistors and the potential of a wider machine. These performance improvements are achieved under a reasonable power budget and design complexity.

Single-instruction multiple-data (SIMD) accelerators are commonly used in current microprocessors to accelerate the execution of multimedia applications. These accelerators perform the same computation on multiple data items using a single instruction. Intel SSE[40] and AMD 3DNow![74] extensions are examples of such instructions for the x86 ISA. Although, SIMD accelerators provide significant performance gains in multimedia applications at low cost and energy overheads, they provide limited gains for general purpose applications.

More recently, several multiple-instruction multiple data (MIMD) accelerators have been proposed that range from programmable to specialized functions [96, 98, 26, 52]. Due to design complexity and lack of compiler and code generation techniques, in order to leverage the accelerators efficiently, these accelerators have not yet been implemented.

Introducing such hardware accelerators needs to be supported by extending the ISA. Applications need to be recompiled to the new ISA in order to use these hardware accelerators. A HW/SW Cd-VM makes a very good fit in this scenario, where a hardware accelerator can be introduced transparently by generating the code using the dynamic compiler.

1.4 Low-Complexity Issue Logic

Transmeta Crusoe [60] and IBM DAISY [39] cuts down the processor complexity by using VLIW microarchitecture. However, as mentioned above even with advanced code analysis there are cases where dynamic ILP cannot be exploited fully, especially in case of irregular applications.

In order to exploit dynamic ILP of an application, modern processors issue instructions in an order which is different from their sequential program order. Along with out-of-order execution exists superscalar issue, where multiple independent instructions are issued together to their respective Functional Units. This enables in exposing the ILP of an application which is hard to determine by the compiler.

However, the complexity [76] and power consumption [46] of out-of-order issue logic is high. Modern microprocessors use CAM based issue logic to issue instructions. The wake-up signal is driven to all the entries of the issue queue. Multiple tag comparisons are performed every cycle to wake-up the dependent instructions.

FIFO based out-of-order logic [76], on the other hand, cuts down the complexity and power consumption drastically yet provides limited out-of-order capability. A *steering*

logic steers instructions to different FIFOs based upon its dependencies. Multiple independent instructions could be issued simultaneously from different FIFOs. The net result is a design point that lies between an in-order processor and a CAM-based out-of-order processor. The performance of a FIFO based out-of-order processor depends upon the *steering heuristic*.

1.5 Bulk Commit Mechanism

The dynamic optimizer selects the scope for optimization by selecting a region to which the optimizations will be applied. In this thesis we consider a dynamic optimizer that generates a trace of instructions such as a superblock. In order to support various speculative code optimizations superblocks are atomic in nature. The atomic property of the superblock implies that the change in the program state due to a superblock be updated only when all the instructions corresponding to the superblock have successfully executed. As a result, the program state must be updated at once, when its known to be safe to do so. Such a commit mechanism in order to commit the program state of multiple instructions together is known as a *Bulk Commit Mechanism*.

In the context of in-order processors the *Bulk Commit Mechanism* has been proposed by Transmeta [60]. In their proposal two copies of the Register File is maintained the *Shadow Copy* and the *Working Copy*. The *Shadow Copy* holds the committed state and the *Working Copy* holds the speculative state. When the last instruction of the superblock commits the contents of the *Working Copy* is copied to the *Shadow Copy*. On misspeculation the execution is rolled back by simply copying the contents of the *Shadow Copy* to the *Working Copy*.

The speculative memory state, on the other hand, is held in gated store buffers [95]. Its contents are discarded on a rollback and committed to memory hierarchy at bulk.

However, in the context of out-of-order processors the above-mentioned register *Bulk Commit Mechanism* is inadequate. This is because just having a single working copy and a shadow copy will significantly affect concurrent execution of superblocks. To allow concurrent execution of Superblocks the state of multiple superblocks needs to be held.

1.6 The Contributions of the Thesis

1.6.1 Co-designed Programmable Functional Unit

In Chapter 4 we propose a novel co-designed programmable functional unit along with a novel execution model. We also propose code generation heuristic specific to instruction

fusion.

The Cd-VM monitor selects instructions for fusion and generates them. The fused instruction is known as a *macro-op*. By collapsing and executing a chain of simple ALU instructions with low latency, performance is improved. Since independent instructions are fused as well, the pressure on processor resources are reduced. For instance, a single ROB entry is required for a macro-op.

Moreover, since we use a co-designed out-of-order processor, we propose a novel bulk commit mechanism. We provide a solution towards *Bulk Commit Mechanism* using an additional Register Rename Table (RRT), the SpecRRT. SpecRRT holds the physical Register Mappings of the speculatively committed instructions. The contents of SpecRRT are copied to the Backend RRT when the tail[7] of the superblock commits.

1.6.2 SoftHV

In Chapter 5 we propose a co-designed in-order processor using two application specific accelerators. In this work, fusion of dependent and independent instructions are considered separately. Dependent instruction fusion is called as *vertical fusion*, whereas independent instruction fusion is called as *horizontal fusion*. The two techniques accelerates the application by combining the most commonly found pairs of instructions in applications.

1.6.3 Co-designed Out-of-Order Processor

In Chapter 6 we propose a co-designed out-of-order processor. We propose new steering heuristics and show that FIFO based out-of-order processor is a middle-ground solution between an in-order processor and a CAM based out-of-order processor. We propose new steering heuristic that narrows the performance gap with CAM based out-of-order processor.

Moreover, we provide another *Bulk Commit Mechanism* in the context of out-of-order processors. In this particular solution we get rid of the ROB entirely and instead maintain the order at the granularity of the superblock using Superblock Ordering Buffer (SOB). The register state of each superblock is held in per Superblock Register Rename Table, the SRRT. SRRT and SOB allows concurrent execution of multiple superblocks.

1.7 Outline

This section presents the outline of the rest of this thesis.

[7] the last instruction in program order of the superblock

Chapter 2 "The Co-designed Virtual Machine" provides the Cd-VM background needed to understand this thesis. This includes an overview of Cd-VM architecture and the corresponding support required in the underlying microarchitecture. In the end we discuss related work on HW/SW Cd-VMs.

Chapter 3 "Experimental Methodology" discusses the simulation environment used in this thesis. This includes the discussion on microarchitectural timing simulator. We also present details of the Cd-VM environment that we have implemented in this thesis. We also discuss the dynamic compilation step, focusing on Superblock formation and code optimizations. We provide some superblock related statistics such as its size, the optimization overhead. We also discuss the limitations of our implementation and in the end we discuss our baseline microarchitectures. Our baseline microarchitectures also include basic support to enable a Cd-VM.

Chapter 4 "The Co-designed Programmable Functional Unit" describes mechanism related to instruction Fusion and Bulk Commit. A Programmable Functional Unit (PFU) is proposed that can execute a collapsed data flow graph. The code generation step that generates the macro-ops is also proposed. Moreover, since the superblocks are atomic in nature, a bulk commit mechanism is provided in the context of out-of-order processors. This work has been published as [31], [32].

Chapter 5 "SoftHV" deals with the issue of instruction fusion. For this purpose we chose two different classes of accelerators, one that executes collapsed instructions with low latency and the one that executes parallel instructions together. This study was performed in the context of in-order processor. The results shows that a co-designed in-order processor can outperform a small instruction window out-of-order processor. This work has been published as [34].

Chapter 6 "A Power-Efficient Co-designed Out-of-Order processor" deals with both instruction Issue and Bulk Commit. Since the Cd-VM provides an opportunity to migrate to a low-complexity low-power microarchitecture, a FIFO based out-of-order logic is chosen. The presence of limited out-of-order logic extracts the ILP that cannot be exposed by the dynamic optimizer. Moreover, a novel bulk commit mechanism is proposed that gets rid of the Reorder Buffer entirely; instead the order is maintained at the granularity of superblocks. This work has been published as [33].

Chapter 7 "Conclusions" revisits the objectives of the thesis and how we had achieved that. We review the key contributions of the thesis and discuss areas that are open for further research in the Future Work section.

Chapter 2

The Co-designed Virtual Machine

The previous chapter introduced the importance of high-performance processors with low-complexity and low-power consumption. It discussed the key objectives and the key contributions of the thesis. It also provided a brief description of Cd-VM, which is the design paradigm that this thesis follows to achieve the stated objectives.

In this chapter first we will revisit the motivation behind Cd-VM in Section 2.1. Next, we will describe the architecture of a Co-designed processor in Section 2.2. This will provide a background on Cd-VM that will enable to understand the rest of the thesis. This chapter also specifies exactly which techniques have been chosen from the large body of existing work. Microarchitectural support required in order to efficiently support the Cd-VM infrastructure will be discussed next in Section 2.3. In the end we will discuss the related work in Section 2.4.

2.1 Motivation

Both Hardware and Software have undergone radical changes over past few decades. At a time when hardware resources were expensive and hardware was simple, ISAs were a direct reflection of the underlying hardware implementation.

Advances in technology have made hardware resources more readily available and inexpensive. In order to use hundreds of millions of transistors, new microarchitectural innovations were made to exploit the application ILP. Furthermore, these innovations were achieved while retaining the software compatibility. Existing ISAs, such as x86, have complex CISC-like instructions, whereas modern processor back-ends typically execute simpler RISC-like μ-ops.

A modern out-of-order processor cracks CISC-like instructions into RISC-like μ-ops, on-the-fly, using complex decoders. The μ-ops are dynamically scheduled in the back-end,

using out-of-order execution logic, to exploit the ILP. A Reorder Buffer (ROB) is used to commit the μ-ops in the original program order. Register renaming helps in breaking the false dependencies, by using Physical Registers and Map-tables.

A Co-designed Processor, on the other hand, translates instructions from the *source ISA* to the *target ISA*. Binary translation and optimization provides high performance equivalent to that of an of out-of-order processor. As a result, a low-complexity and a low-power microarchitecture could be used instead. This emulation of *source ISA*, however, introduces performance overhead. Microarchitectural support is added to mitigate this overhead. Furthermore, additional microarchitectural support is added to enable various code optimizations.

Not only is the user code binary translated, but so is the OS code. Such a co-designed binary translation system is referred to in literature as a Co-designed Virtual Machine System [88].

The goals of a co-designed processor include performance, power-efficiency, and design simplicity by co-designing the processor in hardware and software. These goals can be achieved by introducing new microarchitectural features, or by changing the underlying microarchitecture entirely or by co-designing key performance enablers.

2.2 The Co-designed Virtual Machine Architecture

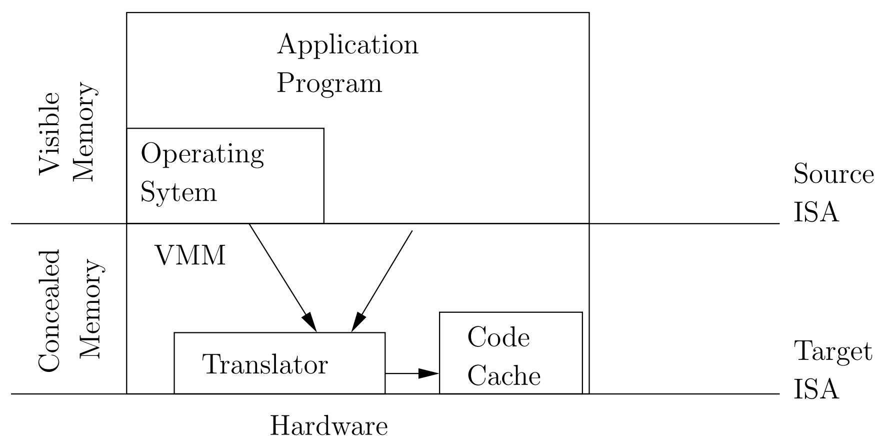


Figure 2.1: The Co-designed Virtual Machine Architecture Overview

Figure 2.1 provides an overview of the architecture of the Cd-VM. The Cd-VM consists of a software layer, known as a Virtual Machine Monitor (VMM), which is concealed from both the OS and the application program. At the very core of the VMM is a dynamic binary translator/optimizer that translates application and OS code from the *source ISA*

to the *target ISA*. A block of instructions are translated and stored as regions in a Code Cache.

Furthermore, not only the source ISA and the target ISA are different, but also the underlying *host microarchitecture* could be entirely different from the microarchitecture that the binary was optimized for - the later is known as the *guest microarchitecture*. Most of the prior work has co-designed the *source ISA* using a host *VLIW* microarchitecture and the corresponding VLIW *target ISA*.

In this thesis, we chose *x86 ISA* as the *source ISA* and *RISC-like μ-op ISA* as the *target ISA*. Furthermore, the *host microarchitecture* chosen is in-order and out-of-order. As a consequence of this cold x86 code can be emulated entirely in hardware, by cracking x86 instructions into μ-ops as is done in modern x86 microprocessors. The hot code, however, is binary translated to the μ-op ISA. These generated μ-ops are further optimized and stored in superblocks.

2.2.1 Basic Execution Model Overview

When the processor boots, it starts by fetching and decoding x86 code in the hardware. The x86 instructions are cracked into μ-ops using x86 decoders and the μ-ops are executed normally. This dual-mode execution is rendered using the dual-mode decoders proposed by Hu et. al. [53].

As a consequence of dual-mode decoders, and in order to reduce the run-time overhead of profiling, profiling hardware, as proposed by Merten et al. [71] is used. On-the-fly profiling helps in identifying both the hot spots in the code and in assisting with the code optimizations.

When a particular block has been executed a certain number of times the VMM is triggered and the optimizer generates a region of code. The region is formed starting at the first x86 instruction of the basic block that had triggered the profile event. The optimizer translates instructions from the *source ISA* to the *target ISA* of the *host microarchitecture*.

In order to reuse the translated blocks, they are stored in a *Code Cache*. After executing a Basic Block, control is transferred back to the VMM, which then determines the next Basic Block to be executed. In order to locate the next Basic Block map-tables are maintained by the VMM. Map-tables are indexed using the source PC (SPC) to locate the corresponding target PC (TPC) of the translation.

2.2.2 The Dynamic Optimizer

Dynamic binary *optimizer* is at the very core of the Virtual Machine Monitor (VMM). The *optimizer* is responsible for dynamically compiling and optimizing both the application

and the OS binaries from the *source ISA* to the *target ISA*.

The dynamic *optimizer* is invoked when a Basic Block is hot enough to be translated and optimized. For certain basic blocks that terminate in biased branches, the overhead of branching back to VMM could be mitigated by *chaining* to the Basic Block corresponding to the frequently taken edge.

The dynamic *optimizer* can further choose to improve the I-Cache performance by placing the chained Basic Blocks together in the memory. This results in a trace of Basic Blocks and is widely referred to as a Superblock [55].

The Superblocks

Superblocks have a single entry point and could either have multiple or single exit points. **In this thesis**, we use superblocks that have a single entry and a single exit point, such superblocks are referred to as *atomic* superblocks, as shown in the Figure 2.2. They are **atomic** because an atomic superblock either completes entirely or not at all.

The atomicity of superblocks is achieved by converting the internal branches of the superblocks into asserts. These assert instructions are not predicted by the branch predictor. The assert condition is tested at the execution stage by the ALU stage.

Moreover, in atomic superblock the intermediate state is not made visible to the outside world. If any of the outcome of any internal branch instruction in the superblock does not point to the appended basic block, then the superblock must be discarded. The execution is rolled back to the unoptimized version of the starting Basic Block.

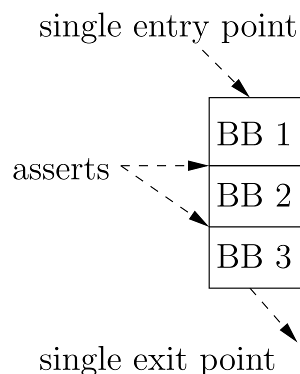


Figure 2.2: Atomic Superblock

This results in a disadvantage that correctly executed instructions within the superblocks are discarded along with the incorrect ones. However, being *atomic* also provides an advantage that various speculative code optimizations could be applied across the basic block boundaries, because superblocks will be rolled-back in case of an exception.

Staged Emulation

A typical binary translation system usually employs interpretation for the cold code, followed by basic block translation for hot code. Superblocks are created for basic blocks that are super hot. Furthermore, speculative code optimizations could be applied to the superblocks that are performance critical. Such a staged emulation technique has been implemented in Transmeta Crusoe [35].

In this thesis, as mentioned above, since the cold code is emulated in the hardware using the x86 decoders, the interpretation stage is not required. Moreover, we use a single stage optimization step that is triggered when a basic block is hot. The single step optimization forms superblocks and optimizes them. In any case there is no reason that prevents the proposed techniques in this thesis to be exposed to other scenarios such as full emulation[1] or multi-staged optimizers.

2.2.3 Frontend Support for Native x86 execution

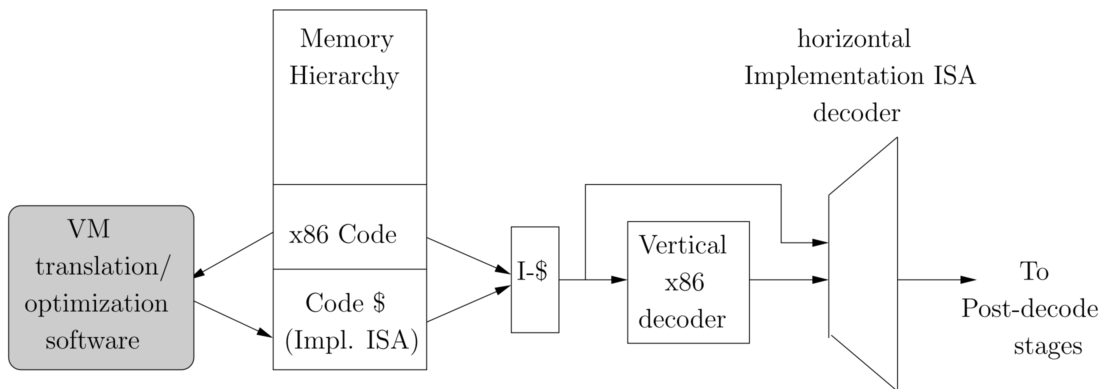


Figure 2.3: Dual Mode x86 decoder Frontend pipeline. Reproduced from [53]. The box in gray is the software part.

In this thesis, we use front-end dual-mode decoders [53] to enable native execution of cold basic blocks. Figure 2.3 illustrates the frontend pipeline of the co-designed processors used in this thesis.

This dual-mode decoder based frontend was proposed by Hu et al[53]. The vertical x86 decoder is the standard x86 decoder that is present in most modern x86 based processors. The horizontal μ-op decoders are added so that the μ-ops can be decoded directly bypassing the x86 decoders.

[1] When the cold code is interpreted.

The vertical x86 decoders are complex in nature and are power-hungry. However, our experiments have indicated that superblocks provide more nearly 90% of *code coverage*. Since these superblocks contains μ-ops, they are decoded by the horizontal μ-op decoder. As a result the vertical x86-decoders are powered-off most of the time, resulting in low power consumption.

2.2.4 Memory System Architecture

Figure 2.4 shows the memory system architecture of the Cd-VM used *in this thesis*. The ICache contains the translated code from the Code Cache, the VMM code and also code from the *Source ISA*. The code from the *source ISA* that enters the I-Cache is the cold code that was mentioned above. However, since this code is cold, it will eventually get replaced. Moreover, when a cold code becomes hot it will be translated, and its translations will be fetched into the I-Cache instead.

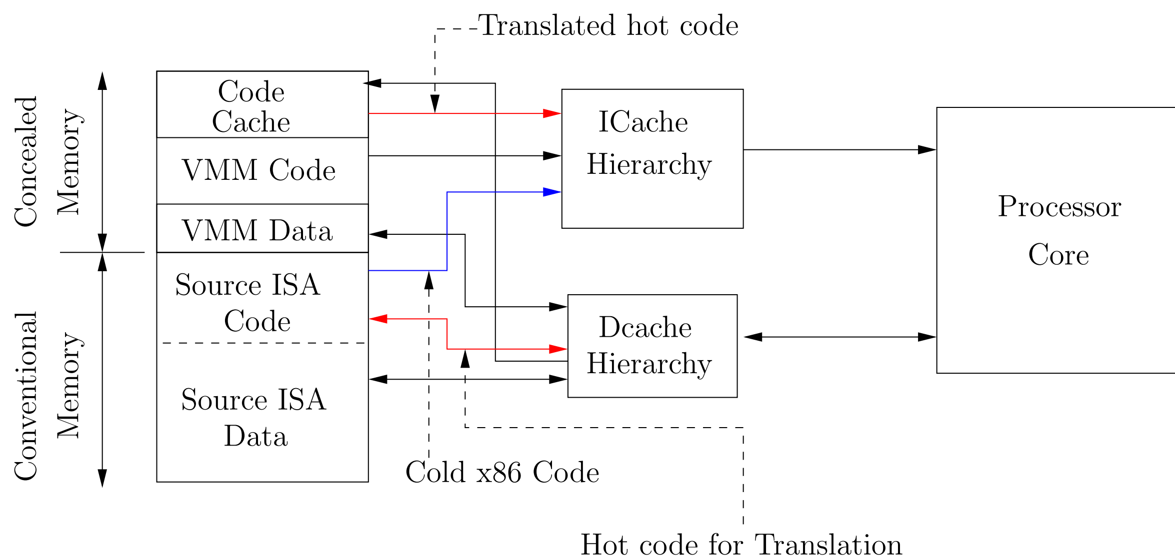


Figure 2.4: Memory System Architecture

Furthermore, as shown in the Figure 2.4, the VMM code, the VMM data and the Code Cache resides in a concealed memory region. Memory region can be concealed by providing a separate logical address space to the VMM. If a separate logical address space is used then the VMM handles its own page tables, whereas the OS handles only the architected page tables. Instead of using a separate logical address space, the VMM code can use the physical address directly. Special memory μ-ops are introduced that when executed do not access the TLB.

In this thesis, we use unused memory region from the applications address space to simulate the effect of concealed memory. The only component of the concealed memory

we simulate is the *Code Cache*. The translated code is transferred to the I-Cache from these memory regions. Moreover, as will be described in Chapter 3, since we do not actually context-switch to the VMM code, no separate page tables were required.

2.2.5 Code Cache

The translated code regions are stored in the *Code Cache*, which along with VMM code and data are mapped into a special memory region. This memory region is concealed from both the Operating System and the Application, as shown earlier in the Figure 2.4.

In this thesis, the translated regions are superblocks, which are stored in the *Code Cache* [88]. Moreover, as described earlier the *Code Cache* is held in an unused memory region in the application's address space.

On the other hand, since the *code cache* has a limited size, a *replacement policy* is required to replace older translations with the newer ones.

In this thesis, we assume an unbounded *Code Cache*. Given our experiments focus on single-threaded applications, with small dynamic instruction footprint, this does not invalidate our conclusions.

2.3 Microarchitectural Support

In this section, first we will discuss the microarchitectural support added to reduce the overhead of the VMM. Next, we will discuss microarchitectural support required to enable speculative code optimizations. Next, we will discuss an existing mechanism to deal with bulk commit of *atomic* superblocks and the mechanism to recover from misspeculation. In the end, we will discuss the hardware profiling support used in order to determine hot spot of the code.

2.3.1 Reducing Overhead in Co-designed Virtual Machine

As mentioned above, the Cd-VM helps in migrating to a low-complexity and a low-power processor. Moreover, equivalent performance is obtained with the help of dynamic code optimizations. However, the Cd-VM comes with a price, as the VMM time-shares the CPU with the application and the OS. The VMM spends a major fraction of its execution time optimizing and translating superblocks, which we refer to as *translation overhead*.

Furthermore, every time a branch instruction is executed, the branch target PC is used to look-up in a table to find the corresponding translated region. When this look-up is

done entirely in software it leads to a *look-up overhead* of dozens of instructions.

Reducing Translation Overhead

One of the major sources of overhead is due to binary translation from the source ISA to the target ISA. S. Hu et al. [53] have categorized and quantified this overhead. They pointed out that start-up overhead comes due to Basic Block Translation and Super Block Translation.

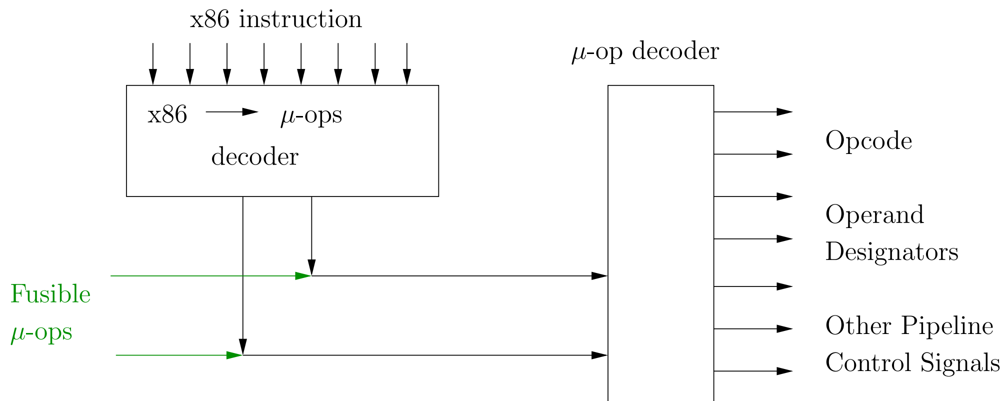


Figure 2.5: Dual Mode x86 Decoder. Reproduced from [54].

However, they also observed that the start-up overhead is primarily due to the Basic Block Translation. In a processor equipped with dual-mode decoders, this translation overhead is eliminated. For the sake of convenience, we reproduce the dual mode decoders in Figure 2.5 as proposed by Hu et al. in [54].

In this thesis we have focused on co-designing the x86 ISA, we use these dual-mode decoders to execute cold x86 cold natively.

Reducing Translation Look-up Overhead

Sources of overhead due to frequent branching between VMM code and the source binary is mitigated by using translated basic blocks and superblocks. However, when a translated superblock completes its execution it transfers control back to the VMM. The VMM then looks-up in a table to find the corresponding mapping of the next superblock/basic block. This table look-up code result in an overhead of upto dozens of instructions [88].

This translation look-up overhead is reduced by adding a hardware known as a Jump Translation Look-aside Buffer (JTLB) [47, 59]. Figure 2.6 shows how the SPC is hashed and presented to the JTLB. JTLB is entirely managed by the VMM by introducing

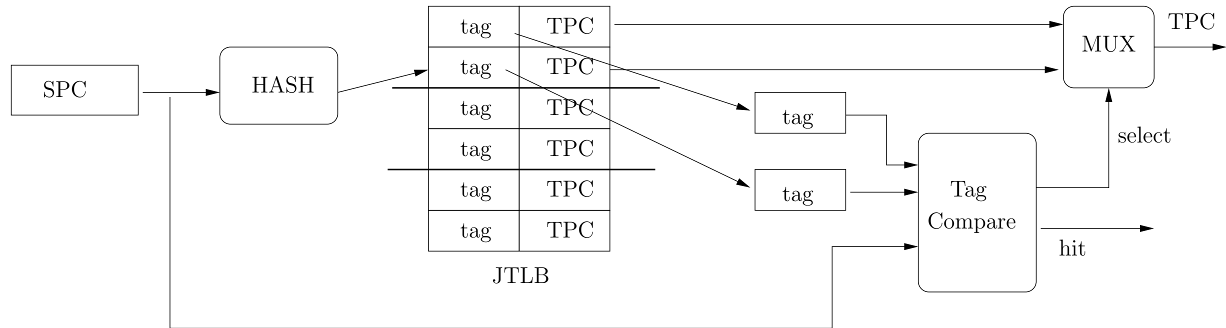


Figure 2.6: The Jump Translated Lookaside Buffer (JTLB)

a new Lookup_jump μ-op that when executed performs the necessary look-up. Tags are compared and on a hit the corresponding TPC is returned to the Lookup_jump instruction.

Since, the Lookup_jump μ-op accesses the JTLB only at the execute stage to determine the TPC, the processor have to stall the fetch until TPC is determined. In order to get around this problem, the Lookup_jump μ-op is predicted like any other branch. However, the existing prediction hardware which consists of Branch Target Buffer holds only the predicted SPC.

In this thesis, we have assumed unbounded JTLB and unbounded *Code Cache*. As a result of which, if a translation is not found, it simply means that the SPC corresponds to a cold Basic Block. Instead of extending the BTB, the *predicted SPC* returned by the BTB is used to look-up for the *predicted TPC*. This method results in an effect similar to that of extending the BTB.

2.3.2 Microarchitectural support to handle Code Optimizations

Various memory related code optimizations are implemented. For instance, Load-hoisting [39] hoists loads above stores, Load-Store Telescoping [39] bypasses a load by forwarding data from the producer μ-op, of an aliasing store μ-op, directly to the dependents of the load. Load hoisting is speculative in nature as it assumes a pair of store and load not to alias. An alias hardware [57] is added in the microarchitecture in order to detect a miss-speculation. In case of a miss-speculation the above-mentioned recovery mechanism is used to rollback and execute the unoptimized version of the superblock.

As mentioned above we use superblocks that have a single entry and a single exit point,

as shown in Figure 2.2. As a consequence, all the branches except the exit branch are converted to asserts [77]. Asserts are special instructions which are executed in an ALU and tests a condition. If the condition is false, the assert is said to have failed. If an assertion fails or any other exception has occurred, the superblock is rolled back.

2.3.3 Miss-speculation Recovery and Bulk Commit mechanism

On-the-fly profile information helps in forming dynamic regions and in speculatively optimizing code regions. However, in order to support these optimizations, the underlying microarchitecture should provide mechanisms to detect miss-speculations and take the necessary corrective action. The general recovery mechanism is depicted in Figure 2.7. As shown in the figure, when an instruction of Superblock 2 traps then the check-point is restored and the unoptimized version of the code is fetched and interpreted.

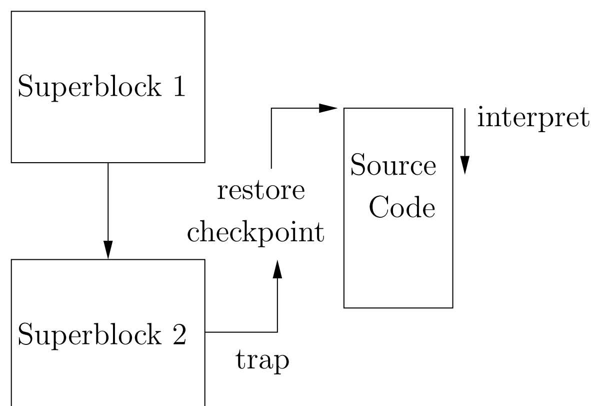


Figure 2.7: General Recovery Mechanism

The corrective action depends on the underlying microarchitecture. For instance, for an in-order microarchitecture a shadow copy of the register file and working copy can provide the support for checkpointing the register state. In case of a misspeculation, the execution is rolled-back and the shadow copy is copied back to the working copy and the code is interpreted.

The memory state is held in Gated Store Buffer [95] and committed at bulk. Its contents are discarded in case of a misspeculation.

In this thesis we had proposed a couple of bulk commit mechanism in Chapter 4 and in Chapter 6. Our solutions are applicable to out-of-order processors, whereas the Transmeta solution can only be applied to in-order processors. In Chapter 5, since we have proposed an in-order processor we use the above-mentioned solution based on shadow copy and working copy to implement effective bulk commit and rollback. For all our chapters we use Gated Store Buffer to hold the store data corresponding to a superblock.

2.3.4 Profiling Support

In this thesis, in order to reduce the run-time overhead of profiling, profiling hardware, as proposed by Merten et al. [71] is used. We collect basic block profiles and edge profiles for each branch instruction. Figure 2.8 illustrates the difference between block and edge profile. The basic block count is incremented whenever it is a target of a branch being committed.

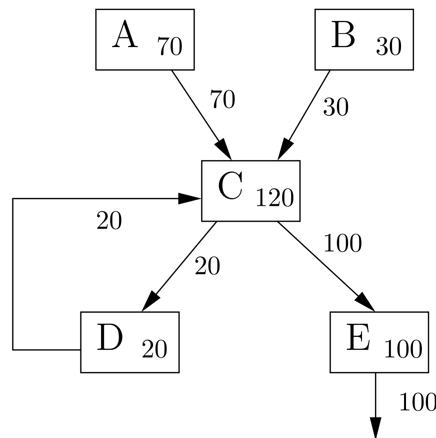


Figure 2.8: Basic Block and Edge profile of a Control Flow Graph. The count inside the boxes represents the number of time the basic block has executed. The count next to edge indicate the number of times the edge was taken.

Similarly, edge count –corresponding to taken or not-taken edge– is incremented when the branch commits. Since in out-of-order processors branches could execute speculatively, counters are not updated when branches execute. Updating the counter at execute will lead to erroneous profile information. Hence all the updates are done when branches commit.

2.4 Related Work

2.4.1 Virtual Machines

Binary Translation Systems and Virtual Machines have existed since the dawn of computing. Smith and Nair [88] have categorized virtual machines into several categories. These are Process Virtual Machines, High Level Language Virtual Machines, Co-designed Virtual Machines and Classic-System Virtual Machines. Although, these Virtual Machines have some overlapping objectives, but each of them serve a unique purpose.

Process Virtual Machines are those that run on top of existing OS and run application binaries that may or may not be compiled to the host ISA. Same ISA Process VM, such as HP Dynamo [9], are essentially binary optimizers, with their primary goal being performance. Different ISA Process VM, such as FX!32 [22], target compatibility for existing application binaries.

High Level Language Virtual Machines have become quite pervasive of late and there are a plethora of them. Examples are JAVA, Python, C# and its .NET framework. In a sense these VMs are a much more cleaner solution to the problem that FX!32 addresses. They have been built around the philosophy of write-once and run-anywhere software development. These systems first compile the source code into a bytecode, and then interpret or binary translate the bytecodes.

Classic-System Virtual Machines serve a very different purpose. The original goal was to provide a time-shared system to each user. Over the years, they grew into providing multiple virtual machines running on a single physical machine. Such a system has grown extremely popular in the IT domain, where the cost of managing machines is higher. Xen [11] and VMware [92] are two popular examples of such systems.

Co-designed Virtual Machines, on the other hand, serve a very different purpose. The key philosophy behind Cd-VM is to co-design the source ISA partly in software and partly in the hardware. This helps in cutting down power consumption and processor complexity while maintaining equivalent performance.

HW/SW co-design is a part-and-parcel of innovations in microarchitecture. A Co-designed approach has primarily been used in order to support legacy ISA features, to cut down the processor complexity and the area cost. Microcodes [93], for instance, are used in order to achieve this. Furthermore, they have also helped in reducing the implementation life-cycle by fixing HW bugs in the microcode, for instance the infamous FDIV bug [27]. Microcodes, however, have their limitations, as they cannot perform aggressive optimizations, which handicaps it from introducing drastic modifications to the microarchitecture.

2.4.2 IBM DAISY

IBM DAISY [39] and BOA [7] projects implemented the Power PC ISA using a dynamic optimizer and a VLIW ISA. The generated VLIW instructions are scheduled by the VMM at the scope of tree-regions, superblocks or atomic superblocks. This provides a much better scope in scheduling than that provided by basic blocks. Using a VLIW microarchitecture drastically cuts down the processor complexity and the power consumption. As a result a higher frequency architecture is obtained, with a shorter pipeline. Various code optimizations [88] such as instruction scheduling, combining, copy propagation, dead code elimination, load-store telescoping are performed in software.

In order to avoid fetch related stall, the branch predictor is accessed to predict the target

PC. Branch predictor structures are modified to hold the target PC along with the source PC. For instance the Branch Target Buffer (BTB) and the Return Address Stack is augmented to hold the target PC as well. In case the source to target PC translation is not available in the JTLB, control is returned to the VMM and a look up is performed into the Source to Target Map Table.

In contrast to the DAISY and BOA projects, we propose both an in-order and an out-of-order microarchitecture in this thesis. Moreover, we propose atomic superblocks as our translation regions. Furthermore, since DAISY does not roll back from the middle of region execution, it does not require a bulk commit mechanism. We propose Bulk Commit Mechanisms for out-of-order co-designed processors. Moreover, using the dual-mode decoders helps us in getting rid of cold code interpretation. Furthermore, the *guest ISA* used in DAISY was the **PowerPC ISA**, whereas in our case we co-design the **x86 ISA**.

2.4.3 Transmeta Crusoe

Transmeta Crusoe [60] is a commercial co-designed processor that uses a translator known as the Code Morphing Software (CMS) [35] to perform multi-staged emulation. Shadow copy of Register File is used to check-point the register state before a superblock starts executing. The working copy, as the name suggests, holds the working register set of the superblock. The memory state, however, is held in gated store buffers [95]. A special commit operation updates the register and the memory state, at once. Similar to DAISY [39] and BOA [7], Crusoe uses VLIW processors to cut down complexity and power. Aliasing HW is added to [57] detect any memory ordering violation.

Transmeta Efficeon, which is a second generation co-designed processor, proposes a speculative cache based solution. Stores write to the cache and a speculative written bit is set. This provides superblocks to have large number of stores.

In contrast to Transmeta's VLIW microarchitecture we propose in-order and out-of-order microarchitectures. These microarchitectures execute the RISC like μ-ops. As a consequence, the cold x86 code is cracked to μ-ops using standard x86 decoders. Moreover, since we also propose a co-designed out-of-order microarchitecture, two novel bulk commit mechanisms are proposed as well. Furthermore, we looked into the benefits of accelerating parts of applications using specialized functional units.

2.4.4 HP Dynamo

Dynamo is a source to source binary optimizer that optimizes the application binaries. In contrast to Cd-VM, it simply optimizes the application binaries. These kind of binary optimizers are widely referred to as Process VMs in the literature [88]. In contrast to

Cd-VM regions, Dynamo's regions last only for a single invocation of a program.

The source to source binary optimization provides an opportunity to run code natively, if required[2]. In existing Cd-VM this feature is not available, because the cold code has to be interpreted. In contrast, in this thesis, since we use dual-mode decoders, our co-designed processor can also run the cold cold natively. Furthermore, Dynamo does not perform speculative code optimization. This is because it runs on unmodified conventional processors. However, the hardware could be modified to enable such optimizations.

2.4.5 Reducing Translation Overhead

Hu et al. have proposed a dual-mode decoder[52], which eliminates the basic block translation overhead. Cold x86 instructions are cracked into μ-ops on-the-fly, whereas μ-ops from superblocks bypass the x86 decoders. In this thesis, we have used these dual-mode decoders to co-design the x86 ISA efficiently.

Moreover, special HW assists can be added to further reduce the overhead of Superblock translation [53]. A special instruction translates x86 instructions into μ-ops. In this thesis, we have not considered using any such kind of assists. This is because in thesis our focus was not reduce the overhead of the VMM, but to propose new co-designed microarchitectures. However, if such assists are used then the superblock translation overhead that we have measured can further be reduced.

2.4.6 Optimizer and Interpreter

An *optimizer* could either interpret each source instruction or binary translate a basic block. Interpretation usually is easier to implement but it involves the overhead of branching back-and-forth between the interpreter routines and the VMM. This overhead, however, can be mitigated by branching directly to the interpreter routine of the following instruction. Such a kind of interpretation is known as threaded interpretation [12, 61].

However, there is still a branching penalty for branching to each instruction's interpreter routine. Binary translation [69, 86], on the other hand, dynamically re-compiles a basic block from the source ISA to the target ISA. Clearly, binary translation has a higher start-up overhead, however, it could be amortized if the basic block is executed sufficient number of times.

In this thesis, we do not have an interpreter. The superblocks are formed and optimized just once.

[2] For instance, when performance degrades due to the optimizer

2.4.7 Superblock Heuristics

There are multiple trace formation heuristics in the literature [42, 64]. We use Superblocks [55] proposed by Hwu et al in 1993. Superblocks are like trace but do not have side entrances and side exits. There are multiple heuristics to form superblocks [23, 38, 21, 14].

In this thesis, we have used the heuristic, described later in Section 3.3.3, which starts with a hot Basic Block and then keeps appending Basic Blocks that are target of the frequently taken edges. This process goes on until a stopping condition is met. Once a superblock is formed it is optimized, and then placed in the *Code Cache*.

2.4.8 Hardware Dynamic Optimizers

Performing dynamic optimization in a user transparent manner need not be done by a software layer only. Several hardware approaches have been considered [77, 82]. The main difference between Cd-VM scheme and RePlay[77] is the flexibility of the software layer to perform different optimizations and analysis with reduced hardware complexity.

2.5 Summary of Design Choices

Table 2.1 lists the design choices that were made for the Cd-VM system in this thesis.

Parameter	**Choice Made**
Source ISA	x86 ISA
Target ISA	μ-op ISA
Translation Regions	Atomic Superblocks
Superblock Heuristic	Described in Chapter 3
Cold Code Execution	Dual-mode decoders enable native execution
Emulation Strategy	Cold code native execution, Hot code binary translated to Superblocks
Microarchitectures	In-order and out-of-order
Bulk Commit	Proposed in Chapters 4, 6
Code Optimizations	Described in Chapter 3
Profiling Support	Hardware support for block and edge profiling
Code Cache Size	Unbounded
JTLB Size	Unbounded
Processor Frequency	2-3 GHz

Table 2.1: Summary of Design Choices made for the Cd-VM system in this thesis.

Chapter 3

Experimental Methodology

The previous chapter provided a general background on Cd-VM, required to understand the rest of the thesis. Moreover, in the previous chapter, we also highlighted some of the key features of the Cd-VM that is used in this thesis.

The current chapter builds upon the previous one by presenting implementation level details of our Cd-VM infrastructure[1]. We mainly discuss the experimental methodology and how we simulated a Cd-VM. All of the research was conducted using PTLsim [99], an execution-driven cycle-accurate microarchitectural simulator for x86 ISA.

We have evaluated our proposed schemes using the SPEC2000 benchmark suite. These benchmarks have been compiled with GCC version 4.1.3 using -O3. Using the developed infrastructure, we have simulated the benchmarks for 100 million x86 instructions after the initialization and a cache warm-up period of 2 million x86 instructions.

In this chapter, first in Section 3.1 we will discuss the reasons behind choosing PTLsim in Section. Next in Section 3.2 we will discuss the enhancements made to PTLsim in order to make it more cycle-accurate. Then in Section 3.3 we will describe the Cd-VM monitor implementation. This description includes the Cod Generation process, Fetch mechanism, profiling and VMM invocation, and some statistics related to superblock size and the corresponding translation overhead.

Next in Section 3.4 we will discuss the limitations of our implementation. Finally, in Section 3.5 we will describe the baseline microarchitectures used in this thesis. This includes the descriptions of the baseline in-order and out-of-order microarchitectures.

---

[1]Parts of this infrastructure were developed jointly with Indu Bhagat, another PhD student of our research group ARCO [1]. However, the joint work focused only to establish an initial version of PTLSIM in order to run a co-design virtual machine. The collaboration did not include either the superblock formation presented in this thesis, or any of the basic code optimizations presented in this chapter, nor any of the proposals done along the thesis.

3.1 The Choice of Microarchitecture Simulator

There are several reasons for which PTLsim was used. Firstly, since the PTLsim implements the x86 ISA, it comes with support for binary translation from CISC-like x86 ISA to RISC-like μ-op ISA. This binary translation support comes with a Basic Block Cache[2], in order to reduce translation overhead of **simulation**. Since the translated Basic Blocks can be used to form larger regions such as Superblocks, PTLsim becomes a natural choice to implement a Cd-VM infrastructure.

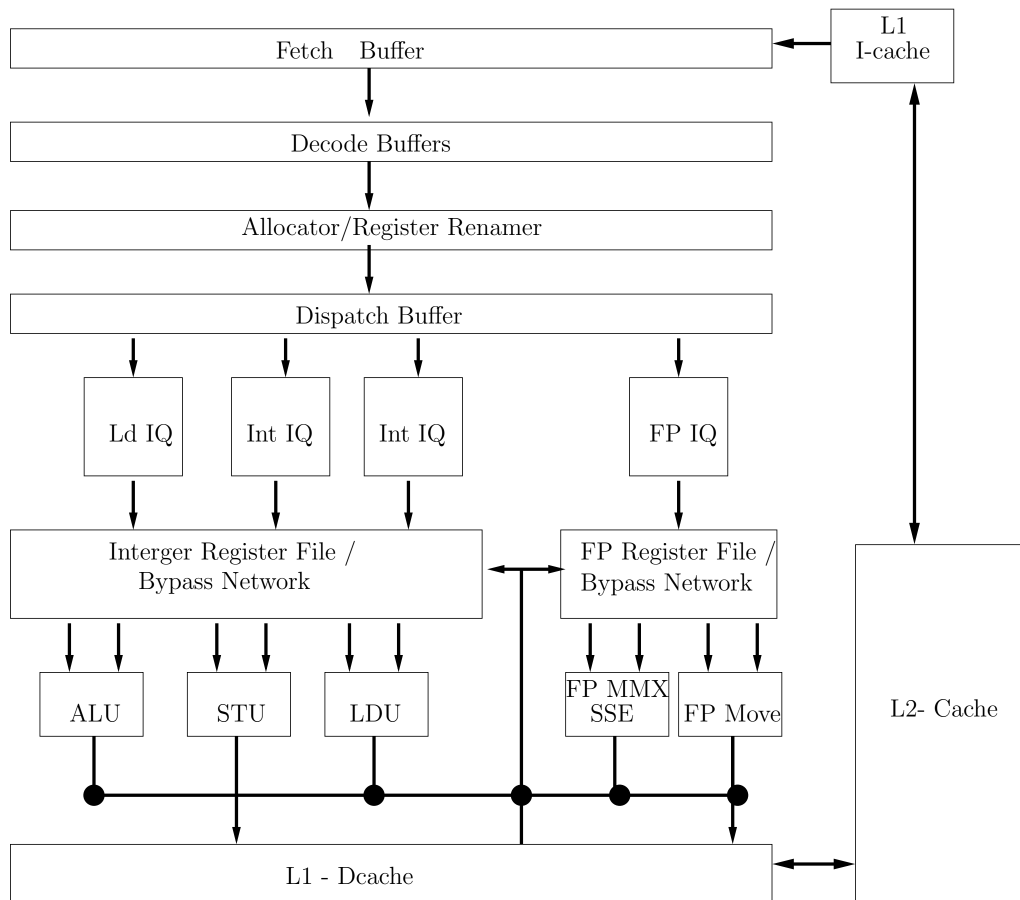


Figure 3.1: Block diagram of the Microarchitecture that PTLsim models.

Secondly, even though PTLsim is an execution-driven simulator, it is very fast. The interpreter routines of the μ-ops are implemented in hand-coded assembly. As a result of which PTLsim cuts down the overhead associated with μ-op interpretation drastically. Moreover, the timing simulator is implemented efficiently by using associative data struc-

[2]This is a cache maintained by PTLsim for the purpose of efficient simulation, not to be confused with a Code Cache.

tures for hardware structures such as Issue Queues, Load Store Queues that in reality also perform an associative search.

Thirdly, PTLsim is a cycle-accurate simulator and its performance has been correlated with an AMD K8 processor. The results were found to be within 5% for all the key performance metrics, such as IPC, L1-miss rate etc.. More details on the correlation study could be found in [99].

Figure 3.1 shows the microarchitecture that PTLsim models. As shown in the figure, it has features similar to AMD K8 [56] and the Netburst Microarchitecture [51]. Multiple issue queues are present that along with the functional units define an execution cluster. Floating point μ-ops are dispatched to floating point Issue Queue. Loads are dispatched to Load Issue Queue, whereas other integer instructions are dispatched to either of the integer Issue Queues.

Separate floating point and integer register files are maintained. A Reorder Buffer is present in the backend to let the μ-ops commit in program order. Moreover, as will be shown later in Figure 3.11 two separate rename tables are held in the frontend and in the backend of the processor. The Frontend Register Rename Table (FRRT) holds the speculative state, whereas the Backend Register Rename Table (BRRT) holds the committed state.

3.2 Timing Simulator Enhancements

Although, as stated above the timing simulator was quite accurate, there were few parts where simplifying assumptions were made. We looked into some of these parts and modified the simulator to make it more realistic by incorporating features from various commercial processors such as AMD K8 [56] and Alpha21264 [58]. We also integrated the Wattch power model into the PTLsim. The enhancements to the timing simulator are generic and so they can be merged into the main PTLsim branch.

3.2.1 Modifying the Execution Pipeline

Firstly, the execution pipeline in the PTLsim had made several simplifying assumptions. In order to understand the simplifications that were made, first we will describe the behavior in modern microarchitectures. In modern out-of-order processors a load is known to hit in the final stage of execution [58]. In order to support back-to-back execution the dependents of the load are issued speculatively, assuming that the load would hit. However, if the load misses, then all the instructions that were issued in the current cycle and the previous one are squashed, as shown in the Figure 3.2. As a result, instructions are either replayed or held in the issue queue for two cycles.

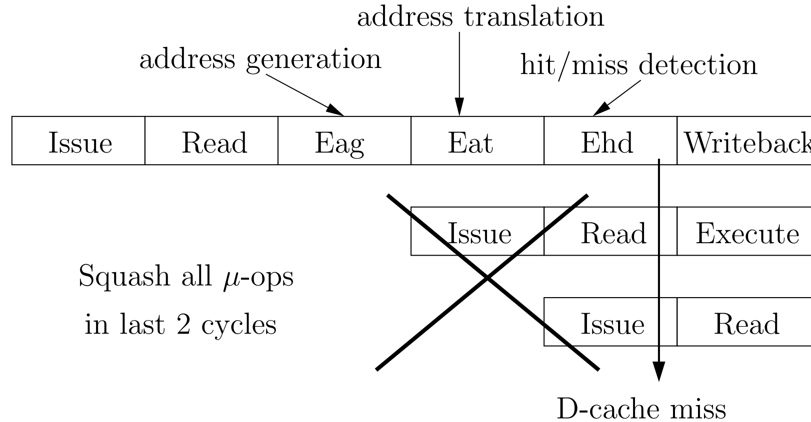


Figure 3.2: Illustration of Load hit Speculation. In this example all μ-ops issued in the current cycle and the previous one are squashed. For this purpose μ-ops are held in the issue queue for couple of cycles after being issued.

This particular mechanism was not implemented in PTLsim. Instead at the time when a load issues, the load determines whether it will hit or not. As a result, loads are not issued from the issue queue until the miss is resolved. However, in reality a missing load should instead be waiting in the Load Queue after placing a request with the memory hierarchy. We had implemented this whole feature in the timing simulator.

3.2.2 Simplifying Memory Disambiguation

Secondly, the baseline memory disambiguation scheme was more aggressive than is implemented in any commercial processors. Since the baseline simulator did not model the execution pipeline described above in Section 3.2.1, the memory disambiguation stage was performed at issue itself.

This implies that whenever a load issues, the Store Queue is checked in order to find if it aliases with any older store. If the load did alias with any older store then the load was not issued. However, in modern processors the memory disambiguation is a separate pipeline stage and coincides with the miss detection stage of Figure 3.2. As a result, when a load aliases it is removed from the execution pipeline and placed into the load queue. We have implemented this feature in our memory disambiguation scheme.

Moreover, the baseline memory disambiguation scheme also implemented a speculative load execution. In this scheme, if at the miss detection/memory disambiguation stage a load finds an older store with unresolved address, the load still completes and is placed in the ROB.

Now when the older store reaches its memory disambiguation stage, the Load Queue is

checked in order to find out whether the store aliases with any younger load. If it is the case then the younger load is marked as invalid. Finally, when this marked load reaches the head of the ROB and is about to commit, the load and its dependents are re-dispatched into the issue queue for re-execution[3].

We instead chose not to implement such a speculative load execution. Our scheme relies on the assumption that a load aliases with all of the unresolved stores, and is described as follows. Whenever the load finds any older store with address unresolved, the load is placed in the load queue.

Eventually, when the store issues it wakes up the load from the load queue. The load issues and checks again for any other aliasing or unresolved store. If the load does not find any such store, then the load completes. Moreover, in order to speedup the critical path the store address and store data are split into two operations. This allows stores to have their addresses resolved faster, which in turn helps loads to complete faster.

Furthermore, the baseline memory disambiguation scheme allows merging of data from several aliasing stores. The youngest store of a series of aliasing stores always has its data merged with the older ones. As a result, the load consumes the data from the youngest store. However, in modern processors such a scheme is quite expensive to implement. Moreover, such a case when a load aliases partially with multiple stores is not that frequent.

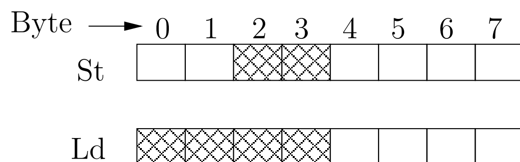


Figure 3.3: Store to load forwarding restrictions. In this example, the older store instruction write two bytes, whereas the load requires four bytes. This implies that the load has to wait until store retires.

Precisely, for these two reasons AMD K8 states in its optimization guide [56], about which kind of store-load aliasing should be avoided for performance reasons. In case if an older store does not fully alias with a load, as shown in the Figure 3.3, then the load is not executed until the store has retired from the pipeline. We implemented the store load aliasing restrictions as mentioned in the AMD K8 Optimization Guide [56].

[3] Instead of re-dispatching the μ-ops, an easier alternative is to re-fetch the instructions following the load.

3.2.3 Write Ports

Thirdly, we had introduced write ports in the Physical Register File. In the original implementation the model was not accurate as contention on write ports was not taken into account. With our implementation, when an instruction issues it not only checks whether the functional units would be available, but also whether the write ports would be available at the write back stage.

3.2.4 Integrating Wattch into PTLsim

Wattch [16] provides an infrastructure to measure power consumption of modern out-of-order processors. It has been shown that Wattch can provide power estimates with an accuracy of 10%. However, Wattch was implemented for simplescalar, which models power of an obsolete out-of-order microarchitecture and is tightly integrated to simplescalar.

We had modified Wattch to model hardware structures that are used in PTLsim. As mentioned earlier the modified version of PTLsim models behavior of a modern out-of-order processor. An ROB is modeled using a SRAM structure. Both the Frontend Register Rename Table and the Backend Register Rename Table are modeled using SRAM structure as well. Issue queues and LSQ are modeled using CAM based structures.

On top of the baseline microarchitecture we have also modeled the structures that we had proposed in Chapter 6. These modeling details are described in that chapter.

However, Wattch only reports dynamic power consumption. As a result, we also only report dynamic power consumption. Static power consumption is one of the major sources of power consumption in current process technology. We have, however, reported the numbers of 0.8μm technology.

Moreover, we have only modeled power consumption for L1 and L2 caches and not beyond that in memory hierarchy. Power consumption due to structures such as Load Fill Request Queue and Miss status handling register is not modeled. Other structures have been modeled, as explained above.

3.3 Virtual Machine Monitor Implementation

In this section we will describe the Virtual Machine Monitor implementation. We will describe the profiling mechanism, the code generation process, and the mechanism to fetch superblocks from the *Code Cache*. Next we will present some statistics related to the superblocks and show a sample superblock that is executed in our simulator.

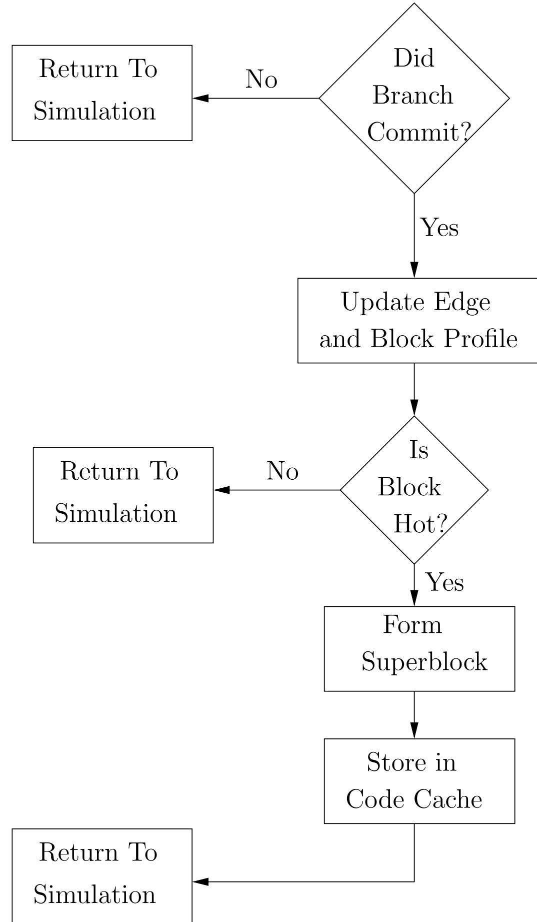


Figure 3.4: Profiling and Invoking Optimizer Flowchart.

3.3.1 VMM Optimizer Overview

Figure 3.4 illustrates when profiling information is updated and how superblock formation is invoked. In reality the superblock formation would require a context switch to the VMM. The VMM code should then be simulated on the microarchitecture simulator. For the sake of simplicity, we do not do this, instead we form the superblock when the branch commits. However, the overhead of superblock translation is measured and presented later.

Profiling support is added to the simulator to collect edge profile and Basic Block profiling data. Whenever a Basic Block executes a number of times given by the *hotspot threshold*, the superblock formation is invoked by trapping into VMM. The Superblock formation heuristic is described later in Section 3.3.3.

Once a Superblock is formed, control is transferred back to the simulator. In a real

implementation whenever the control is transferred to VMM to form the superblock there is a context switch, and the VMM code runs on the processor. We do not simulate the VMM code executing on the processor. Instead, we simply invoke our binary translator that forms optimized superblocks.

However, we do measure the overhead related to superblock translation in terms of instructions executed to optimize each source instruction. Later in Section 3.3.5 we measure the overhead due to superblock translation and optimization.

The translated superblocks are stored in an unused memory region of the process's address space. I-cache contains μ-ops from these memory region along with cold x86 instructions, which is stored in the usual code segment of a process. However, in the steady state, the I-cache mainly contains μ-ops from translated superblocks. This helps in modeling the I-cache behavior accurately.

3.3.2 Fetching μ-ops from Superblocks

In a real implementation a Lookup_jump instruction looks-up into the JTLB to find a corresponding SPC to TPC translation. Every time the Lookup_jump misses the control is transferred back to the VMM which then a looks-up in a software Map Table to find the TPC.

We do not implement a software table look-up, instead we assume an unbounded JTLB and code cache. So in-case there is a JTLB miss, it implies that a cold Basic Block is to be executed. The cold x86 code is executed natively by decoding them using frontend dual-mode decoders, as mentioned earlier in Section 2.3.1.

Moreover, in reality the Lookup_jump instruction when executes, looks-up in the JTLB to find the TPC. We, however, assume that the JTLB look-up is performed at fetch itself using the predicted SPC as described below.

In reality a Lookup_jump instruction's PC is used to predict. The branch predictor structure, mainly the BTB is extended to hold TPC along with the SPC. For the sake of simplicity we do not modify the BTB; instead we use the predicted SPC to look-up in a map table to find a corresponding TPC, as shown in the Figure 3.5. The resultant effect is the same as if the BTB were extended. If TPC exists then μ-ops are fetched from the superblock, otherwise cold x86 instructions are fetched.

3.3.3 Code Generation

This section describes the binary translation process that generates the optimized superblocks. The code optimization process implemented is shown in Figure 3.6. However, some of the steps such as, Code Optimization and List Scheduling are only present for

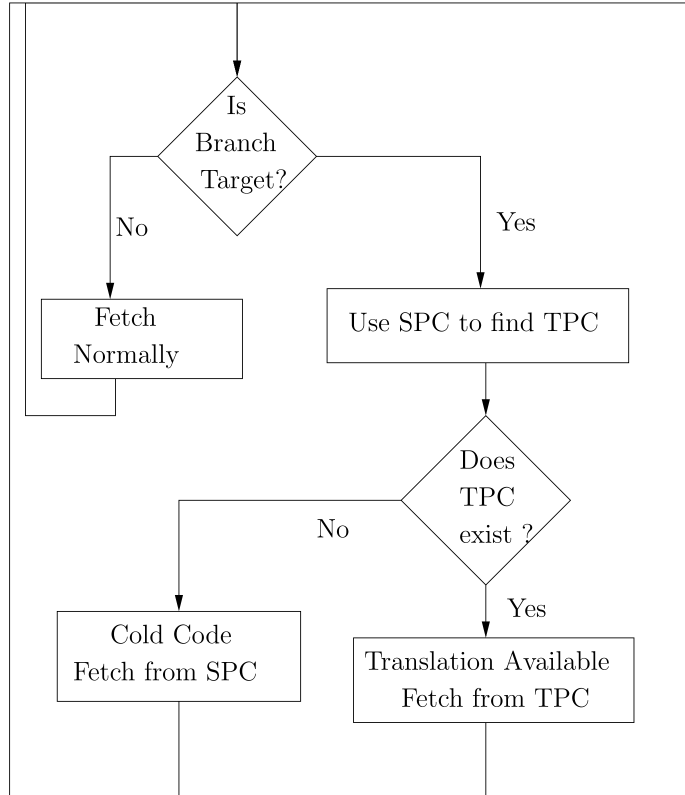


Figure 3.5: Fetch Flowchart.

Chapter 5 and Chapter 6. All the remaining steps are common to all the chapters. Since Chapter 4 and Chapter 5 deal with instruction fusion, they have an additional fusion step as well.

First, we will first briefly explain our superblock formation technique. Next, we will describe the code analysis and data-flow graph generation step. This step is a precursor to the code optimization step. Register allocation and final code generation follows in the end.

Superblock Formation

In this thesis, we use atomic superblocks as our translation region. As was described in Section2.2.2, in the Figure 2.2, atomic superblocks have a single entry point and a single exit point. This is achieved by inverting the sense of those internal branches whose taken path is included. Moreover, all the internal branches are converted into asserts [77].

These atomic superblocks require the program state to be committed in bulk, only when all the μ-ops within the superblock have successfully executed. Special hardware support

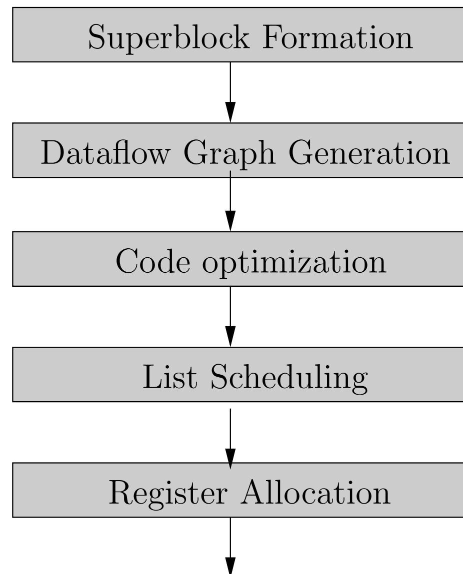


Figure 3.6: Code Generation Flow Chart

is implemented to support bulk commit and rollback for these atomic superblocks. The atomic property, however, enables several speculative code optimizations that are difficult to achieve in normal superblocks.

There has been vast amount of research dedicated to heuristics of forming superblocks. However, most of the heuristics are quite similar to each other, but vary from each other in subtle ways. Most of the heuristics start building a Superblock when a Basic Block has been executed a certain number of times - given by the hotspot threshold. The branch target of the Basic Block is used to find the next Basic Block to be appended. The heuristic continues in this fashion until a stopping condition is met.

Our superblock generation heuristic is a mixture of some of the heuristics proposed in the past. Before describing our superblock formation heuristic, we describe the key metrics that we used to guide the heuristics. For instance, a good choice of the atomic superblock would be the one that completes executing successfully most of the time. This leads us to define a metric *completion rate*, which is the probability of the superblock completing successfully. We use this metric to stop appending a Basic Block to the Superblock, if by adding the Basic Block lowers the *completion rate* of the Superblock.

Moreover, since superblock translation is a major source of overhead, one has to consider only those Basic Blocks that are **hot**. For this we use a widely known metric the *hotspot threshold*. The *hotspot threshold* indicates whether a Basic Block is hot enough to be considered as a *starting point* of a new superblock.

A lower *hotspot threshold* results in generating too many superblocks, thereby increasing

the superblock translation overhead. Whereas, a higher *hotspot threshold* results in forming too few superblocks, resulting in a lower code coverage. The implication of this is that performance benefits provided by superblocks will be lost. Hu et al. [54] had analyzed this problem and came up with an analytical model to determine a threshold value. We, however, select a threshold value empirically, which is similar to the one used by HP's Dynamo [9].

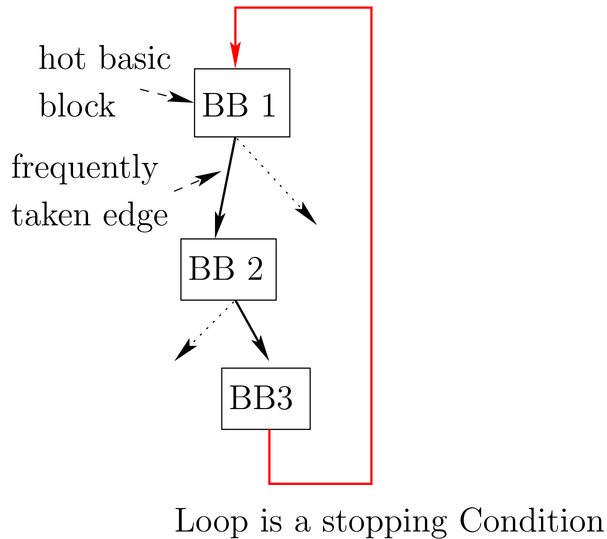


Figure 3.7: Superblock Heuristic

The heuristic considered in this thesis to form superblocks starts by first considering a hot Basic Block, that has reached *hotspot threshold*. Next Basic Blocks, that are targets of the branches that were frequently taken, are appended. The superblock formation continues in this fashion until any stopping condition is met. Figure 3.7 illustrates the superblock formation heuristic, where loop is a stopping condition.

The stopping conditions are described as follows:

- when the edge leads to a Basic Block that is already included in the current superblock, or
- when the edge leads to a Basic Block which is head of another superblock, or
- when the size of the superblock exceeds maximum, or
- when the *completion rate*, by adding the Basic Block, decreases below the *threshold completion rate*.

Although we had implemented loop unrolling and procedure inlining, we chose not to consider them. This was done primarily to keep a check on code replication. Moreover, since the benefits of loop unrolling and procedure inlining have been well studied, we instead confined ourselves with basic optimizations. Loop unrolling and procedure inlining would lead to larger regions and benefits both the baseline optimization all our HW/SW schemes.

Code Analysis and Data Flow Graph Generation

After forming the superblocks, the μ-ops are converted into Static Single Assignment (SSA) form[73]. In this transformation virtual registers are used in order to remove WAW and WAR dependencies.

While converting the code into SSA, live-in values and live-out values are kept into the appropriate architectural registers. Moreover, in the same traversal over the code, branches are converted into asserts. Basic code analysis is performed, as well, in order to identify aliasing among pairs of memory instructions. Data flow graph (DFG) is built considering both the memory and the register dependencies.

Code Optimizations

Once the dataflow graph is available, the code optimization step could be applied on the superblock. We apply various kinds of code optimizations[73], including simpler optimizations such as constant folding, copy propagation, common sub-expression elimination and dead-code elimination. Since superblocks provide a larger scope the benefits of these optimizations–which could not be enjoyed by a static compiler[4]–are exploited. A breakdown of performance benefits from various code optimizations is shown in Section 3.3.4.

Various advanced code optimizations are applied across the basic block boundaries including load hoisting, code reordering past the branches. Moreover, and depending on the context, we also apply further optimizations. For instance, in Chapter 5 we apply list scheduling and load-store telescoping [39] on the superblock.

Load-store telescoping is shown in Figure 3.8, and is applied only in Chapter 5 and 6. The add instruction shown in red, gets its input operands directly from the sub instruction. In Chapter 4 we aggressively re-order instructions.

In addition to the above-mentioned optimizations we also consider further optimizations that are contributions of this thesis. Specifically in Chapter 4 we introduce a performance tracking step that schedules the superblock after each fusion operation. This provides a metric in deciding whether or not the fused instruction will provide a performance benefit

[4] static compilers can form superblocks, but is not done in practice, primarily due to code replication

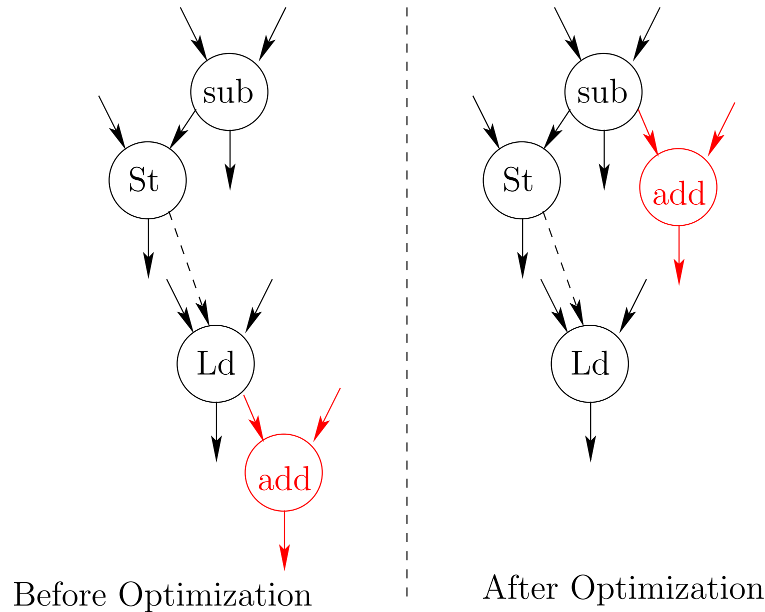


Figure 3.8: Load Store Telescoping. The critical path in the example is reduced since add can consume the data directly from the producer of store's data.

---

or not.

Furthermore, for superblocks that are loops the performance tracking step schedules several iterations of the loop. This provides a larger scope, which enables the performance tracking step to make a better judgment regarding the fused macro-op. As discussed in Chapter 4, this provides a major performance benefit in several kernel oriented benchmarks.

Scheduling

The scheduling step is applied only in Chapters 5 and 6. Since the underlying processor is in-order, instruction scheduling is an important step. Moreover, since the scheduling is performed over a superblock, which are larger than basic blocks, the effectiveness of scheduling is increased.

For the purpose of our research we implement a list scheduling technique that gives priority to the instructions in the critical path of the dependence graph [29] . The improvement in performance due to list scheduling is evaluated and shown in Section 3.3.4.

Priority is computed using dynamic programming approach using the following formula. The DFG $G = (N, E, E')$ has a node $n \epsilon N$ for each micro-op. Edges $e = (n_i, n_j) \epsilon E$ represents dependencies between micro-ops. Edges in E' represent false-dependence and

is referred as an anti-edge.

$$priority(n) = \begin{cases} latency(n) & \text{if } n \text{ is a leaf} \\ max(latency(n)+ & \\ max_{(m,n)\epsilon E}(priority(m)), & \\ max_{(m,n)\epsilon E'}(priority(m))) & \text{otherwise} \end{cases}$$

Algorithm 1 List Scheduling Algorithm

```
 1: cycle = 0
 2: ready-list = root nodes of DFG
 3: in-flight-list = empty list
 4: while ready-list or in-flight-list not empty, and an issue slot is available do
 5:    for op = all nodes in ready-list in descending priority order do
 6:       if a FU exists for op to start at cycle then
 7:          remove op from ready-list and add to in-flight-list
 8:          add op to schedule at time cycle
 9:          if op has an outgoing anti-edge then
10:             Add all target's of op's anti-edges that are ready to ready-list
11:          end if
12:       end if
13:    end for
14:    cycle = cycle + 1
15:    for op = all nodes in in-flight-list do
16:       if op finishes at time cycle then
17:          remove op from in-flight-list
18:          check nodes waiting for op in DFG and add to ready-list if all operands available
19:       end if
20:    end for
21: end while
```

Algorithm 1 lists the list-scheduling heuristic. The ready-list initially consists of μ-ops that could be issued at the first cycle, as shown in Line 2. The order among the μ-ops in the ready-list is decided by the priorities that were computed using the approach described above.

μ-ops are moved from the ready-list to the inflight-list, as shown in Line 7, only if a FU existed for it, as shown in Line 6. These μ-ops are then removed from the inflight-list when their output is ready, as shown in Line 17. Furthermore, their dependents μ-ops are inserted into the ready-list, if all of the operands of the dependent μ-ops are available, as shown in Line 18.

Register allocation

A register allocation step is performed to map virtual registers to architectural registers. The register allocation is based upon a linear scan register allocator [79].

In order to allow for the efficient execution of the code, additional scratch registers are added at the microarchitectural level that are not visible at the x86-level.

Final Code Generation

As a final step in the code generation flow, the optimized version of the code in the superblock is stored in the *Code Cache*. As mentioned earlier *Code Cache* is maintained in an unused region of memory in the application's address space. Moreover, the look-up table that keeps track of SPC to TPC mappings is updated at this step, as well.

3.3.4 Performance benefits of Code Optimizations

Above we have discussed the code generation process that we have implemented. We have also discussed various optimization techniques implemented in our optimizer. In this section we report the benefits of these code optimizations. For this purpose we chose a 4-way co-designed in-order processor and report speedup normalized to a 4-way in-order processor. The baseline in-order processor is described later in Section 3.5.2.

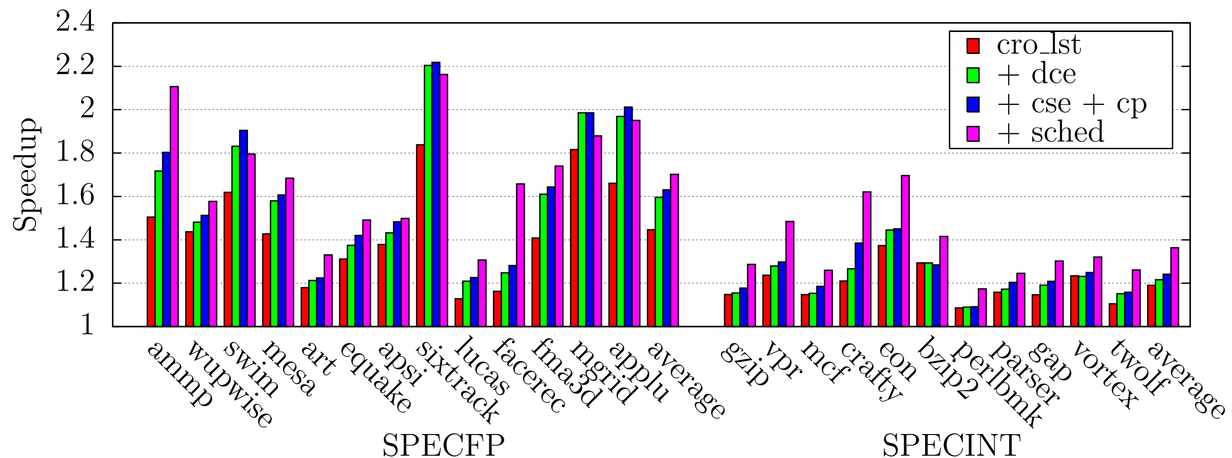


Figure 3.9: The Effect of Code Optimizations. Various code optimizations are applied. *cro_lst* stands for various memory optimizations such as load hoisting, load store telescoping and load hoisting. *dce* stands for dead code elimination. *cse* stands for common sub-expression elimination. *cp* stands for copy propagation. *sched* stands for list-scheduling.

Figure 3.9 shows the breakdown of benefits due to code optimizations and code scheduling. The plus sign (+), in the legend, indicates that the current optimizations is run with all

SPECFP		SPECINT	
ammp	26.01	gzip	11.53
wupwise	18.78	vpr	12.90
swim	61.46	mcf	9.78
mesa	21.72	crafty	16.64
art	24.56	eon	17.08
equake	14.51	bzip2	17.91
apsi	30.81	perlbmk	11.86
sixtrack	62.95	parser	9.44
lucas	10.46	gap	11.28
facerec	13.82	vortex	11.09
fma3d	29.97	twolf	11.90
mgrid	73.11	avg	12.86
applu	31.90	-	-
avg	32.31	-	-

Table 3.1: Superblock Size in μ-ops

the optimizations of above it. For instance, + **dce** indicates that this optimization was run on top of the optimization **cro_lst**. This implies that the right most bar in the Figure 3.9 corresponds to the speedup with all the optimizations.

The average speedup in SPECFP is higher than that of SPECINT. This is because superblocks are larger in SPECFP when compared to that in SPECINT, as shown in Table 3.1. Superblocks in SPECFP on an average consist of nearly 32 μ-ops, whereas in SPECINT superblocks are smaller, nearly 13 μ-ops. A larger superblock opens up opportunities for optimizations resulting in better performance.

Even at the benchmark level this fact can be verified. For instance, swim, mgrid and sixtrack have superblocks that are nearly twice as large as the average size for SPECFP. These benchmark obtain maximum speedups of nearly 100% for mgrid, 120% sixtrack, and nearly 90% for swim.

Now lets take a deeper look into the performance benefit from each of the different optimization, when applied on top of the previous one. The first optimization is Load hoisting and load-store telescoping (**cro_lst**). Together it provides 45% and 20% speedup in SPECFP and SPECINT, respectively, as shown in the Figure 3.9.

Note that the baseline is an in-order machine with an issue width of 4 which is rarely used. With the load hoisting technique there is a better usage of the issue width as this limited code reordering provides an increased ILP. However, not only effective cycles are reduced but also stall cycles are reduce with such a technique given that our processor is an stall-on-use machine. Load hoisting is performed to reduce stall cycles by placing

producer loads far from the consumers, but also hiding miss latency.

Load hoisting is particularly helpful for SPECFP codes where the region scope is large enough to provide significant reduction on the On the other hand, the use of load-store telescoping removes a significant amount of memory instructions. Most of them are spill code instructions, whose removal not only reduces the pressure on the issue width but also provides a better scope for reordering when later on list scheduling is applied. This is true for both SPECINT and SPECFP.

Traditional code optimization techniques such as dead code elimination (**dce**), along with copy propagation and common sub-expression elimination (**cse** + **cp**) provide a further performance improvement of 18% in SPECFP. For SPECINT, additional speedup of 5% is obtained. These configurations were run on top **cro_lst**.

Note that these two sets of basic optimizations have been combined with **cro_lst**. The main reason for the effectiveness of these techniques is the large scope that is not available at compile time. The superblock scope is not only larger but given the atomicity property of these superblocks the amount of dead code removal is larger. Besides, the fact that **cro_lst** removes memory instructions also provides larger opportunities for dead code removal.

List-scheduling (**sched**) is another optimization that is applied and is specially effective for SPECINT. Eon and crafty obtain 25% additional speedup due to list-scheduling. On an average, 15% additional speedup is obtained. However, since, list-scheduling is a heuristic it can degrade performance in some cases. This is especially notable in some SPECFP benchmarks such as swim, sixtrack, mgrid and applu as shown in Figure 3.9. Overall the code optimizations result in speedups of nearly 70% and nearly 37% for SPECFP and SPECINT, respectively.

3.3.5 Superblock Details

In this section we provide some statistics regarding our superblocks. We also show our sample superblocks pre and post optimizations. In our experiments, we have set the **hotspot threshold** to 100 and the **completion rate** to 90%. We have also restricted out superblocks to 200 μ-ops. The above-mentioned stopping conditions are conservative and result in small superblocks as shown in the Table 3.1.

Size of Superblocks

The average size of superblocks, in term of μ-ops, in SPECFP is 32 and in SPECINT is 13. Since SPECFP has more regular code and the branches are highly predictable, it results in larger superblocks. The larger the superblocks the more benefits it brings for

SPECFP		SPECINT	
ammp	1818K	gzip	460K
wupwise	758K	vpr	1292K
swim	209K	mcf	281K
mesa	2086K	crafty	4756K
art	366K	eon	2463K
equake	1144K	bzip2	126K
apsi	2347K	perlbmk	1605K
sixtrack	412K	parser	2797K
lucas	84K	gap	2349K
facerec	644K	vortex	3223K
mgrid	3955K	twolf	1517K
applu	81K	-	-

Table 3.2: Overhead in terms of x86 instructions

the following reason.

A larger superblocks provide a larger scope for the dynamic optimizer to play with. This results in a better optimized code, by reordering the instructions. A property that a large instruction window out-of-order processors exploits.

Superblock Translation Overhead

To measure the overhead of Super Block Translation we used estimates provided in [53, 39]. DAISY [39] report a conservative estimate of nearly 4000 source instructions to optimize a single source instruction of a superblock. They, however, mention that this a very conservative estimate and quote a reasonable estimate to be 1000 instructions. Hu et al. [53] measured the overhead of their dynamic optimizer to be around 1000 instructions.

Table 3.2 shows the overhead of Super Block Translation, in terms of x86 instructions, that we measured using the above-mentioned estimates. Since most benchmarks execute billions of x86 instructions, this overhead is less than 1%. Moreover, the longer the benchmarks run, the more the costs could be amortized, if the same superblocks are reused.

We show the overhead for the estimate using 1000 instructions. Its obvious from the table above even with a very conservative estimate of 4000 instructions the overhead is around 1%. This low translation overhead has also been observed in Dynamo [9] and by Mathew Merten et al. [71].

A Sample Superblock

Listed below are two Basic Blocks with start address 0x8066f8e and 0x8066da9 from *ammp* benchmark, which were selected for superblock formation. The instructions listed below are μ-ops translated from x86 ISA. The first Basic Block has got 9 μ-ops, whereas the second one has got 6 μ-ops.

[som] indicates start of *macro* (x86) instruction, whereas [eom] indicates end of macro. This implies that the first three μ-ops correspond to a single x86 instruction of the type read-modify-write and its size is 7 bytes. The μ-ops are committed only when all the μ-ops corresponding to the x86 instruction have successfully executed. [zco] indicates that the μ-op sets the ZAPS[5], carry and the overflow flags. The first Basic Block terminates in a conditional branch that is dependent on the values of the zf (zero flag) and the cf (carry flag).

```
1   BasicBlock 0x8066f8e
    0x8066f8e: ldd tr0 = [rbp,-232] [som] [7 bytes]
3   0x8066f8e: addd tr0 = tr0,1 [zco]
    0x8066f8e: std.+ mem = [rbp,-232],tr0 [eom] [7 bytes]
5   0x8066f95: ldd tr0 = [rbp,-328] [som] [7 bytes]
    0x8066f95: addd tr0 = tr0,32 [zco]
7   0x8066f95: std.+ mem = [rbp,-328],tr0 [eom] [7 bytes]
    0x8066f9c: ldd.- rcx = [rbp,-416] [som] [eom] [6 bytes]
9   0x8066fa2: ldd tr0 = [rbp,-232] [som] [6 bytes]
    0x8066fa2: subd.+ tr0 = tr0,rcx [zco] [eom] [6 bytes]
11  0x8066fa8: br.l.- rip = zf,of [taken 0x8066da9, seq 0x8066fae] [som] [eom
      ] [6 bytes]
    Basic Block terminates with taken rip 0x8066da9, not taken rip 0x8066fae
13  BasicBlock 0x8066da9
    0x8066da9: ldd.- rax = [rbp,-232] [som] [eom] [6 bytes]
15  0x8066daf: ldd.- rbx = [rbp,-516] [som] [eom] [6 bytes]
    0x8066db5: adda tr8 = rbx,zero,rax*4 [som] [3 bytes]
17  0x8066db5: ldd.- rcx = [tr8,0] [eom] [3 bytes]
    0x8066db8: ldd.- rbx = [rdi,1260] [som] [eom] [6 bytes]
19  0x8066dbe: andd.+ tr0 = rbx,rbx [zco] [som] [eom] [2 bytes]
    0x8066dc0: br.le.- rip = zf,of [taken 0x8066fb3, seq 0x8066dc6] [som] [
      eom] [6 bytes]
21  Basic Block terminates with taken rip 0x8066fb3, not taken rip 0x8066dc6
```

Listed below are the Basic Blocks in the Single State Assignment form. All the rnm* registers are virtual registers. Figure 3.10 shows the corresponding DataFlow Graph. The edge between μ-op 0 (ld) and μ-op 2 (st), shows that the two memory instructions completely alias. Similarly, μ-ops 3 and 6 alias completely. At this point superblock can be formed and is ready to undergo various code optimizations.

[5] Zero, Auxiliary, Parity and Sign flags

```
0x8066f8e: ldd rnm0 = [rbp,−232] [som] [7 bytes]
0x8066f95: addd rnm1 = rnm0,1
0x8066f9c: std.+ mem = [rbp,−232],rnm1 [eom] [7 bytes]
0x8066fa3: ldd rnm2 = [rbp,−328] [som] [7 bytes]
0x8066faa: addd  rnm3 = rnm2,32
0x8066fb1: std.+ mem = [rbp,−328],rnm3 [eom] [7 bytes]
0x8066fb8: ldd.− rnm5 = [rbp,−416] [som] [eom] [6 bytes]
0x8066fbe: mov rnm4 = zero,rnm1
0x8066fbe: subd.+ rnm7 = rnm1,rnm5 int:[zo] [eom] [6 bytes]
0x8066fc4: br.l.− rnm8 = rnm7,rnm7 [taken 0x8066da9, seq 0x8066fae] [som]
     [eom] [6 bytes]
0x8066fca: mov rax.t = zero,rnm1
0x8066fca: ldd.− rnm6 = [rbp,−516] [som] [eom] [6 bytes]
0x8066fd0: adda tr8 = rnm6,zero,rnm1∗4 [som] [3 bytes]
0x8066fd3: ldd.− rcx = [tr8,0] [eom] [3 bytes]
0x8066fd6: ldd.− rbx = [rdi,1260] [som] [eom] [6 bytes]
0x8066fdc: andd.+ tr0 = rbx,rbx [zco] [som] [eom] [2 bytes]
0x8066fde: br.le.− rip = tr0,tr0 [taken 0x8066fb3, seq 0x8066dc6] [som] [
    eom] [6 bytes]
```

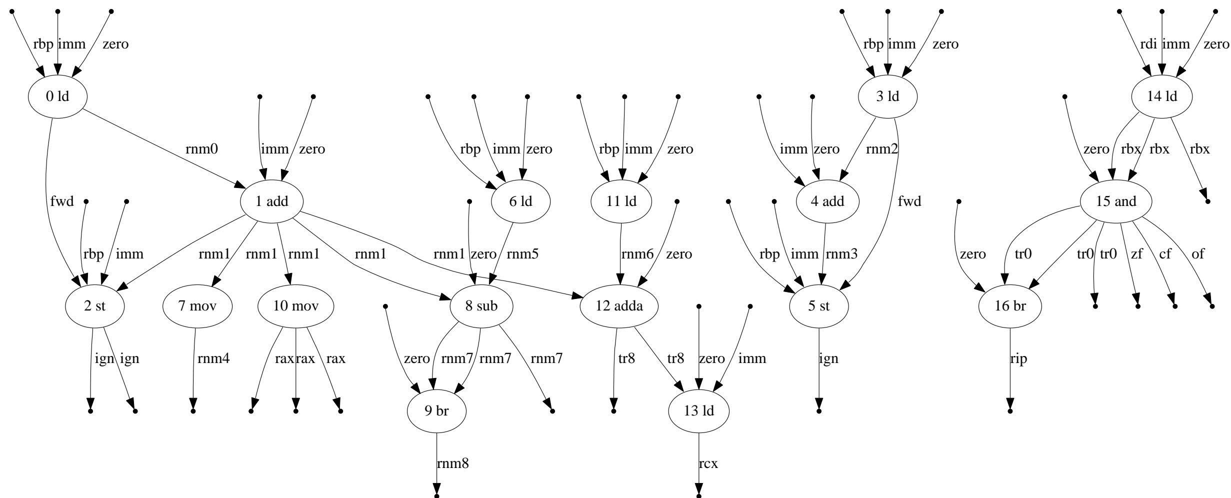


Figure 3.10: Generated Data Flow Graph

Listed below is the corresponding superblock after code optimizations. Note that the order of μ-ops have changed, because of instruction scheduling. As you can see most of the loads that depend on rbp are independent of each other and are hoisted above the stores. Moreover, μ-ops producing flags that don't have any consumers have their flags reset.

```
1  BasicBlock 0x1b4
   0x1b4: ldd rnm0 = [rbp,-232] [som] [64 bytes]
3  0x1b4: ldd rnm2 = [rbp,-328]
   0x1b4: ldd rnm6 = [rbp,-516]
5  0x1b4: ldd rnm5 = [rbp,-416]
   0x1b4: ldd rbx = [rdi,1260]
7  0x1b4: addd rnm1 = rnm0,1
   0x1b4: addd rnm3 = rnm2,32
9  0x1b4: adda tr8 = rnm6,zero,rnm1*4
   0x1b4: std mem = [rbp,-232],rnm1
11 0x1b4: std mem = [rbp,-328],rnm3
   0x1b4: andd tr0 = rbx,rbx [zco]
13 0x1b4: ldd rcx = [tr8,0]
   0x1b4: subd rnm7 = rnm1,rnm5 int:[zo]
15 0x1b4: mov rax = zero,rnm1
   0x1b4: assert rnm8 = rnm7,rnm7
17 0x1b4: br.le.- rip = zf,of [taken 0x8066fb3, seq 0x1f4] [eom] [64 bytes]
   Basic Block terminates with taken rip 0x8066fb3, not taken rip 0x1f4
19 BasicBlock 0x1f4
   0x1f4: bru.- rip = zero,zero [taken 0x8066dc6, seq 0x8066dc6] [som] [eom]
      [4 bytes]
21 Basic Block terminates with taken rip 0x8066dc6, not taken rip 0x1f8
```

The superblock is stored in a new region of memory with a virtual address 0x1b4, which is the TPC and 0x8066f8e is the corresponding SPC. Moreover, the head of the superblock is marked som, whereas the tail of the superblock is marked eom. This ensures that the μ-ops of the superblock will commit only when all other μ-ops have successfully been executed, similar to the commit of an x86 instruction.

Furthermore, there is an exit branch to fall-through path, as indicated in the second Basic Block. As you can see the contents of the two Basic Blocks are placed in a single Basic Block, which we refer to as a Superblock.

3.4 Simplifications to the Implementation

In reality the VMM time shares the processor resources along with the application and OS binaries it emulates. In order to implement this behavior one must execute the VMM code on the timing simulator. When the VMM code runs on the timing simulator, it also occupies the I-Cache and D-Cache lines. This implies the effective cache size available for the application is reduced.

In our implementation we do not model this behavior. We instead run the VMM offline and measure only the overhead due to superblock translation. The effect of reduced effective cache size was not measured. However, we do model the effect of sharing I-cache

between original x86 instructions and the translated μ-ops from the *Code Cache*.

Our primary goal was to get insights in the performance of a co-designed processor executing translated binaries in the steady state. Executing the VMM code in the timing simulator and context-switching between application and the VMM would require a significant engineering effort. Moreover, even if we had executed our VMM monitor on the simulator to estimate the overhead, it would not be close to efficient industrial implementation.

Moreover, as the PTLsim we are using is not a full system simulator, the Cd-VM that we have implemented is not a full system virtual machine. We instead model a Process Virtual Machine similar to Digital FX!32[22] and HP's Dynamo[9]. This means the OS instructions are not simulated. Our key goal was to understand the benefits of combined HW/SW optimizations on applications performance.

The implication of not implementing a fully system virtual machine is that we could model the page fault behavior precisely. An accurate implementation of Cd-VM must deal with page faults. For instance, imagine a scenario in which the translated Basic Blocks are from different source pages. As a result it could happen that the OS could have swapped one of the pages to the disk. The application when running on a normal processor would have incurred a page fault.

This implies that the Cd-VM should also take this fault by checking whether the translations are valid before executing code from a different source page. There are multiple ways to deal with page faults and are described in [88].

Self-modifying code also require combined HW/SW solution to be tackled efficiently. Some implementations use a write-protect bit in the TLB to mark the source code page translations. However, keeping the protection information at the granularity would result into frequent flush of translations for pseudo self-modifying code[6].

To deal with pseudo self-modifying code protection at finer granularity is maintained. A write-protect table holds write-protect bitmasks corresponding to smaller regions in the page. This case of pseudo self-modifying code is held using fine-grained write protect bitmask[35]. Since in this thesis we have used SPEC2000 benchmarks, none of which have self-modifying code, we chose not to implement this.

Furthermore, we assume that we have unbounded JTLB and unbounded code cache. Since the goal of the thesis was to gain insights on HW/SW optimizations on the steady state behavior[7], we chose not to implement it. Surely the size of JTLB can impact the cycle time of the processor and hence a realistic implementation will have to look-up in the software map-table.

[6] Pseudo self-modifying code are those where data in interspersed with code in the same page

[7] We made an assumption that in steady state, a small subset of superblocks can provide a good coverage. The JTLB could be large enough to hold this small subset of superblocks.

3.5 Baselines Used

In chapters 4, 5 and 6 we will propose three different kinds of co-designed processor. For instance, in Chapter 4, we will propose a co-designed out-of-order processor with a Programmable Functional Unit (PFU). This proposed co-designed processor is based upon an out-of-order processor microarchitecture that is described in Section 3.5.1.

Common Parameters	
I-Cache	16 KB, 4-way, 64 B line, 2 cycle access
Branch Predictor	Combined Predictor 64 K 16-bit history 2-level, 64 K bi-modal, 1K RAS
Fetch Width	4 μ-ops / x86 instructions up-to 16 bytes long
Issue Width	4 (2 loads, 2FP, 2 INT, 2 st)
L1 Data Cache	32 KB, 4-way, 64 B line, 2 cycles
L2 Cache	256 KB, 16-way, 64 B line, 6 cycles
L3 Cache	4 MB, 32-way, 64 B line, 14 cycles
Main Memory	154 cycles
Out-of-order Parameters	
Rename	8 source, 4 destination operands
Issue Queue	16 entry FP, Int, Mem
Functional Units	2 LDU, 2 ALU, 2 AGU, 2 FPU
Register File	128 entry INT RF, 128-entry FP RF, 4 write ports each
ROB	128 entry
LSQ	80 entry (48 loads + 32 stores)
Load Fill Request Queue	8 entry
Miss Buffer	8 entry

Table 3.3: Baseline 4-wide Out-of-Order Processor Configuration

Furthermore, in Chapter 5 we will propose a co-designed in-order processor with two general purpose accelerators. For this purpose, we use a co-designed in-order processor as a baseline, which is described in Section 3.5.2.

Furthermore, in Chapter 6 we will propose a co-designed out-of-order processor. For this purpose, we will use several out-of-order baselines, depending upon the context. The first baseline is a conventional ROB based out-of-order processor, which is described in Section 3.5.1. Moreover, this baseline processor has two variants, with and without support for hardware memory disambiguation.

Moreover, in the same chapter, we propose a new steering heuristic for FIFO based issue logic. The performance of this FIFO base issue logic is normalized to a baseline co-designed processor with a CAM based issue logic. All other features of the baseline co-designed processor are exactly same to our proposed co-designed processor.

Common Parameters	
I-Cache	16 KB, 4-way, 64 B line, 2 cycle access
Branch Predictor	Combined Predictor 64 K 16-bit history 2-level, 64 K bi-modal, 1K RAS
Fetch Width	8 μ-ops / x86 instructions up-to 16 bytes long
Issue Width	8 (4 loads, 4 FP, 4 INT, 4 st)
L1 Data Cache	32 KB, 4-way, 64 B line, 2 cycles
L2 Cache	256 KB, 16-way, 64 B line, 6 cycles
L3 Cache	4 MB, 32-way, 64 B line, 14 cycles
Main Memory	154 cycles
Out-of-order Parameters	
Rename	8 source, 8 destination operands
Issue Queue	16 entry FP, Int, Mem
Functional Units	4 LDU, 4 ALU, 4 AGU, 4 FPU
Register File	256 entry INT RF, 256-entry FP RF, 8 write ports each
ROB	256 entry
LSQ	160 entry (96 loads + 64 stores)
Load Fill Request Queue	64 entry
Miss Buffer	64 entry

Table 3.4: Baseline 8-wide Out-of-Order Processor Configuration

3.5.1 The Baseline Out-of-Order Processor

Our baseline Out-of-Order processor is based upon a state-of-art modern out-of-order processor and the details are available in Table 3.3. Moreover, in Figure 3.1 the microarchitecture of this baseline is illustrated.

ROB and physical registers are allocated at the rename stage as done in Pentium 4[51]. Frontend Register Rename Table (FRRT) and Backend Register Rename Table (BRRT) hold speculative and committed register states respectively, as shown in Figure 3.11.

Loads and stores have 3 execution pipeline stages, which are address generation, address translation and memory disambiguation. Load hit speculation is also modeled as described in [58]. Memory disambiguation is conservative and store-to-load forwarding restrictions are derived from [56]. Earlier, in Section 3.2 we provided detailed description on simulator enhancements we made the baseline out-of-order microarchitecture cycle accurate.

In Table 3.4, we also list the configuration of a 8-wide processor that we use for comparison in Chapter 4.

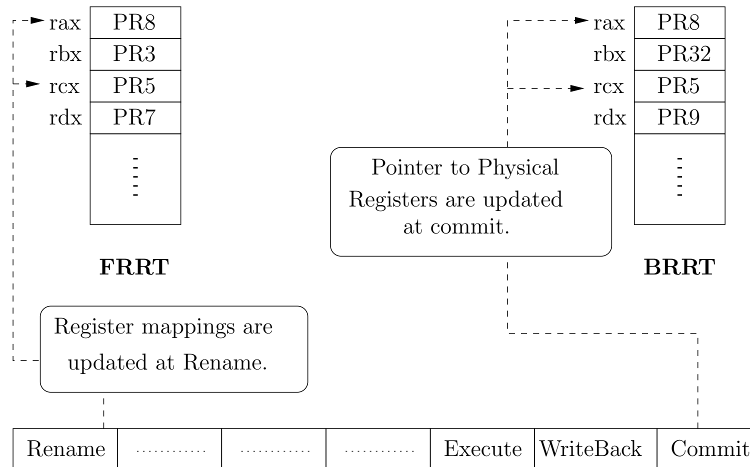


Figure 3.11: Illustration of RRT. At rename stage the FRRT is updated with new mappings, whereas the BRRT is updated at commit stage.

3.5.2 The Baseline Co-designed In-Order Processor

Figure 3.12 shows the block diagram of the proposed co-designed in-order microarchitecture. Fetch buffer, decoder buffer help in decoupling the frontend from the backend. This allows the stalls in backend to be tolerated. Table 3.5 provides detailed information of the microarchitecture of the baseline processor. The baseline microarchitecture is a two-way/four-way in-order processor.

Issue logic contains FIFO queue and counters are maintained to ensure back-to-back execution of μ-ops. Figure 3.13 shows the pipeline of the proposed processor. This baseline processor was used to support execution of fused μ-ops that are proposed in Chapter 5. The shaded stages are when a pair of fused μ-op is treated separately. Align/Fuse stage checks the first bit of each μ-op and fuses the current μ-op with a succeeding one. All the subsequent stages treat a pair of fused μ-op as a single entity.

Moreover, in order to support *bulk commit* of *atomic superblocks*, a shadow copy and a working copy of Register File is maintained. A special commit micro-op copies all the working registers into their corresponding shadow register, when the superblock execution is successful. On the other hand, in case of an exception, a rollback operation copies the shadow copy back to the working copy.

Commit and rollback is applied to stores as well; and store data is held in gated store buffer [95]. The data in gated store buffer is committed to memory when superblock successfully finishes executing all the micro-ops.

4/2-way In-order Processor Parameters	
I-Cache	16 KB, 4-way, 64 B line, 2 cycle access
Branch Predictor	Combined Predictor 64 K 16-bit history 2-level, 64 K bi-modal, 1K RAS
Fetch Width	4/2 μ-ops / x86 instructions up-to 16 bytes long
Issue Width	4/2 (2/1 LD, 2/1 FP, 2/1 INT, 2/1 ST)
L1 Data Cache	32 KB, 4-way, 64 B line, 2 cycles
L2 Cache	256 KB, 16-way, 64 B line, 6 cycles
L3 Cache	4 MB, 32-way, 64 B line, 14 cycles
Main Memory	154 cycles
Issue Queue	16 entry
Functional Units	2/1 LDU, 2/1 ALU, 2/1 AGU, 2/1 FPU
Register File	32-entry INT RF, 32-entry FP RF, 4 write ports each

Table 3.5: Baseline In-order processor configuration

3.5.3 Co-designing the baselines

A few hooks were added in order to support the execution of the Cd-VM. Hardware Profiling (block and edge) [28] identifies frequently executed code. Moreover, as described earlier, dual-mode decoders were added to run x86 instructions natively by cracking them into μ-ops. The x86 mode pipeline of the processor is shown in the Figure 3.13. Each cycle 16-byte chunk of instruction bytes is fetched from the I-cache. Instruction boundaries are determined, x86 instruction go-to first level decoder while μ-ops go directly to second-level decoder.

The primary benefit of using a dual-mode decoder is to cut down on overhead due to cold-code binary translation [53]. However, as a consequence of dual-mode decoders I-Cache holds both the x86 instructions and the μ-ops. The μ-ops are stored in a *Code Cache*, which in our implementation is an unused region of memory in the application's address space. Moreover, an unbounded JTLB is modeled that holds the SPC to TPC mappings.

Since in Chapter 4 and 6 we propose a co-designed out-of-order processor, two separate *bulk commit mechanisms* are proposed as well. This enables concurrent execution of multiple superblocks. The *bulk commit mechanism* discussed above as proposed in baseline co-designed in-order processor will not work.

For Chapter 5, the *bulk commit mechanism* consists of a *shadow copy* and a *working copy* of the Register File. *Gated store buffer* [95] holds the store data corresponding to a superblock. This *bulk commit mechanism* is exactly the one proposed by Transmeta's Crusoe [60], and was described earlier.

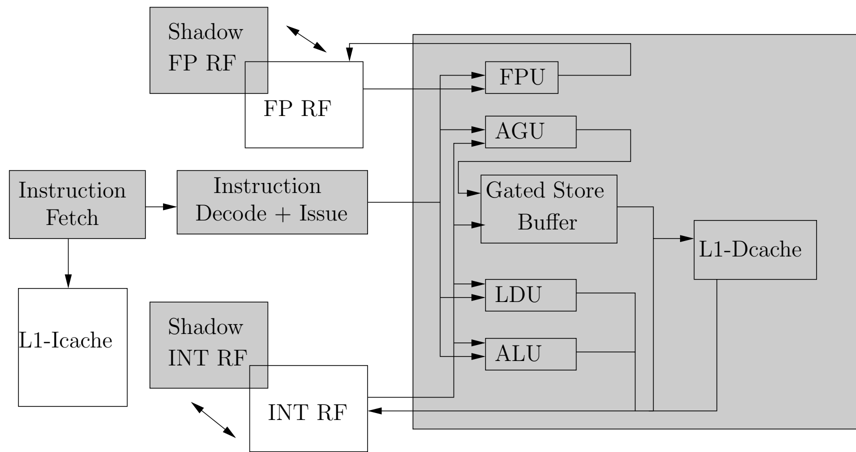
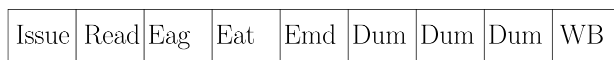


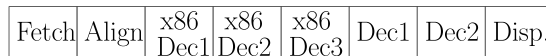
Figure 3.12: The Co-designed 2-way In-order Processor Overview. A decoupled in-order microarchitecture is used. Shadow Register Files holds the committed state, whereas the Working Register Files holds the speculative state. Stores are held in Gated store buffer until commit.

ALU Pipeline Backend



Load Pipeline Backend

FP Pipeline Backend



micro-op Mode Pipeline Frontend x86 Mode Pipeline Frontend

Figure 3.13: μ-op/macro-op pipeline. Multiple decode stages are removed from the μ-op pipeline.

Chapter 4

Co-designed Programmable Functional Unit

In this chapter, we propose a novel programmable functional unit (PFU) to accelerate general purpose application execution on a modern out-of-order x86 processor in a complexity-effective way. A Co-Designed Virtual Machine Monitor (Cd-VM) binary translates applications and generate instructions that run on the processor. Groups of frequently executed micro-operations (micro-ops) are identified and fused into a macro-op (MOP), which runs on the PFU.

Results presented in this chapter show that this HW/SW co-designed approach produces average speedups in performance of 17% in SPECFP and 10% in SPECINT, and up-to 33%, over modern out-of-order processor. However, with a slight modification in the proposed MOP model we obtain improvements in performance of 29% in SPECFP and 19% in SPECINT. Moreover, we also show that the proposed scheme not only outperforms dynamic vectorization using SIMD accelerators but also outperforms an 8-wide issue out-of-order processor.

4.1 Introduction

In this chapter, we propose a novel programmable functional unit (PFU) to accelerate general purpose application execution, in a complexity-effective way. We leverage the fact that reducing the execution latency and increasing the width of the processors leads to a performance improvement.

We use a HW/SW co-designed approach to build a out-of-order x86 processor by transparently optimizing applications, and improve performance, without increasing the width of the processor. We propose a novel split-macro-op (split-MOP) model to efficiently use

the PFU and also describe the microarchitectural changes required in order to integrate the PFU in the existing out-of-order processor.

A significant fraction of the execution cycle in out-of-order processors is dedicated to operand forward logic [41, 76]. In our PFU design we remove the operand forward logic from PFU to other FUs and dedicate this fraction of execution cycle to execution.

On the other hand, researchers [26, 78] have shown that chain of simple ALUs can be collapsed in a single cycle. CCA [26], for instance, have validated this conclusion by evaluating their design for various depth levels. Hence, based up on the above two arguments a single cycle or at-most two cycles is a reasonable estimate for the execution latency of the PFU.

In the proposed scheme, the PFU is programmed using a Cd-VM. The software layer dynamically profiles the application code, and identifies frequently executed regions. It then optimizes these regions by fusing a sequence of micro-ops into a macro-operation (MOP). This transformed code is stored in a concealed memory and is executed instead of the original code. We also propose dynamic compilation techniques required in order to achieve this.

Results presented in this chapter show that the use of a PFU provides a significant average speedup of 17% in SPECFP and 10% in SPECINT, and speedup of up-to 33% for some benchmarks, over current out-of-order processor. Moreover, we also show that the proposed scheme not only outperforms SIMD accelerators when they are dynamically managed by the Cd-VM, but also outperforms an 8-wide issue out-of-order processor.

The key contributions of this chapter are as follows:

- We propose a novel Programmable Functional Unit, along with a novel split-MOP execution model. We also discuss the microarchitectural changes required in order to incorporate the PFU in a complexity-effective manner.
- We describe an effective algorithm to dynamically fuse instructions using a Cd-VM. Our dynamic compilation scheme handles memory and loop-carried dependencies to aggressively reorder the code and generate MOPs, appropriately.
- We propose a *bulk commit mechanism* in order to commit atomic superblocks. Firstly, this *bulk commit mechanism* enables superblocks to be larger than the ROB. Secondly, it reduces the stall at frontend, that are caused due to the atomic property of the superblocks.

The rest of the chapter is organized as follows. First, In the Section 4.2, the proposed PFU is described along with its execution model and microarchitecture. Next, In the Section 4.3, we describe the problems arising due to atomicity of superblocks, which requires a

bulk commit mechanism. In the same section we propose the bulk commit mechanism in the context of out-of-order processors.

Dynamic compilation techniques are discussed in Section 4.4. A detailed evaluation and analysis of the PFU, its design points and comparison with alternate schemes is presented in Section 4.5. Finally, related work is reviewed in Section 4.6 and we conclude in Section 4.7.

4.2 The Co-designed PFU proposal

We propose a novel Programmable Functional Unit (PFU) that executes a set of fused μ-ops called a MOP. Application binary is dynamically recompiled and MOPs are generated. We propose a split-MOP execution model to execute the MOPs.

In this section we describe 1) the split-MOP execution model, 2) the PFU, and 3) the required changes in the microarchitecture to incorporate the PFU.

4.2.1 Split-Mop Execution Model

A MOP like any other instruction requires inputs and outputs to execute. However, the input and output parts of MOP are split into several μ-ops using a split-MOP model. Our split-MOP model consists of following μ-ops : (1) a set of loads to provide inputs from memory (ld-set), (2) a set of register moves to provide inputs from register file (mv-set), (3) a computation macro-op (CMOP), and (4) a store set (st-set). Figure 4.1 shows an example of split-MOP. Note, that irf0, irf1, irf2 in Figure 4.1b indicates the IRF (Internal Register File, discussed later) registers.

Read Macro-Op RMOP

The inputs to CMOP are provided by RMOP which consists of ld-set and mv-set. Ld-set as described above contains a set of loads that brings inputs from memory, whereas a mv-set brings inputs from the register file. Ld-set consists of independent loads that can be issued in parallel. The number of μ-ops in either the ld-set or the mv-set are constrained by the internal storage of the PFU, in other words the internal register file (IRF), which is discussed in details in Section 4.2.3. Note that loads in the ld-set write to both the conventional physical register file and the IRF. Mv-set, however, writes only to the IRF as shown in Figure 4.1b.

```
ld rcx = [rax, 8]
add rdx = rcx, rax
add rsp = rbx, 1
sub rbp = rsp, rdx
st [rbp, 4], rax
```

(a) μ-ops before fusion

```
ld irf0, rcx = [rax,8] : ld-set
mov irf1, irf2 = rax, rbx : mv-set
cmop rdx, rsp, rbp
st [rbp, 4], rax
```

irf* : Internal Register File Tag

(b) Macro-op after fusion

Figure 4.1: Split-MOP Model. The left side shows a normal code sequence. On the right is a Split-MOP, which consists of a *ld-set* that has a single load. Following is a *mv-set* which consists of single mov that moves rax, rbx to internal register irf1 and irf2, respectively. *cmop* follows next that has three destination operands rdx, rsp, rbp.

Computation Macro-Op (CMOP)

The ld-set and mv-set is followed by a CMOP, which contains information of the fused μ-ops. A CMOP is encoded into an opcode a unique identifier, and destination registers. Transient registers are not reflected in CMOP's destination register. CMOP does not contain any source operands, because, it reads input values from the IRF, but it writes directly to the physical register file. Hence, the number of destination registers in CMOP is constrained by the number of write ports in the physical register file.

The encoded data corresponding to each fused μ-op in a CMOP is known as a configuration. Configurations are appended to the superblock and stored in the code cache. These configurations are located using the unique identifier encoded in a CMOP.

The unique identifier is used as an index into a configuration TLB (CTLB). The CTLB is a set-associative structure which is indexed by the unique identifier. The entry in the table uses the indentifier as a tag and stores the virtual address[1] of the configuration corresponding to the CMOP.

The CTLB is maitained by the VMM, an entry is loaded into it when a CMOP is formed. A miss in CTLB must trap into the VMM, allowing VMM to load the corresponding mapping. However, in our experiments, we have assumed an unbounded CTLB.

[1] Since the line size is 32 bytes and so is the size of a configuration, we only need 27(32-5)bits.

Execution Model

The CMOP issues when all the loads in the ld-set and moves in the mv-set have issued. CMOP is executed in the PFU and a typical execution pipeline of the split-MOP of Figure 4.1 is shown in the Figure 4.2.

ld	Issue	Read	Eag	Eat	Emd	WB		
mov	Issue	Read	Mov					
cmop				Issue	Read	Execute	WB	
st					Issue	Read	Eag	Eat

Figure 4.2: Execution Pipeline. *Cmop* issued back-to-back with the ld-set and the mv-set.

The execution pipeline of the split-MOP as described in Figure 4.1 is illustrated in Figure 4.2. The four execution pipelines correspond to load, mov, CMOP and store respectively of 4.1b. A 3 cycle execution pipeline for load is implemented and is described in Chapter 3. The pipeline stages Eag, Eat and Emd stands for address generation, address translation and memory disambiguation respectively.

Moreover, back-to-back execution is ensured between RMOP (ld-set + mv-set) and CMOP. CMOP dependence with the ld-set and mv-set is built at runtime. This is achieved using a hierarchical issue queue model as described in Section 4.2.3.

However, in this model, if a live-in value provided by a load or a move is delayed for some reason (e.g. load miss), then the CMOP execution is also delayed. This implies that if CMOP fuses independent μ-ops, then delay of one μ-op causes the other independent μ-ops to be delayed as well. This can impact the critical path of an application.

4.2.2 Alternate Split-Mop Execution Model

We have also proposed an alternate split-mop model [32], where we get rid of the mv-set. This implies the CMOP now contains both the destination and the source operands. This alternate model constrains the size of the CMOP and number of its input and destination operands further. Moreover, as will be shown in the experiments further, we found out that the Register File Ports requirement for a CMOP is satisfied with current designs. As a result a CMOP can directly read from the Physical Register File. However, we still require ld-set to bring data from D-cache.

Furthermore, getting rid of mv-sets adds to performance benefits as back-to-back execution is not compromised. Moreover, the MOP fusion heuristic ends up forming more CMOP, because the objective function of the fusion heuristic sees benefit in forming mv-set free MOPs. This will be clearer in Section 4.4.2.

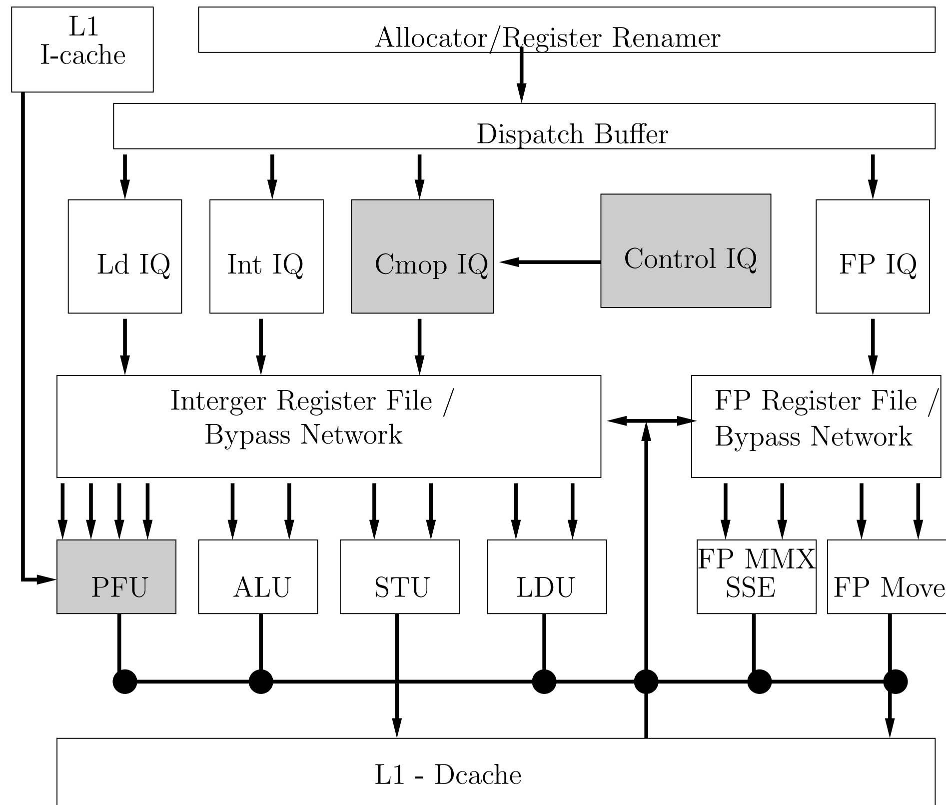


Figure 4.3: Modified Microarchitectural Block Diagram. The gray blocks are added, which consists of a *control IQ*, *Cmop IQ*, *PFU*. *Control IQ* enables issue of split-mop by sending signal to cmop which is stored in *Cmop IQ*. The cmop is then executed in the *PFU*.

4.2.3 PFU Microarchitecture

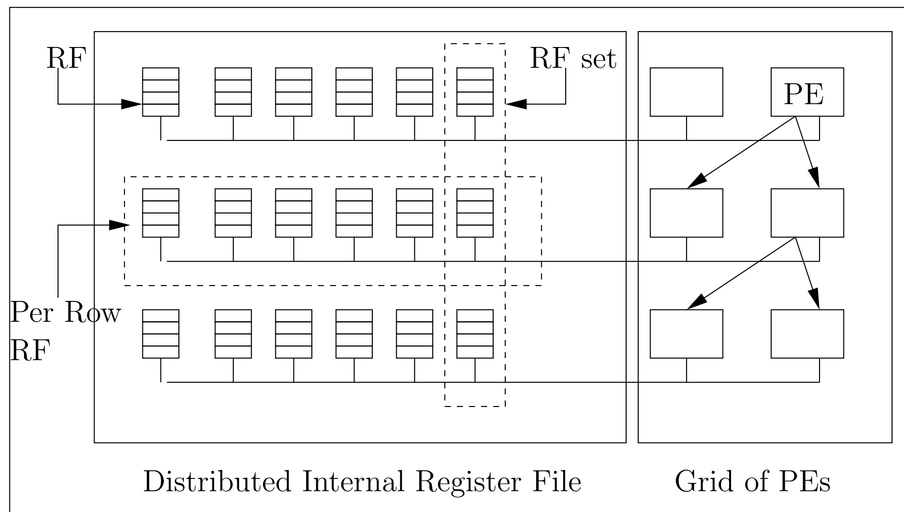
The microarchitecture that supports split-MOP execution is discussed in this section. Figure 4.3 shows a block-level microarchitecture diagram, and the added components to incorporate the PFU are shown in gray[2]. The added components are a PFU, a CMOP issue queue and a control issue queue.

The key features of our microarchitecture consist of : (1) the programmable functional unit, (2) the distributed internal register file (IRF), (3) hierarchical issue queue model, and, (4) pipeline stage modifications. Each of these are described in detail in the following subsections.

[2]For the sake of simplicity, in this figure we do not show the structures required to support VMM.

Programmable Functional Unit

We propose a PFU which has two major components: 1) Distributed Internal Register File (IRF), and (2) a grid of Processing Elements (PE). Data flows from one row to the following in the grid of PEs as shown in Figure 4.4, an organization similar to [26]. Note, that there are no latches between the PEs of two different rows. The effects of varying the grid size and PFU execution latency is studied and discussed in Section 4.5. Based upon the results we propose a grid of 2 columns and 3 rows. The inputs required by each μ-op in the grid of PEs is provided by the IRF.



Programmable Functional Unit

Figure 4.4: Programmable Functional Unit. A distributed Internal Register File is shown, which consists of multiple RFs per row that satisfies the input needs of the FUs of that row. A RF set is a column of RFs which consists of replicated data corresponding to a cmop. Our proposed PFU consists of three rows of Internal Register Files, each row consits of five register file with a capacity of four entries and four read ports and zero write ports on each. Grid of PEs consists of FUs connected in dataflow fashion. They are optimized to execute with shorter latency, due to the absence of latches and forwarding path.

Distributed Internal Register File

The proposed PFU with six PEs (2 columns, 3 rows) requires up to twelve read ports to execute all the μ-ops of CMOP simultaneously. Providing so many read ports to the physical register file is certainly not complexity-effective. Hence, in order to deal with this, we propose a separate register file, the internal register file (IRF), that is contained inside the PFU.

The inputs of multiple CMOPs are brought into the IRF from the L1 D-cache and from the physical register file. For successful execution, the IRF should have sufficient capacity, in order to hold the inputs of multiple CMOPs. A lack of this capacity can significantly handicap out-of-order execution, by causing unnecessary stalls.

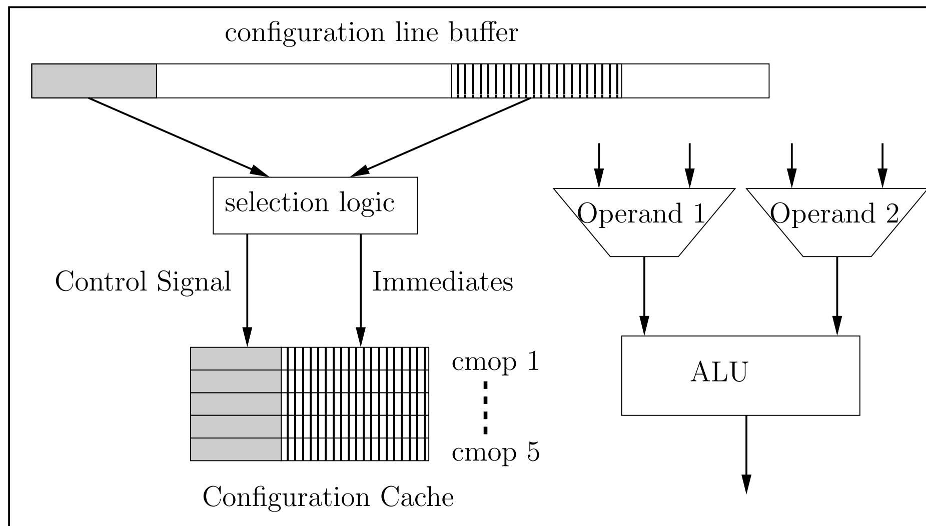


Figure 4.5: Processing Element. PE consists of an ALU and a configuration cache. Configuration cache has the capacity to hold configurations of upto 5 μ-ops, corresponding to their *cmops*, which was placed in this PE.

Moreover, this internal register file is distributed in order to provide sufficient bandwidth, as shown in Figure 4.4. IRF contains multiple register file sets, each of which is allocated to a MOP in the dispatch stage. A register file set contains replicated copies of register file, one copy corresponding to each row. Each register file has 4 entries and has 4 read ports and 4 write ports. Recollect that the CMOP writes to the conventional physical register file directly. Hence, the write ports on IRF are used by the ld-set and mv-set only, to write the inputs of the CMOP.

There are 5 different register file sets, so inputs for 5 different MOPs can be stored at the same time. Hence, the total size of this IRF is 60 (number of entries per RF*number of rows*number of RF sets = 4*3*5). Dispatch stalls in case a register file set cannot be allocated to the MOP. It is obvious from such a distributed organization that a PE could access only the register file of the row that it belongs to and to that of the MOP that is currently being executed. Note, that total number of entries both in the ld-set and the mv-set are constrained by the size of one register file. The values in the IRF are discarded only when CMOP is successfully executed.

PE and Configuration Cache

Figure 4.5 provides a deeper look into the PE. Each PE contains 1) an ALU, which is connected to the ALUs of following rows, and 2) a configuration cache. Configuration cache holds configuration of 5 CMOPs in a distributed manner. The configuration contains pre-decoded control signals of all the fused μ-ops pertaining to the CMOP as shown in the Figure 4.6.

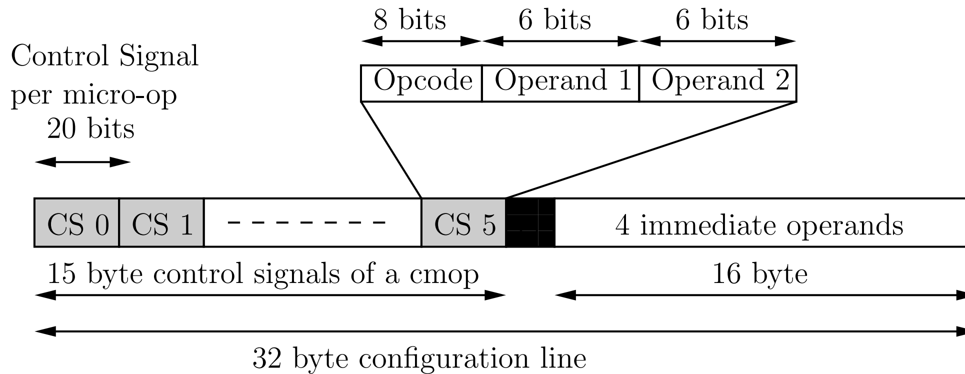


Figure 4.6: Configuration Line. A 32 byte configuration line is divided into 15 byte control signals and 16 byte immediate operands. Upto six control signals are shown, each control signal is further divided into opcode and operand bits.

Configurations are 32 bytes long and is equal to half of L1 I-cache line size. The lower 16 bytes contains opcode and source operand information of all the fused μ-ops, and the upper 16 bytes holds the immediate operand values. Opcode needs 1 byte, while the source operands can be represented with 6 bits. A total of 20 bits are required as control signal for each μ-op. Hence, six μ-ops (corresponding to six PE) can be represented with 15 bytes. Four immediate operands require 16 bytes.

We assume any combination of simple ALU μ-ops for fusion. But some μ-ops, such as shift, rotate etc. are not allowed. Neither are loads or stores allowed.

A direct access is made to the L1 I-cache to read in the configuration line corresponding to a CMOP. The configuration line is then distributed to all the PEs. Each PE contains also a line buffer to store the configuration line. The PE then selects the appropriate μ-op control signal and immediate operand, if any, from the configuration line buffer. The control signals are stored in the configuration cache. If the configuration is present in the configuration cache then it can be loaded in a cycle. The size of distributed cache is estimated to be 210 bytes.

The configuration line in the line buffer is discarded in the following cycle. We propose a single cycle selection logic. Hence, 3 cycles are sufficient to ensure that the configuration corresponding to a CMOP is properly distributed. A request for loading the configuration

is, hence, sent in the rename stage. CMOP issue stalls if the configuration is not yet loaded. Due to the spatial locality of instructions we don't discard the configuration. The PFU can hold up to 5 different configurations at any time. The configurations are managed using a simple LRU scheme.

Bypass Network

To support back-to-back execution, all the 6 PEs should receive source operands from the bypass. The PEs, however, receive inputs only from the 2 load units (LDUs) and 2 ALUs. In the evaluation Section 4.5, we however show that not all the 6 PEs need the source operands to be bypassed. For a 2x3 grid a bypass network to 4 PEs is more than sufficient.

On the other hand, a significant fraction of execution cycle of an ALU in a modern out-of-order processor is consumed by the destination operand forwarding [41, 76]. Hence, in order to support a PFU that collapses three ALUs and execute with low latency, we remove the forwarding logic from PFU to other ALUs, and dedicate this fraction of execution cycle completely to execution. Our studies indicate that such a constraint has negligible impact on performance.

Pipeline Stages

Now we discuss the microarchitectural changes required in the pipeline in order to execute our split-MOP model. We focus mainly on the pipeline stages that are impacted the most.

- **Rename** The width of a typical out-of-order processor determines the number of μ-ops that could be renamed. For instance, a 4-wide machine could rename up to four μ-ops per cycle. However, in MOP model we constrain renaming to the number of registers and not to the μ-ops. So, if a CMOP has four destination registers then only the CMOP is renamed in that cycle. However, if a CMOP requires two destination registers, two other μ-ops can be renamed in the same cycle.

- **Dispatch** Loads of the ld-set go to the traditional Issue queue, and an entry in the control issue queue is allocated for each ld-set. The control issue queue entry contains issue queue tags of all the loads in the ld-set. The same holds true for all the moves in the mv-set. This hierarchical issue queue model is described below, and illustrated in Figure 4.7.

- **Commit** Similar to the rename stage, if a CMOP has four destination registers then only the CMOP is committed in that cycle. However, if a CMOP requires two destination registers, two other μ-ops can also be committed in the same cycle.

In Figure 4.7 Ld1 is the tag associated with the load in the load set. Control issue queue entry corresponding to tag Lds depends upon ld1 issue queue entry, as shown by a backward arrow in Figure 4.7. Similarly, CMOP depends upon the Lds and Mvs control issue queue entry tags. Ld1 issue queue tag is broadcast to the control issue queue and Lds issue queue tag is broadcasts to CMOP issue queue, where CMOPs are held. CMOP's dependence with ld-set and mv-set entry is built at runtime using information encoded in the CMOPs. Such a model ensures that CMOP issues only when both the ld-set and the mv-set have issued, without having the need of explicit source operand encoding in a CMOP.

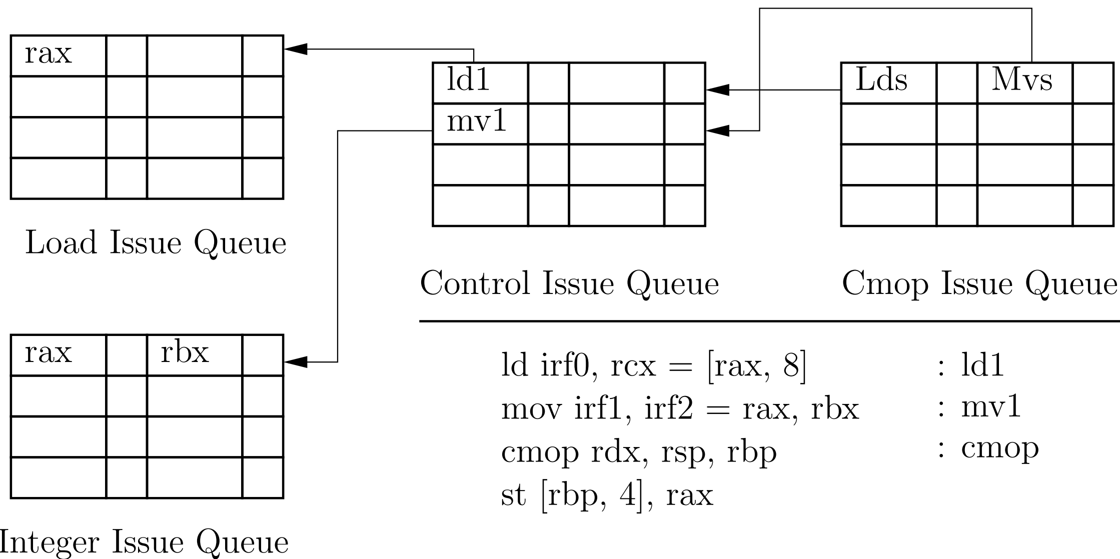


Figure 4.7: Hierarchical Issue Queue Model. As shown in the figure Control IQ contains entries corresponding to ld1 and mv1. When all the loads in ld-set, only ld1 in this case, is ready signal is sent to Cmop Issue Queue indicating Lds is ready. Similarly, when all the moves in mv-set, only mv1 in this case, is ready signal is sent to Cmop Issue Queue indicating Mvs is ready.

4.3 The Co-designed Out-of-Order Processor

Since we use a co-designed processor, it requires HW support to efficiently implement the source ISA using a VMM. These HW features include JTLB, dual-mode x86 address decoders among others. All of these features are described in Chapter 3 and are used as a baseline co-designed processor configuration. Some of these features are required irrespective of whether a processor is in-order or out-of-order.

Moreover, since in this chapter we propose a HW/SW co-designed **out-of-order** processor, we propose a novel Bulk Commit Mechanism. The baseline out-of-order processor

that we use, in this chapter, is described in Section 3.5.1.

In this section first we describe the problems associated with bulk commit of atomic superblocks in the context of out-of-order processor. Next we propose a solution to support bulk commit of atomic superblocks.

4.3.1 Bulk Commit of Atomic Superblocks

The atomic property of the superblocks requires that all the instructions of the superblocks be successfully executed before the superblock could be committed. This implies all the instructions corresponding to the superblock reside in the ROB; waiting for the last instruction[3] to have executed.

However, such a requirement leads to poor performance. Next we will describe couple of problems associated with bulk commit of atomic superblocks.

Bulk Commit Problem 1

In a conventional modern out-of-order processor a ROB is present in the back end to ensure in-order commit of μ-ops. As mentioned above the atomic property of superblocks requires that all the μ-ops corresponding to the superblock are successfully executed before the superblock could be committed. One implication of such a bulk commit in the context of out-of-order processor is that the size of the superblocks are restricted to that of the ROB.

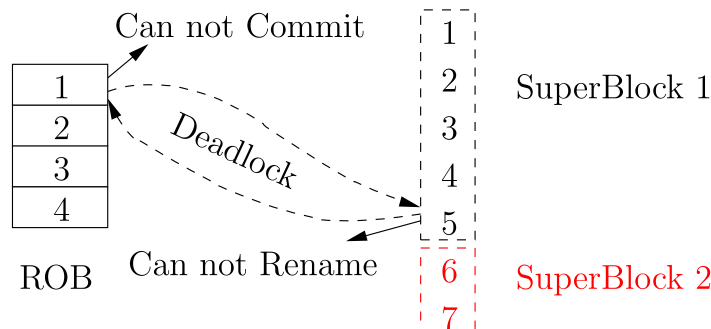


Figure 4.8: Bulk Commit Limits the Size of Atomic Superblocks.

Figure 4.8 illustrates this using a simple example, where the size of superblock is five, whereas the size of the ROB is four. As shown in the figure, suppose the first four μ-ops of the superblock have a ROB entry allocated. μ-Op 5 cannot be renamed because ROB

[3] in the execution order

does not have any entry available. Furthermore, μ-op 1 cannot commit because μ-op 5 has not yet been renamed. As a result, a cyclic dependency is created between μ-op 1 and μ-op 5, which leads to a deadlock.

The simplest solution to tackle this problem is to limit the size of the superblock to that of the ROB. This way there would not be any deadlock created. However, smaller superblock handicaps the dynamic binary optimizer's ability to apply effective code optimizations. For instance, code scheduling when applied on a larger superblock is more effective as a larger instruction window is provided to the dynamic binary optimizer.

Bulk Commit Problem 2

The second problem with bulk commit of atomic superblocks in the context of out-of-order processors is related to resource related stalls at the frontend. Figure 4.9 illustrates the problem with a simple example. As shown in the figure, suppose all the μ-ops of the superblock have a ROB entry allocated. Next suppose that μ-op 3 misses in L2-cache, as a result μ-op 1 cannot commit because it has to wait for μ-op 3 and other μ-ops in the superblock to complete execution.

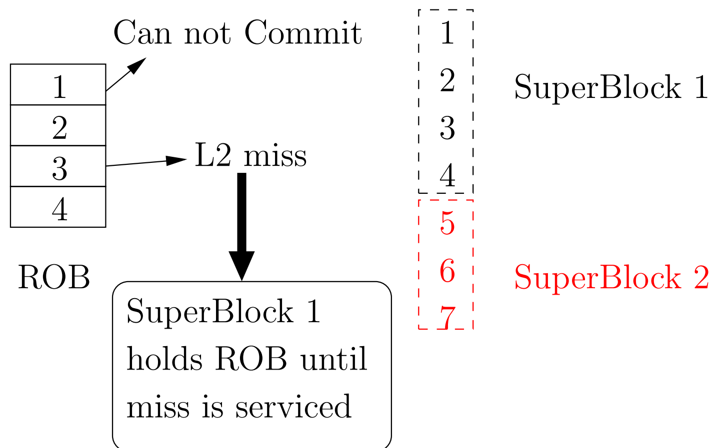


Figure 4.9: Bulk Commit leads to frontend stalls. Since μ-op 3 has not yet missed in L2, all the μ-ops of the superblock wait in the ROB.

This implies that the superblock 1 holds the ROB until the L2-cache miss is serviced. As a result the superblock 2 cannot execute, and hence concurrent execution of superblocks is impacted.

One way to limit the impact of such a scenario could be to increase the size of the ROB. However, increasing the size of hardware structures affects the cycle time and dynamic power consumption of the processor. Moreover, even though scaling the structures lead

to performance improvement, past studies have shown that such increase provides diminishing returns.

4.3.2 Bulk Commit using a Speculative Map Table

In order to tackle the two above-mentioned problems, we speculatively retire the μ-ops from the ROB. The μ-ops are retired in program order from the head of the ROB, and the state associated is updated in a speculative structure.

We introduce the Speculative Register Rename Table (SpecRRT) that holds this speculatively retired register state. At the cycle when the tail of the superblock retires the contents of the SpecRRT is committed to the Backend Register Rename Table (BRRT).

Moreover, only the μ-ops that produce live-out updates the SpecRRT. Such μ-ops are marked by the VMM during the code generation process. As a matter of fact, it is not necessary that such μ-ops be marked by the VMM. This is because when the μ-ops retire they simply overwrite the previous mappings in the SpecRRT.

For instance, suppose that a superblock contains two μ-ops that write to **rax**. Since the μ-ops are retired in the program order from the ROB, the youngest μ-op that writes to **rax** will always update the mapping in SpecRRT after the older one. This way at the end when the tail retires the SpecRRT will hold mappings of only the live-outs.

However, by explicitly marking the live-outs we save unnecessary accesses to SpecRRT when μ-ops that do not produce live-outs retire. As a result, we co-design the commit in order to save dynamic activity in the processor. This implies the μ-op encoding should be extended to incorporate a single bit indicating whether the μ-op produces a live-out or not.

The SpecRRT contains register mappings of only a single superblock at any given point in time. This is simply because the μ-ops are retired from the ROB in program order and when the tail of the superblock commits the contents of the SpecRRT are committed to the BRRT.

Figure 4.10 illustrates the bulk commit mechanism using the SpecRRT. As shown in the figure, all μ-ops update the FRRT at the rename stage. Now let's suppose *rax* and *rcx* are the live-outs of a superblock. When the two live-out producing μ-ops that write to *rax* and *rcx* commit[4] they update the SpecRRT, as shown in the Figure. At the cycle when the tail of the superblock commits, the contents of the SpecRRT is copied to the BRRT.

However, our bulk commit requires ROB in the backend to ensure atomic update of the program state. In Chapter 6 we will provide a ROB-free bulk commit mechanism using

[4] when they reach the head of the ROB

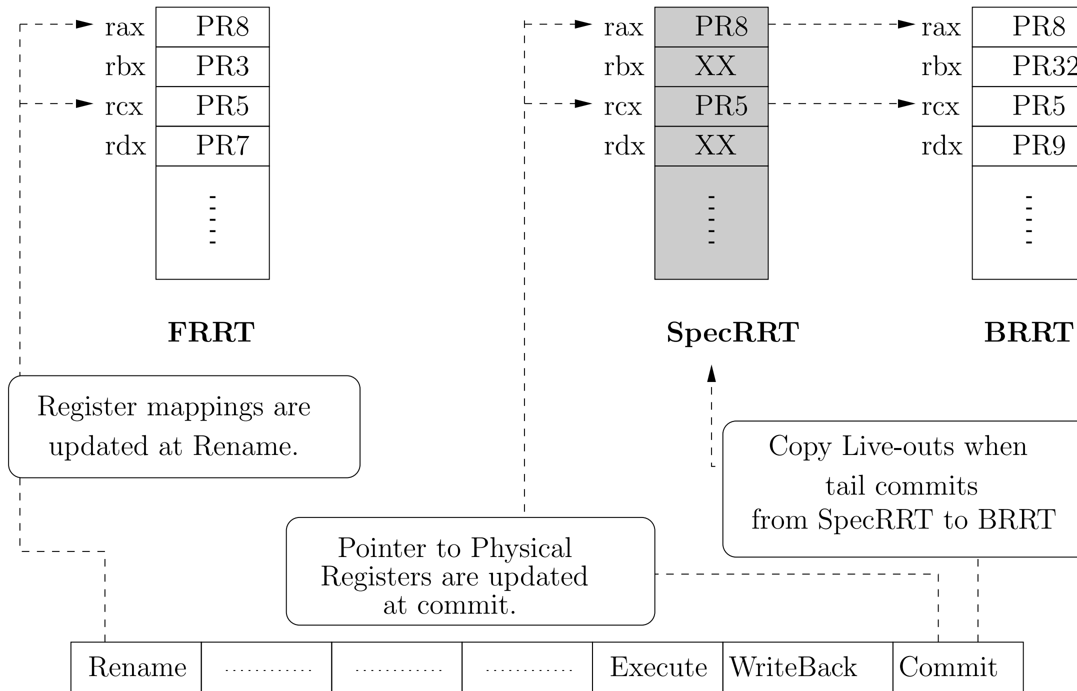


Figure 4.10: Bulk Commit with SpecRRT. rax and rcx are the live-outs of the current superblocks. Hence, rbx and rdx entries in the SpecRRT are invalid. As a result at commit only rax and rcx are updated in the BRRT.

multiple per superblock map tables.

4.4 Code Generation

The Virtual Machine Monitor (VMM) plays an important role in dynamically compiling code for an efficient use of the Programmable Functional Unit (PFU). The optimization process implemented by the proposed Cd-VM is shown in Figure 4.11. This optimization process is based on some of the basic steps described in Chapter 3, such as superblock formation, Dataflow Graph Generation, Code Optimizations and Register Allocation.

We introduce some steps which are specific to this chapter. First, we propose a pre-scheduling step that aggressively re-orders a superblock. This aggressive code re-ordering enables better fusion opportunities.

Following, pre-scheduling is a MOP Fusion step that selects μ-ops for fusion. Once a MOP is formed a performance objective function is invoked that determines whether the formed MOP shortens the latency of the superblock. In case, the formed MOP decreases the performance the MOP is discarded. All these additional steps are described in greater

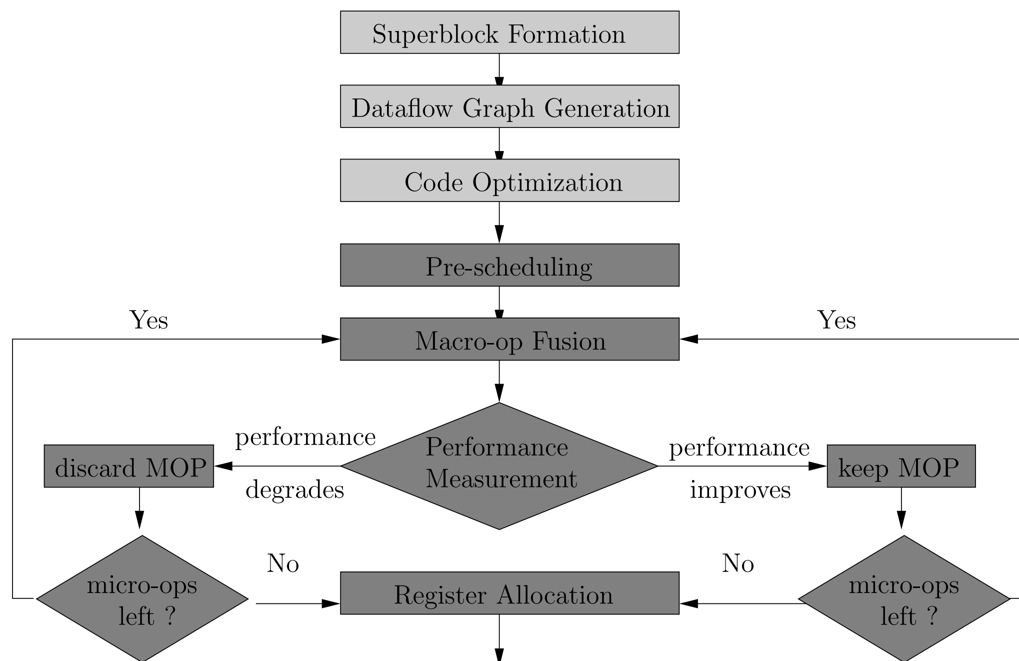


Figure 4.11: Code Generation Flow Chart. The first four are standard superblock formation steps. The Macro-Op fusion step as shown is an iterative step, where instructions are fused and then scheduled, which acts as a performance objective function. The performance objective function indicates whether the fusion is beneficial. If the fusion increases performance the MOP is kept, discarded otherwise. The fusion algorithm goes on iteratively until all the μ-ops have been considered.

details below.

4.4.1 Pre-Scheduling

Instruction pre-scheduling is performed prior to the generation of complex macro-ops. The pre-scheduling heuristic is based upon list-scheduling, but assumes unbounded execution resources. Moreover, the priority of nodes is based on the original program order, whereas in list-scheduling heuristic priorities are computed using critical path analysis.

Algorithm 2 lists the pre-scheduling heuristic. This heuristic is very similar to the list scheduling heuristic, Algorithm 1, listed earlier in Chapter 3. The ready-list initially consits of μ-ops that could be issued at the first cycle, as shown in Line 2. The order among the μ-ops in the ready-list is decided by the original program order. μ-ops are moved from the ready-list to the inflight-list, as shown in Line 6. These μ-ops are then removed from the inflight-list when their output is ready, as shown in Line 15. Furthermore, their dependents μ-ops are inserted into the ready-list, if all of the operands of the dependent

Algorithm 2 Pre-Scheduling Algorithm

```
1: cycle = 0
2: ready-list = root nodes of DFG
3: in-flight-list = empty list
4: while ready-list or in-flight-list not empty do
5:   for op = all nodes in ready-list in original program order do
6:     remove op from ready-list and add to in-flight-list
7:     add op to schedule at time cycle
8:     if op has an outgoing anti-edge then
9:       Add all target's of op's anti-edges that are ready to ready-list
10:    end if
11:  end for
12:  cycle = cycle + 1
13:  for op = all nodes in in-flight-list do
14:    if op finishes at time cycle then
15:      remove op from in-flight-list
16:      check nodes waiting for op in DFG and add to ready-list if all operands available
17:    end if
18:  end for
19: end while
```

μ-ops are available, as shown in Line 16.

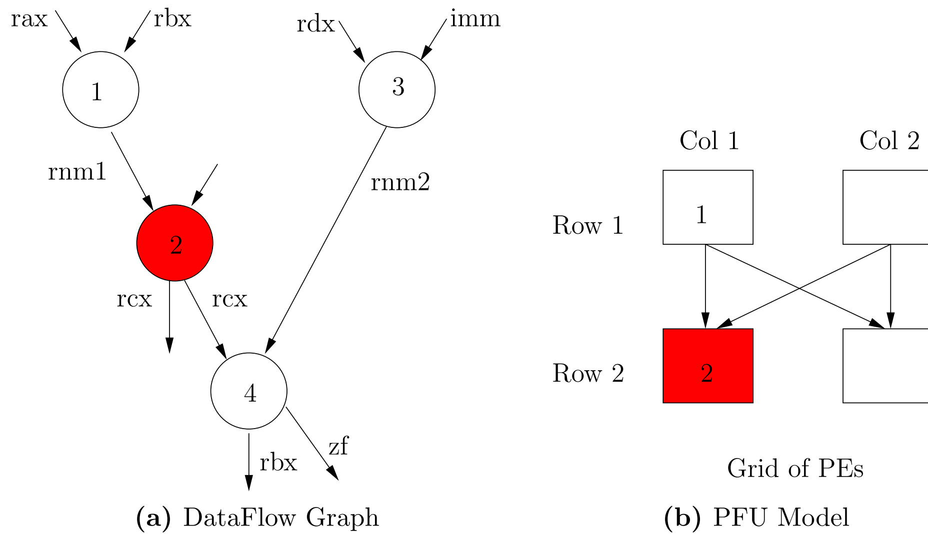
This pre-scheduling step helps in aggressively reordering μ-ops, including load and stores. As a result of which, the μ-op fusion algorithm get more opportunities to find instruction groups for fusion.

4.4.2 Macro-op Fusion

After the μ-ops have been pre-scheduled, the Macro-op fusion heuristic is invoked by the VMM. The heuristic uses a model of PFU defined by the microarchitecture. This model includes information regarding the number of rows and columns in the grid, number of input and output operands. Algorithm 3 lists the heuristic.

First, the dataflow graph is traversed in the pre-scheduled program, as shown in Line 1. A μ-op is placed such that it is placed always in the rows below the ones in which its predecessors are placed in the PFU model, as shown in Line 7.

For instance, as shown in the Figure 4.12 μ-op 2 is placed in row 2 even though row 1's col 2 is unoccupied. If the μ-op could not be fused in the current MOP it is skipped, as shown in Line 8, and will be considered in subsequent MOPs. If, however, the μ-op could



(a) DataFlow Graph **(b)** PFU Model

Figure 4.12: Macro-op Fusion Illustration. The figure in the left shows a dataflow graph and the figure on the right is a model of PFU. The fusion algorithm uses this model to place μ-ops. Data dependencies are respected by placing dependent μ-ops in a row lower than the producer's row. For instance, the μ-op 2 is placed in row 2 even though row 1's col 2 is unoccupied.

be placed, then its added to MOP, as shown in Line 11.

If a μ-op cannot be fused then its skipped as well, as shown in Line 4. Following is a list of μ-ops that can not be fused.

- Loads that are dependent on any of the μ-op of the current MOP are not included.
- Also complex μ-op (e.g. FPDIV) not considered for fusion.
- Dependent μ-ops of not included μ-ops are discarded as well.
- The algorithm considers structural constraints that prevent μ-ops for not being included in a MOP, including: (1) read ports, (2) write ports, (3) the number of rows and columns in the grid, (4) IRF register file size, and (5) μ-ops that require inputs from IRF to be placed in ALU that is connected to IRF and the bypass network.

Performance Objective Function

Note that, a CMOP can only be issued when both the ld-set and mv-set have issued, which creates artificial dependencies and could delay the critical path. In order to avoid

Algorithm 3 Macro-op Fusion Algorithm

```
1: while pre-scheduled-list not empty do
2:   start-op = first node in pre-scheduled-list
3:   for op = all nodes in pre-scheduled-list younger than start-op do
4:     if non-fusible op then
5:       continue
6:     end if
7:     place op in any row below from its producer
8:     if no such row found or not enough input output registers available then
9:       continue
10:    end if
11:    add op to MOP
12:    update input-register list and output-register list
13:  end for
14:  if MOP formed then
15:    call performance objective function
16:    if performance degrades then
17:      Discard MOP
18:      Remove start-op from pre-scheduled-list
19:      continue
20:    end if
21:    Place MOP in the superblock and continue
22:    Place all the ops of the MOP in included-list
23:    Remove all the ops of the MOP from pre-scheduled-list
24:  end if
25: end while
```

situations where a sub-optimal fusion degrades performance, the algorithm for fusion, as listed in Algorithm 3, uses a performance objective function.

Once a MOP is formed, the performance of the actual state of the generation is estimated. The performance function models the execution of the partial code generated by means of a scheduling step.

In particular, every time a MOP is formed, the MOP and the remaining code is scheduled. If the current MOP degrades performance, the MOP is discarded. The algorithm then reiterates over the remaining μ-ops and exits when all the μ-ops have been considered.

The performance objective function is illustrated above in Figure 4.14, corresponding to the dataflow graph of Figure 4.13. In this example the critical path is increased after the fusion by a single cycle. As a result the MOP will be discarded.

For those superblocks that are loops the performance objective function schedules schedules multiple iterations of the superblock. From our studies we have observed that schedul-

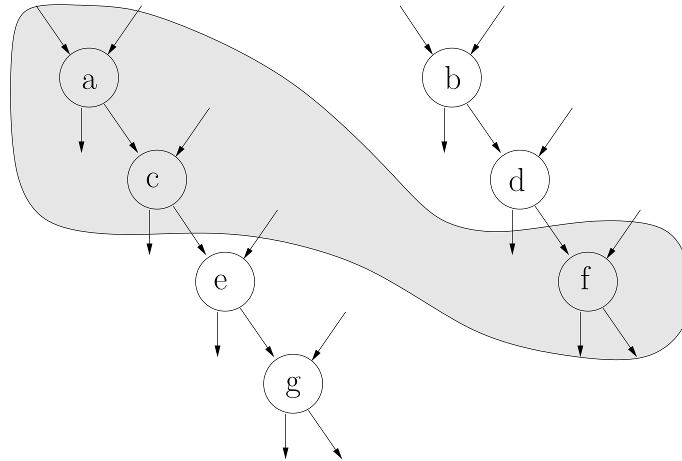


Figure 4.13: Dataflow graph showing the fused Macro-op.

ing 4 iterations is sufficient. Note that, when scheduling multiple iterations, loop-carried dependencies are considered.

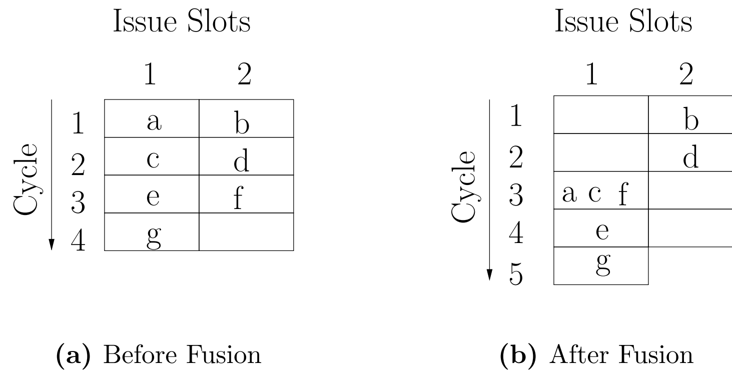
The key reason behind scheduling multiple iterations of a superblock that is a loop, is that it provides a larger scope to estimate the performance of the MOP. Instead, if the superblocks that are loops, were unrolled at the first place while forming them, then such a scheduling of multiple iterations of the superblock would not be required. However, loop unrolling not only leads to code expansion, but for atomic superblocks they cut down the **completion rate** as well.

The scheduling step of the performance function models the execution of the code in the hardware underneath by considering (1) the issue queues; (2) the issue constraints; and (3) the impact of the memory disambiguation scheme implemented in hardware. Moreover, it considers the cases where a superblock is loop by scheduling multiple iterations (i.e. 4) of the superblock.

By combining all these heuristics, we have a simple yet powerful scheme to estimate the benefits of fusion. This allows us to discard a MOP that could degrade the performance. However, since our performance objective function is a heuristic, there are scenarios where some MOPs could be discarded incorrectly.

4.4.3 Final Code Generation

As a final step in the code generation flow, the optimized version of the code in the superblock is stored in the code cache for later use. The configurations of the CMOPs corresponding to a superblock are appended to the superblock and stored in the code cache.



(a) Before Fusion

(b) After Fusion

Figure 4.14: Schedule before and after the fusion.

The configuration of a CMOP, which contains the pre-decoded control signals and the immediate values of the MOP, is associated to a unique identifier. Identifiers are encoded in 10 bits allowing up to 1024 CMOPs to be present in the code cache, which is more than sufficient based on our experiments.

A counter stored as data of the Cd-VM is increased each time a new CMOP is generated. When this counter overflows, identifiers are reused. Once an identifier is reused the superblock associated to the CMOP is discarded.

4.5 Performance Evaluation

Table 4.1 provides detailed information of the microarchitecture of the simulated processor and the proposed PFU configuration. The details of baseline out-of-order processor can be found in Chapter 3 in Table 3.3.

In this evaluation we have studied the performance of the proposed PFU using different PFU configurations: varying the grid size, and constraining the number of PEs that could receive data from the bypass network. Next, we have studied the impact of constraining the number of write ports in the Physical Register File. This constrains the number of destination registers in a CMOP. We have also studied the impact of fusion heuristics on performance.

Finally, we compare our proposal to a processor with SIMD FU, where SIMD instructions are generated dynamically. We also show that its possible to outperform a wider machine using our proposal. For this study we compare with two different variants of eight-wide issue processor.

Common Parameters	
Issue Width	4 (2 loads, 2FP, 2 INT, 2 st, **1 CMOP**)
Out-of-order Parameters	
Rename	8 source, 4 destination operands
Issue Queue	16 entry FP, Int, Mem, **CMOP and Control**
Functional Units	2 LDU, 2 ALU, 2 AGU, 2 FPU, **1 PFU**
Rename Table	1 FRRT, 1 BRRT, **1 SpecRRT**
PFU Parameters	
Grid size	**2 columns, 3 rows**
Internal Register File	**5 RF sets, 3 RFs per set,** **4 entries per RF,** **4 read and 4 write ports per RF**
Configuration Cache	**5 (7 byte) entries per PE**
Execution Latency	**1 or 2 cycles**

Table 4.1: Proposed Processor Configuration. The items in bold are added over the baseline out-of-order processor of Table 3.3.

4.5.1 Impact of Microarchitectural Constraints

As mentioned earlier in Section 4.4, microarchitectural constraints are considered while fusing micro-ops. These constraints are PFU grid size, number of write ports in conventional register file , number of PEs connected to the bypass and latency of the PFU. Hence, in this section we do a sensitivity analysis to understand the impact of these constraints.

PFU as described earlier is a grid of PEs. We try to vary the number of columns in the grid from one to four, but keeping the number of rows fixed to three as shown in Figure 4.15. In SPECINT the 2x3 grid is the best performing configuration. For SPECFP, the 2x3 grid provides performance close to that provided by a 4x3 grid. Therefore, we choose 2x3 grid configuration for PFU.

Note, however, that in some cases increasing the number of columns in the grid (e.g. wupwise, applu, twolf) results in a lower performance improvement. The main reason for this is the fact that the heuristic for fusion, fuses as many micro-ops possible. Thus, in some cases, micro-Ops that are independent are also fused,and so delay in input for one micro-op delays the execution of the CMOP.

The write ports of the Physical Register File is another important constraint. For the purpose of this study we choose two grid configuration 2x3 and 3x3. For 2x3 grid we vary write ports (wp) from four to six, while for the 3x3 grid write ports are varied from six to eight as shown in Figure 4.16.

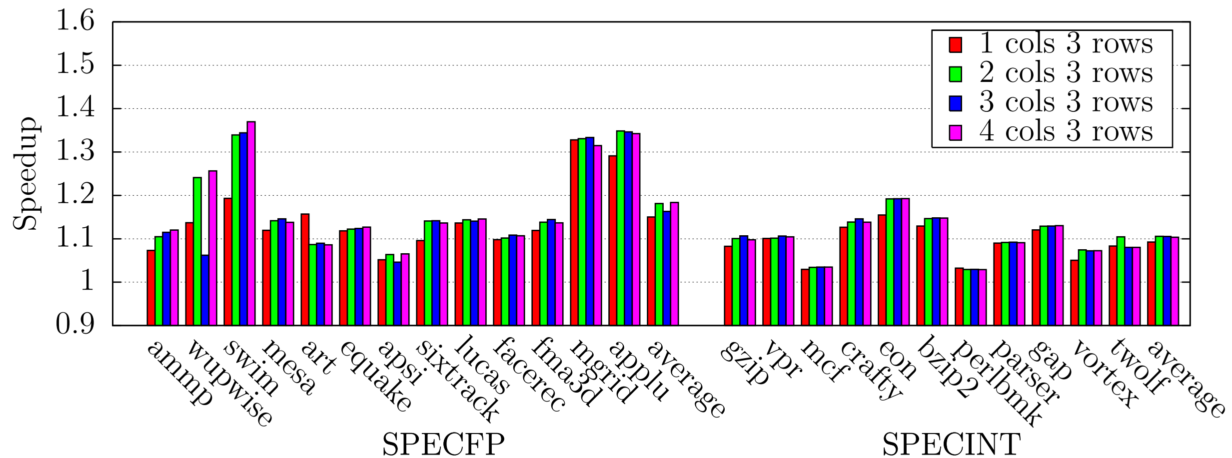


Figure 4.15: Impact of grid size. The number of rows is kept fixed to 3, while the number of columns are varied from one to four. PFU latency = 1, bypass to PE = unbounded, Physical Register File Write Ports = 4.

Note that, although in SPECFP 3x3 grid with six write ports is the best performing configuration but it exceeds 2x3 grid by not more than 1%. In SPECINT 2x3 grid with six write ports is the best performing configuration. Note that, increasing the write ports in a 3x3 grid from six to eight, however, decreases the performance. This is again due to aggressive fusion heuristic as explained above and the behavior is evident in wupwise, art etc.

To support back-to-back execution of CMOP, register operands should be bypassed to each of the PEs of PFU. This bypass however is needed only from the ALUs in which the mv-set executes and from the LDUs where the loads execute. A fully connected bypass network, where data is bypassed to all the PEs of PFU is not complexity-effective. Our simulation results suggests a design point with both the PE in the first row and one PE each in the second and the third row provide performance within 0.5% of a configuration where all the PEs are connected to the bypass.

We chose four different configurations, from all the PEs to only one PE in each row connected to the bypass. The tags r1-r2-r3 in Figure 4.17 indicate the number of PEs in the ith row connected to bypass. A 2x3 grid with 2-1-1 input constraint appears to be a good trade-off between complexity and performance.

On the other hand, PFU execution latency also is another important factor that contributes to performance gain. After all, fusing a chain of micro-ops and executing them in fewer cycles have been shown to provide benefit [26, 52, 96, 98]. Interlock collapsing ALUs [52] collapses two ALUs and execute in a single cycle. Moreover, as mentioned in Section 4.2.3, we do not introduce forwarding logic from PFU to other ALUs is removed. Based on the above observations, we consider two PFU execution latencies of 1 cycle or

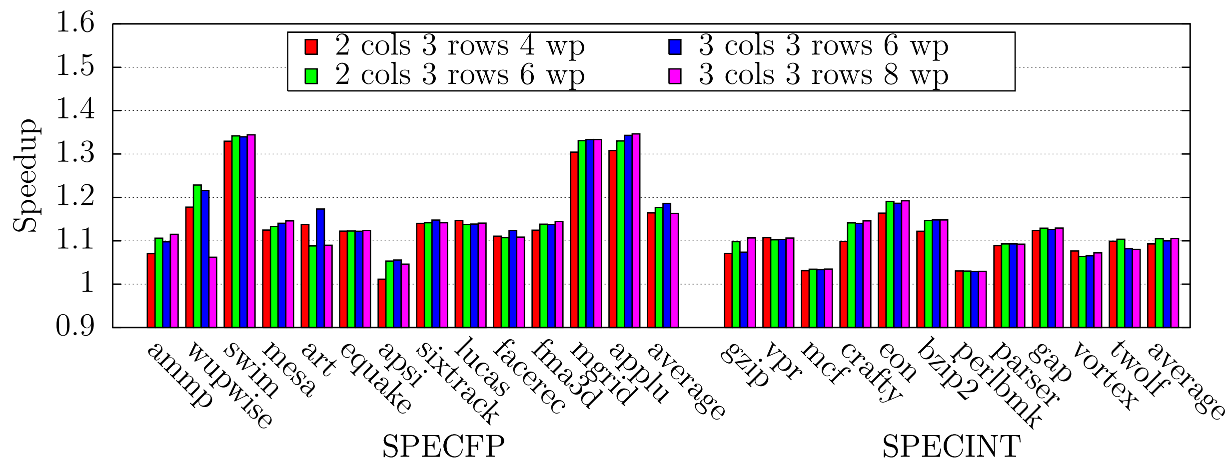


Figure 4.16: Impact of increasing conventional register file Write ports. PFU latency = 1, bypass to PE = unbounded.

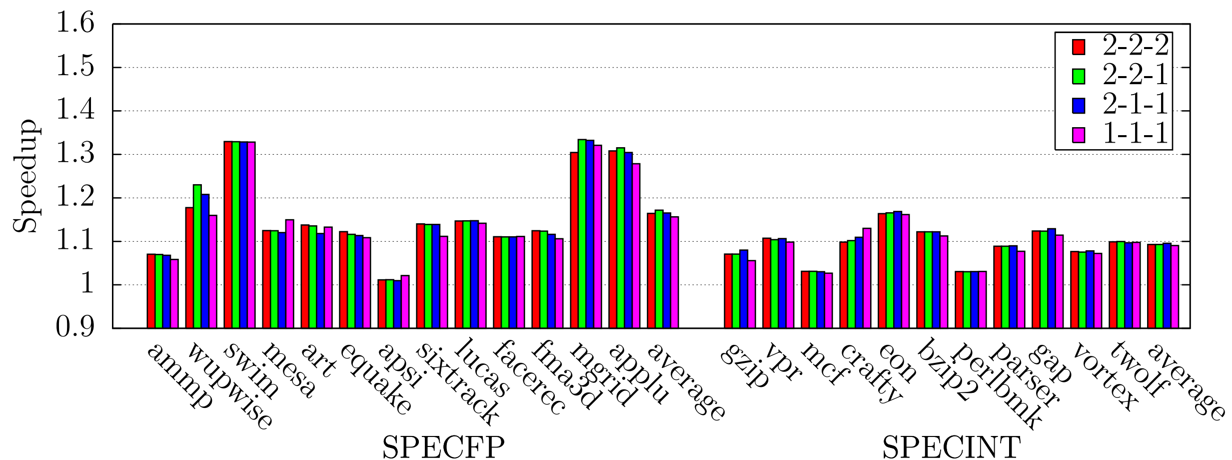


Figure 4.17: Impact of constraining the bypass to PEs. PFU latency = 1, Physical Register File Write Ports = 4.

2 cycles[5].

Figure 4.18 shows the effect of execution latency on a 2x3 and 3x3 grid configuration. SPECINT is particularly sensitive to this increase in latency. Mesa, however, shows a reverse trend, recollect that our code fusion algorithm tracks the performance of fusion by scheduling the generated instructions. If the fused MOP degrades the performance it is discarded. This results in few MOPs being generated, which provides better performance than aggressively fused MOP.

[5] We chose this experiment because we could not determine the cycle time of the processor. The cycle time of a processor depends upon a lot of factors that are beyond the scope of this thesis. Hence we consider different latencies that we believe are reasonable and are based on previous work[26]

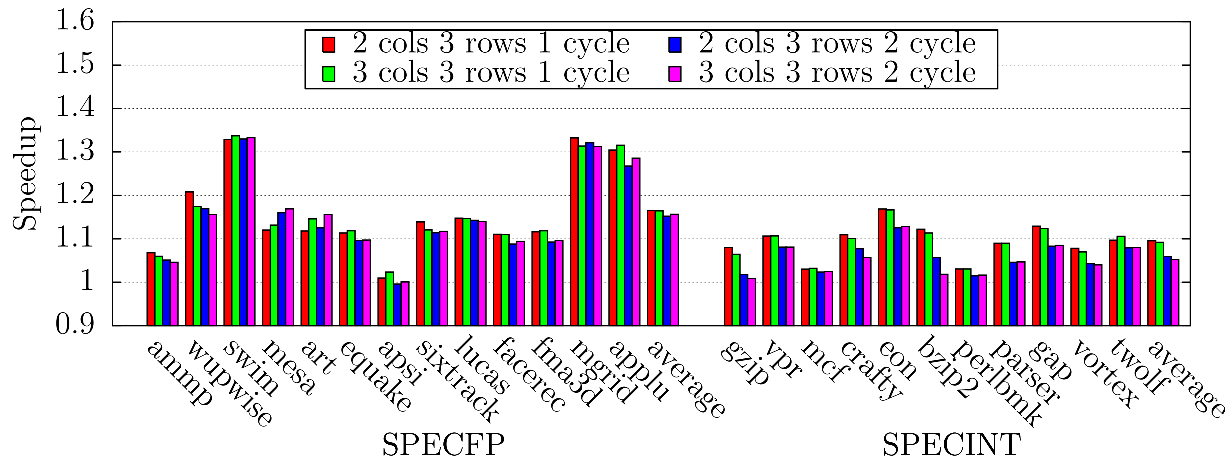


Figure 4.18: Impact of varying PFU latency. Bypass to PE = 2-1-1, Physical Register File Write Ports = 4.

4.5.2 Impact of Fusion Heuristics

In the HW/SW co-designed processor the fusion heuristic also plays an important role in improving performance. This impact in performance is clearly visible in all the benchmarks as shown in Figure 4.19. In a couple of benchmarks, such as lucas and wupwise, the *Basic Fusion* actually slows down the application. Recollect that our MOP-fusion algorithm tries to fit in as much micro-ops as possible. As a result of which artificial dependencies are created, leading to lengthening the critical path.

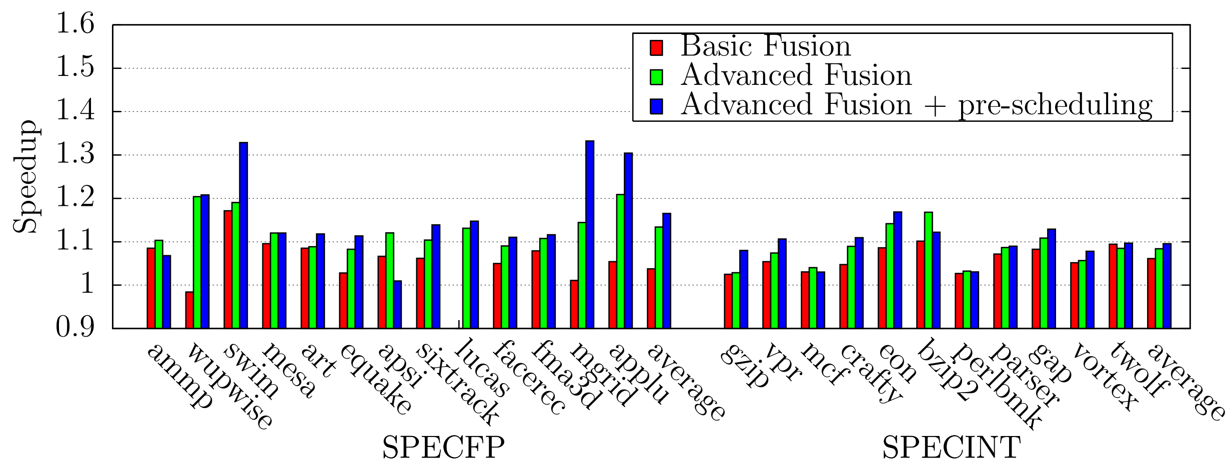


Figure 4.19: Impact of Fusion Heuristics. *Basic Fusion* stands for the case when instruction fusion is guided with the performance objective function. *Advanced Fusion* stands for the case when the performance objective function is extended in order to scheduled multiple iterations of loopy superblocks. *pre-scheduling* is a code optimization technique where instructions are re-ordered aggressively. This provides a better opportunity for fusion. PFU latency = 1, bypass to PE = 2-1-1, Physical Register File Write Ports = 4, Cols = 2, Rows = 3.

As mentioned in Section 4.4, performance of fusion is evaluated using a performance objective function based on a scheduling step. However, the scope of this monitoring is limited to the superblock. Getting a global scope for all superblocks is a cumbersome task. However, for loopy superblocks in order to get a semi-global scope, multiple iterations of the superblocks are scheduled. The impact of this optimization can be seen from Figure 4.19 by the bar with tag *Advanced Fusion*. The performance benefit in wupwise, mgrid and applu are particularly noticeable. This is simply because many of the superblocks in these applications are loops.

As discussed earlier in Section 4.4, fusion is preceded by pre-scheduling. Superblocks are aggressively reordered in the pre-scheduling phase and a new program order is obtained. The impact of pre-scheduling step can also be seen from Figure 4.19 in the third bar with tag *Advanced Fusion + pre-scheduling*. This aggressive re-ordering can have a negative impact in performance, as evident in ammp, apsi and bzip2. This happens because some store instructions are pushed down, resulting in loads from succeeding superblocks to be delayed.

4.5.3 Impact of Mov-set

From Figure 4.15 we have concluded that a PFU configuration with two columns and three rows is a good trade-off. Moreover, in Subsection 4.5.1 we conclude that only four of these six (2*3) PEs need input from bypass or the Physical Register File. This provides us with an opportunity of removing mov-set entirely. The number of read ports in Physical Register File is eight and can satisfy the need of PFU. However, Load-set are still required and they write their outputs to both the Physical Register File and the Internal Register File. Consequently, the cmop contains both multiple source and destination operands, subject to a total of four operands.

The benefit of removing the mov-set is significant and is shown in Figure 4.20. On an average, we obtain a performance benefit of 29% and 19% for SPECFP and SPECINT, respectively. This is a speedup of nearly 10% over the version with mov-set. This benefit is primarily due to gain in back-to-back execution between the producers of cmop and the cmop. Furthermore, by removing mov-set we reduce the dynamic instruction count and reduce the pressure on execution resources of the processor.

4.5.4 Comparison with alternate designs

As mentioned earlier chaining of FUs has been shown to provide performance benefits[26, 52]. Also at the same time vector functional units increases performance[40, 74]. Our proposal tries to leverage on these two facts in a more efficient way with little change over the baseline architecture.

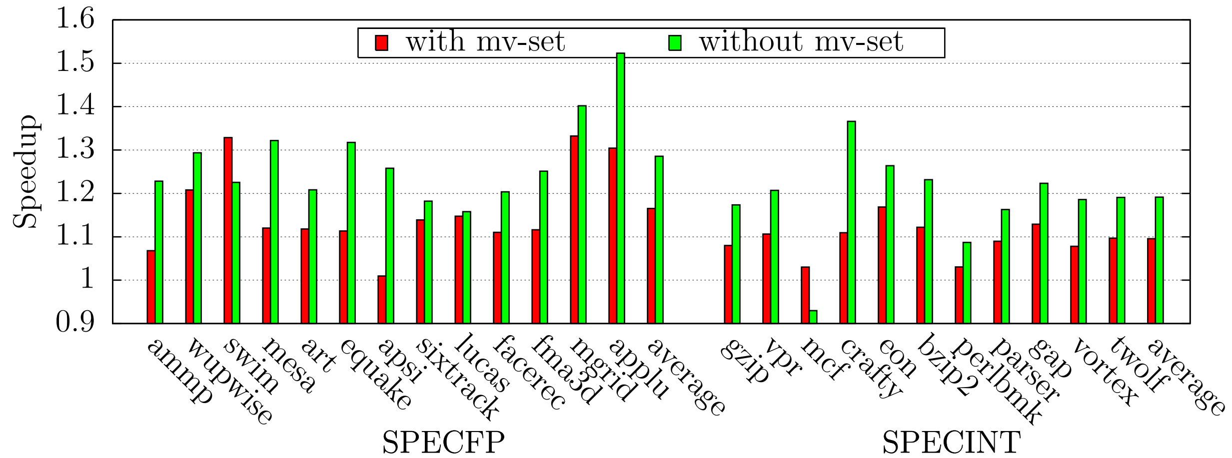


Figure 4.20: Impact of removing mov-set. PFU latency = 1, bypass to PE = 2-1-1, Physical Register File Write Ports = 4, Cols = 2, Rows = 3.

For the purpose of putting our proposal in perspective, we have compared our proposal with other alternatives to increase the effective execution of the processor. First of all we have compared our approach with an scheme that performs dynamic auto-vectorization or SIMDification [82]. In the implemented auto-vectorization scheme the VMM generates SIMD instructions using the similar code generation heuristics as described earlier in 4.4.

However, instead of fusing instructions to reduce the total execution latency here the instructions are fused in order to be executed in the SIMD units and therefore increasing the effective execution of the machine. The SIMD instructions considered are SSE2 and hence act upon four 32-bit operands in parallel. The impact of the auto-vectorization performance is shown in Figure 4.21 by the bar with *dynamic SIMD* tag.

The 2x3 PFU grid adds six more ALUs to the baseline processor, whereas the baseline processor does not have additional FUs[6]. Hence, for the sake of a comparison with an alternate process with similar execution resources, we chose two variants of the baseline out-of-order processor. In order to utilize the additional execution resources we increase the issue width to eight and double the number of read and write ports.

In the first eight-wide issue variant as shown in Figure 4.21, by the bar *8-wide ALU*, we add six more ALUs to the baseline processor and increase the issue width from four to eight. The results show that a HW/SW co-designed processor outperforms an eight-wide issue machine. Note that, in this eight-wide issue variant the bypass is provided from all the FUs to all the newly added ALUs. Whereas in our model we provide a limited bypass as evaluated in the section 4.5.1.

In the second eight-wide issue variant we consider a more aggressive design, in which

[6]Note that, our proposed processor with PFU has an issue width of four and is further constrained by total number of read and write ports.

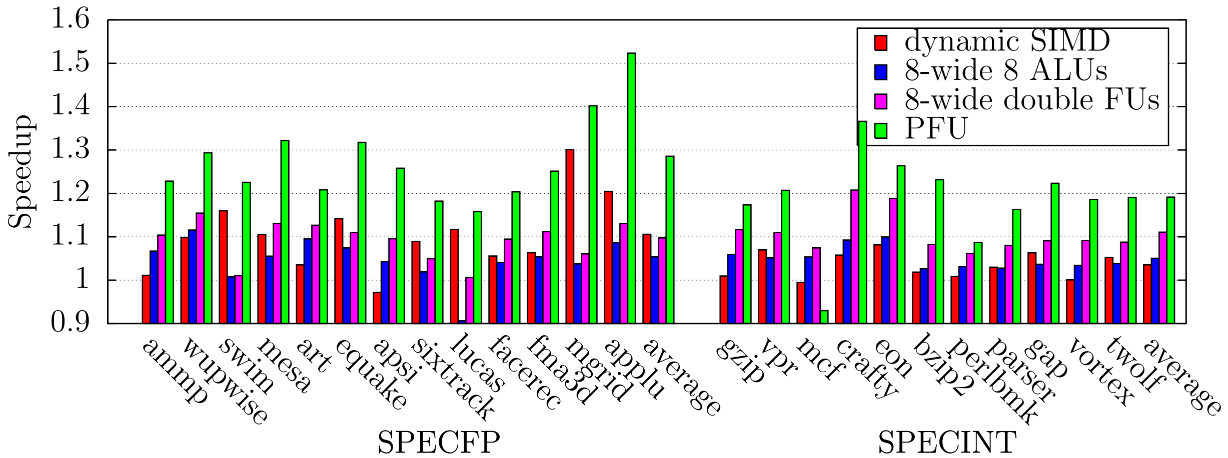


Figure 4.21: Comparison with SIMD and 8-way out-of-order. Two versions of 8-way issue out-of-order were chosen. One with six additional ALUs and one with doubling all the FUs. PFU latency = 1, bypass to PE = 2-1-1, Physical Register File Write Ports = 4, Cols = 2, Rows = 3, No mv-set.

all the FUs are doubled, including memory and complex FUs such as, FPDIV. Such an eight-wide issue machine performs well in case of SPECINT and is evident in Figure 4.21 by the *8-wide double FUs* bar. Adding more store and load units has a stronger impact in performance in SPECINT. However, our proposal outperforms this eight-wide issue processor without doubling any memory or complex FUs.

Doubling the issue width of the processor has an impact not only in the area, but also on the cycle time. Moreover, a fully-connected bypass network is required which adds further to complexity. Hence, we argue that our HW/SW co-designed PFU approach with limited bypass and an issue width of four is complexity-effective as well.

Furthermore, in our co-designed processor, as mentioned earlier in Section 4.2, we do not increase the number of ports in either of the following the register file, the rename tables. This implies that if a CMOP is being renamed and has four destination operands, then only the CMOP could be renamed in that cycle. The same condition holds true when a CMOP is being committed.

Figure 4.22 compares our proposed co-designed processor with an out-of-order processors that has all the resources doubled, such as ROB is 256 entry and so is Physical register File size. The details of the microarchitecture can be found in Chapter 3, Table 3.4. The performance of the PFU based co-designed processor is very close to that of the 8-wide processor in SPECFP. This is primarily because superblocks in SPECFP are larger. As a result, the superblocks in SPECFP enjoy the same benefit that a large instruction window processor does.

Moreover, the SpecRRT mechanism allows early release of ROB entries and the Physical Register File entries. This in a way creates the net effect of a large instruction window

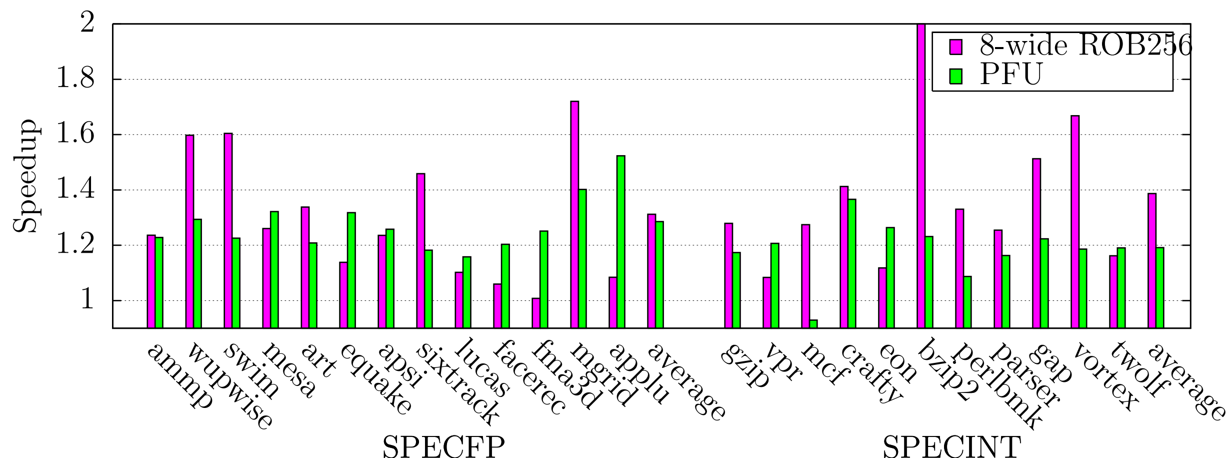


Figure 4.22: Comparison with 8-way ROB 256 out-of-order. All the resources of the processor has been doubled. PFU latency = 1, bypass to PE = 2-1-1, Physical Register File Write Ports = 4, Cols = 2, Rows = 3, No mv-set.

---

out-of-order processor. Furthermore, the PFU helps in reducing execution latency of the fused instructions. However, for SPECINT, the 8-wide processor outperforms our proposal by 17%. This is due to smaller superblocks in SPECINT.

4.5.5 Qualitative Discussion on PFU

A significant fraction of the execution cycle in out-of-order processors is dedicated to operand forward logic [41, 76]. In our PFU design we remove the operand forward logic from PFU to other FUs and dedicate this fraction of execution cycle to execution.

On the other hand, researchers [26, 78] have shown that chain of simple ALUs can be collapsed in a single cycle. CCA [26], for instance, have validated this conclusion by evaluating their design for various depth levels. Hence, based up on the above two arguments a single cycle or at-most two cycles is a reasonable estimate for the execution latency of the PFU.

In Section 4.2.3 we described the organization of PFU. We described the two major components of the PFU 1) the distributed internal register file, and 2) grid of PE. The distributed Internal register file consists of a grid of register files with five columns and three rows. Each register file has four entries and has at-most four read ports. Four write ports are provided for the inputs from ld-set and mv-set. The total size is 240 bytes. Since, this register file is both small and has fewer read ports it is not in the critical path.

The grid of PE consists of ALUs connected in a dataflow fashion and a distributed configuration cache. The size of distributed cache is estimated to be 210 bytes. Since, the request of loading the configuration from I-cache is made in the rename stage we try to

remove the configuration access from critical path. However, in case of unavailability of space configuration cache the issue is stalled. Our experiments have indicated that a configuration cache with entries for five different configurations is sufficient. The selection logic that extracts the corresponding control signal is fairly simple and each PE can extract its control signal in parallel.

4.6 Related Work

Application acceleration through accelerators have been subject to extensive research. The basic principle behind such an approach is to identify recurring instruction patterns and execute them on specialized function units. The identification process can be done by a static compiler by feeding back the profile information. Hardware dynamic optimizers could also perform this task. The identification process is not only limited to identification but also to generating the fused instruction that executes on the accelerators.

These accelerators can be broadly classified into coarse-grained or fine-grained. A coarse-grained accelerator is the one which executes a large set of instructions that are fused together. Typically these accelerators are a co-processor. Such accelerators are more suitable for kernel oriented applications. Several such kind of accelerators have been proposed in the past[83, 30, 49, 48, 45].

Most of these are FPGA based such as [45, 49, 30, 83]. Piperench[45] was proposed to accelerate multimedia streaming applications. Xputer[48] was proposed for Digital Signal Processing applications. In contrast to coarse-grained accelerators, in this chapter we have proposed a fine-grained accelerator that is present in the processor's datapath along with other functional units.

Moreover, many different types of fine-grained accelerators [26, 98, 96, 52, 81, 20] have been proposed in the past, as well. Most of them [26, 96, 20, 81] extend the ISA. The fused instruction that executes on this accelerator are programmed using static code generation scheme. However, as applications evolve, these accelerators may require some changes from generation to generation.

As a result, adding instructions at ISA level may pose important constraints for future processor generations. In contrast in our approach the x86 ISA is not extended, instead the target ISA which is only visible to the microarchitects is extended. Moreover, in our scheme the fused instructions are generated dynamically.

However, dynamic and transparent management of these accelerators have been studied in the past. Some works [26, 98] have focused on managing accelerators with hardware-based dynamic optimizers. To keep the complexity and power costs of these hardware optimizers in check, they are design to handle simple cases. A dynamic software binary optimizer, on the other hand, is more flexible and can generate instructions using more

advanced heuristics.

Moreover, some of the fine-grained accelerators [96, 20, 81, 98] are FPGA based Functional Units that are added to the datapath of the processor. The flexibility of the FPGA comes with the price of long latency. Moreover, FPGA reconfiguration latency is usually longer. As a result, FPGAs are more suitable for applications where a large number of instructions could be offloaded. In contrast in this chapter we have proposed an accelerator that is a grid of functional units.

The work closest to the one proposed in this chapter is CCA [26]. The authors had made a key observation that when collapsing dataflow subgraphs for customized instruction set extension, the flexibility of an FPGA is generally more than necessary. Most of the dataflow subgraphs can be collapsed using a basic set of primitives such as simple arithmetic and logical functional units.

However, there are several key differences between our proposal and CCA. First of all the CCA proposes two code generation schemes, one is static compiler based, while the other one is using the hardware dynamic optimizer the rePlay framework[77]. In contrast we use a dynamic binary translator, which is flexible and provides binary compatibility as well. Moreover, CCA had focused on multimedia applications, whereas we consider general-purpose applications.

Moreover, we had proposed an internal register file was proposed to hold the input operands. A ld-set and mv-set was proposed to bring data from the conventional register file to the internal register file. Furthermore, the pre-decoded control signals corresponding to the fused instructions are stored in the configuration cache.

However, using accelerators in HW/SW co-designed approach has been proposed in past. Hu et. al. [52] had proposed a complexity-effective out-of-order processor. In their work they use ICALU [78] to execute a fused pair of dependent ALU instruction in a single cycle. They propose a VMM to find and fuse instruction pair within a superblock, same as we do.

However, they focus only on SPECINT benchmarks as it has abundance of dependent ALU instruction pair. They compared their proposal, which is a two-way out-of-order processor with ICALU, with a four-way out-of-order processor. In contrast, in this work we focus on exploiting the benefits of a larger and powerful accelerator over larger regions.

Mini Graph [15] collapses multiple instructions in order to amplify processor bandwidth and introduces a handle which is similar to a configuration. However, we proposed MOPs mainly to gain performance. The side-effect of fusing instruction is also reduction in the pressure on processor resources.

The large differences between the proposals made in the prior works as well as the completely different frameworks used for their evaluation make the comparison among the different proposals not an easy task sometimes not even possible. Therefore we have left

that as a future work.

On the other hand a recent work has considered the benefits of using a co-design virtual machine to deal with the changes on the SIMD vector ISA from generation to generation [25]. They propose an abstract ISA for SIMD instruction; a binary translation layer takes care of recompiling from abstract ISA to that of the underlying architecture. In contrast, in this work we proposed a new PFU and we show that this scheme outperforms SIMD-based accelerators. As far as we know this is the first work where these alternatives are compared.

4.7 Conclusions

A HW/SW co-designed approach is a complexity-effective way of providing high performance for general purpose application. A Cd-VM reorders a superblock and fuses a sequence of micro-ops to generate a MOP. The fusion is done by taking into account the existing microarchitectural resources and performance of fusion is monitored. A split-MOP model allows inputs to be provided both from memory and conventional register file.

A novel PFU executes the CMOP design is proposed which exploits both ILP and chaining to gain performance. We obtain average speedups of 17% in SPECFP and 10% in SPECINT. We also obtain speedups of up-to 33 % is some benchmarks. Moreover, with a slight modification in the proposed MOP model we obtain improvements in performance of 29% in SPECFP and 19% in SPECINT.

We measure the impact of various microarchitectural constraints on performance. We also demonstrate that, by introducing some code generation scheme performance is improved. We also show that our split-MOP model outperforms not only a dynamic SIMD machine but also 8-wide issue machine. Hence, we conclude that a new generation of out-of-order processors can be co-designed for higher performance in a complexity-effective way.

Chapter 5

SoftHV

In the previous chapter we had proposed a HW/SW co-designed out-of-order processor with a Programmable Functional Unit. The goal was to accelerate execution of general purpose programs executing in a modern out-of-order processor.

In this chapter, however, we pursue a low-complexity yet a high-performance processor design. In order to cut down complexity we use a simpler in-order microarchitecture equipped to run a Cd-VM Monitor. In order to achieve higher performance we propose the use to two kinds of accelerators. These application specific accelerators execute fused μ-ops either with low latency or provide higher bandwidth to improve performance. The key contributions of our proposal, SoftHV, is a high-performance HW/SW co-designed in-order processor that executes horizontally or vertically fused μ-ops.

SoftHV consists of a co-designed virtual machine (Cd-VM) which reorders, removes and fuses μ-ops from frequently executed regions of code. On the hardware front, SoftHV implements HW features for efficient execution of Cd-VM and efficient execution of the fused μ-ops. In particular, (1) Interlock Collapsing ALU (ICALU) is included to execute a pair of dependent simple arithmetic operations in a single cycle, and (2) Vector Load unit (VLDU) is added to execute a pair of independent loads.

Results presented in this chapter show that SoftHV produces average performance improvements of 9% in SPECFP and 10% in SPECINT, and up-to 28%, over a co-designed four-way in-order processor that is equipped with code optimizations. For a two-way in-order processor configuration SoftHV obtains improvements in performance of 9.5% and 11.5% for SPECFP and SPECINT, respectively.

5.1 Introduction

In this chapter we consider a HW/SW co-designed microprocessor based on an in-order core and using a Cd-VM. Using an in-order core, we drastically cut complexity and power consumption. The Cd-VM helps in optimizing code and reordering instructions by exploiting the ILP and provides performance benefits. HW support is added in order to aggressively reorder memory instructions.

Moreover, the use of a Cd-VM further facilitates the introduction of new microarchitectural features transparent to application software and operating system. Such features are only visible to the Cd-VM, which is responsible for generating appropriate code to reap benefits of these HW features. One example of this is the use of specialized or programmable functional units that allows execution of certain code sequences faster than in conventional FUs.

The use of specialized or programmable FUs results in a power-efficient way to invest the transistor budget. However, the challenge when dealing with this specialized hardware is twofold: first, the hardware needs to be designed in a way such that it fits the most common properties of the different workloads, and second code needs to generated in order to make efficient use of these FUs.

Our proposal SoftHV considers a HW/SW co-designed in-order processor where the Cd-VM performs vertical and horizontal fusion of instructions leveraging two types of specialized FUs: (1) Interlock Collapsing like ALUs (ICALU) [78] that executes a pair of dependent simple arithmetic/logic μ-ops that have been previously fused into a macro-op (MOP), that is vertical fusion; and (2) Vector Load Units (VLDU) [37] executes a pair of fused parallel loads known as a vector load (VLD), that is horizontal fusion.

ICALU provides performance improvement by collapsing a dependent pair of simple ALU μ-op in a single cycle, as already shown by Hu et al. [52]. However, in this chapter we go beyond Hu's proposal by using both ICALU and VLDU. This provides additional benefit of higher effective processor bandwidth by decreasing the dynamic instruction count.

The Cd-VM in SoftHV uses hardware profiling to form superblocks. It optimizes the superblock by performing some code optimizations such as copy propagation, limited dead code elimination, load-store telescoping, as described in Chapter 3. It then selects pair of dependent simple arithmetic operations μ-ops and fuses them to generate macro-ops (MOP). Also pair of parallel load operations are fused into a vector load instruction (VLD).

The key novelty of SoftHV is the right combination of the two types of accelerators with code generation techniques that provide higher performance for two different application suites. The two different kinds of applications suites we use are: (1) SPECINT, where pairs of dependent simple arithmetic/logic operations are very common; and (2) SPECFP, where the limited amount of architectural registers results in more spill code and therefore

the bandwidth of memory access is a critical constraint.

Performance results presented in this chapter shows that SoftHV provides average improvements in performance of 9% for SPECFP and 10% for SPECINT over a conventional four-way in-order processor. Also, SoftHV provides average improvements in performance of 9.5% and 11.5% for SPECFP and SPECINT over a two-way in-order processor, respectively.

The rest of the chapter is organized as follows. In Section 5.2 we discuss some further insights on the motivation behind using horizontal and vertical fusion. In Section 5.3, we provide the microarchitecture details of our proposal. We also discuss the microarchitecture with special attention paid to ICALU and VLDU. Pipeline stages that are impacted in order to support such a co-designed processor is discussed.

The code generation technique is discussed in Section 5.4. A detailed evaluation and breakdown of performance benefits is provided in 5.5. Finally, related work is reviewed in Section 5.6 and we conclude in Section 5.7.

5.2 Motivation for Horizontal and Vertical Fusion

SoftHV performs horizontal and vertical fusion of instructions, that is parallel instructions and dependent instructions are fused and then executed in specialized FUs. The combination of these two types of fusion is critical for a general domain processor to achieve competitive performance results for different types of workloads. For instance, SPECINT and SPECFP applications.

In case of SPECINT, pairs of dependent simple arithmetic/logic operations are very common, and therefore a vertical fusion of dependent instructions is a key feature for a processor in order to reduce the latency of the more critical instructions.

On the other hand, SPECFP presents a higher degree of parallelism. Such a degree of parallelism combined with limited amount of architectural registers results in more spill code. In turn this results in an increase of memory accesses. However, it is also frequent that spilled-code consists of many pairs of parallel loads that depend on a single source operand register and differ only in the immediate offset. Therefore a proper hardware to execute vector loads reduces the amount of accesses and effectively increases the throughput of the machine.

5.3 SoftHV Architecture Overview

SoftHV is a HW/SW Co-designed Virtual Machine (Cd-VM) system with an in-order microarchitecture. As described earlier in Chapter 2, a Cd-VM consists of a Virtual Machine Monitor (VMM). As mentioned earlier, in order to gain performance we have used two kinds of functional unit based HW accelerators.

The first accelerator the ICALU [78] executes a pair of dependent ALU instructions in a single clock cycle. The second accelerator the VLDU executes a pair of independent loads, that depend upon the same base register. The fused instructions are generated by the VMM at the time of superblock formation. A simple but efficient fusion heuristic is proposed in order to decide which instructions to fuse and how(vertically or horizontally).

5.3.1 Microarchitecture Overview

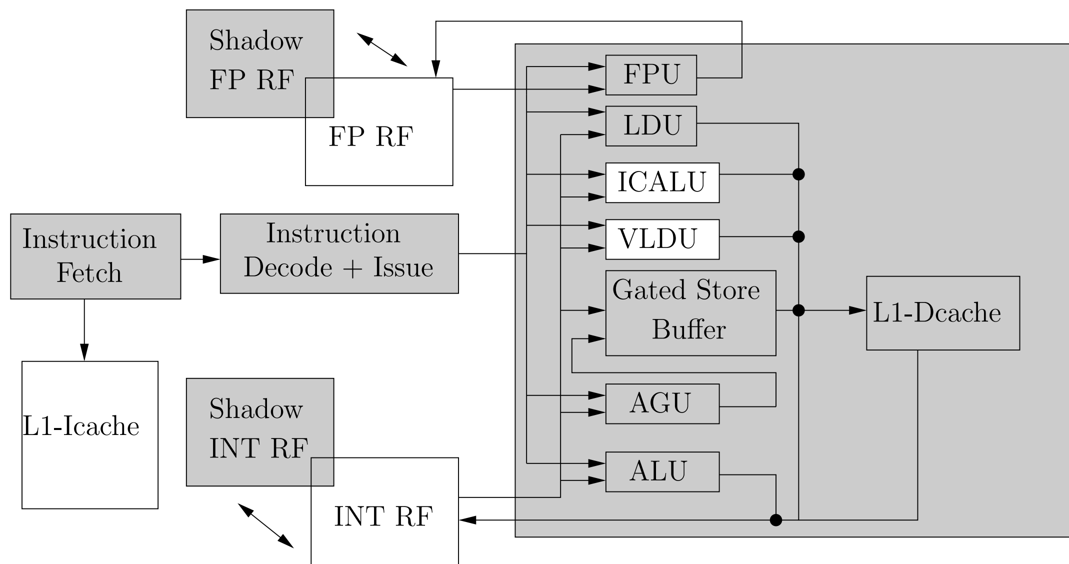


Figure 5.1: The Co-designed Processor Overview for a 2-way in-order configuration. A decoupled in-order microarchitecture is used, and both the ICALU and VLDU are added to the datapath of the processor. Normal loads are sent to ALUs for their address generation, followed by access to D-cache. Shadow Register Files holds the committed state, whereas the Working Register Files holds the speculative state. Stores are held in Gated store buffer until commit.

Figure 5.1 provides an overview of the proposed co-designed processor based on decoupled in-order microarchitecture. Fetch buffers decouple fetch from decode while issue queues decouple decode from the execution back-end of the processor.

A two-level decoder as proposed in [52] is used. The ability to execute x86 code natively

avoids the need for an interpreter and hence, the slowdown associated with it. X86 instructions are decoded to RISC-like μ-ops. μ-ops are seven bytes in length. The first bit of the μ-op indicates whether the μ-op is fused with the immediately following μ-op. We implement a similar host μ-op ISA as proposed by HU [52].

Cd-VM allows dynamic and aggressive code optimizations compared to a traditional compiler. However, in order to support these the underlying microarchitecture should provide mechanisms to detect miss-speculations and take the necessary corrective action.

The SoftHV Bulk Commit Mechanism is very similar to that of Transmeta, and is described in Chapter 3, in Section 3.5.2. The register state and the memory state is committed at bulk when all the instructions corresponding to the superblock have successfully executed. The x86 register state is shadowed by maintaining two copies a working copy and a shadow copy as shown in Figure 5.1.

The pipeline Back-End

A FIFO based issue queue is used, which holds the μ-ops until they are ready. The readiness of an instruction is determined using a scoreboarding mechanism that predicts when operands will be available, using counters. These counters holds the number of cycles until a valid result will be available for forwarding. The value in each scoreboard entry counts down every cycle. Such a mechanism ensures back-to-back execution. The accelerators ICALU and VLDU co-exist with other FUs in the datapath as shown in the Figure 5.1.

Scalar loads are executed in LDU, with their address being generated. After that their addresses are translated and the load access the D-cache and the Gated Store Buffer.

Stores, on the other hand, are first sent to AGU and then their addresses are translated. Finally, the store data is placed in the Gated Store buffer. The entries from the gated store buffer are held until all the μ-ops of the superblock have successfully executed.

5.3.2 HW Accelerators : ICALU and VLDU

Based on the characteristics of SPECINT and SPECFP applications, we introduce two functional units namely ICALU and VLDU. ICALU collapses a pair of dependent ALU μ-op and execute in a single cycle. Only a specific pair of dependent ALU μ-ops are allowed, while shifts among others are not considered. Such a fused pair of μ-ops is called as a macro-op (MOP).

VLDU, on the other hand, executes a fused pair of independent load μ-op. Only those loads that have same base register and differ in the immediate offset are considered. Such a pair of fused loads are called a vector load (VLD).

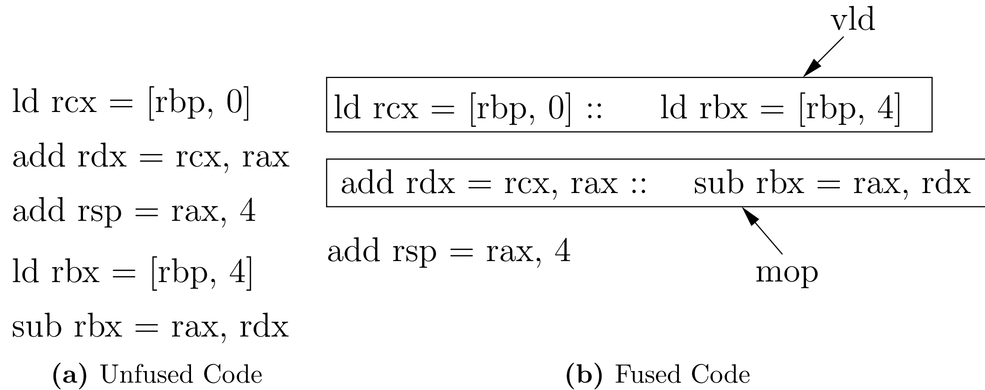


Figure 5.2: Code before and after fusion. The right side shows a pair of independent loads fused as a vector load. Next is a pair of dependent ALU instructions fused as a mop.

Cd-VM not only enables in transparently introducing these functional units, but also providing a larger scope and more opportunity for using these functional units. Superblocks are analyzed and pairs of dependent ALU and independent load μ-ops are extracted. These pair of μ-ops are identified by setting the first bit of the μ-op to indicate whether its fused.

Figure 5.2 shows a code snippet before and after fusion. Note, how the fused μ-ops are placed together in the generated code. In this example, a pair of fused loads is followed by a pair of fused addition and subtraction as shown in Figure 5.2b. Fused μ-ops are identified by the first bit of each μ-op.

The two fused loads share same base register **rbp**, but differ in immediate offset by four bytes. Such a pair of loads is very common as they access different elements of a data structure, or different entries from an array. Hence, clubbing them together as a single entity increases processors effective bandwidth.

The **sub** μ-op in Figure 5.2 is dependent on **rdx** which is produced by the **add** μ-op. This fused μ-op pair not only provides benefit by executing the dependent pair in a single cycle but also provides better processor resource utilization. For instance, the sub μ-op does neither have to access Physical Register File nor the bypass network for **rdx**.

5.3.3 Interlock Collapsing ALU

ICALUs [78] consists of a pair of ALUs collapsed to execute a pair of dependent instructions in a single cycle. Figure 5.3 presents a block diagram of ICALU. Only a subset of ALU instructions are considered that satisfy constraints such as number of input operands,

simple arithmetic operations, logical operations; Shift and Add-shift are not considered. The fused pair of μ-ops are executed in a single cycle instead of two cycles, leading to speedup.

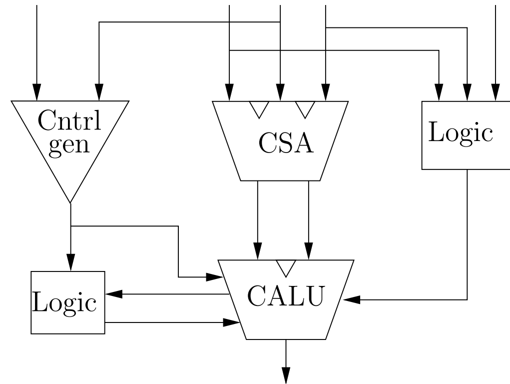


Figure 5.3: Interlock Collapsing ALU. Carry-Save Adder produces inputs for a Controlled ALU.

Introduction of ICALU not only provides speedup by reducing the execution latency but also provides better utilization of microarchitectural resources such as issue width, read/write ports, buffers etc.

Moreover, our experiments have showed us that most of the times the alu μ-op that is the head in the pair produces value only for the μ-op it is paired with. This implies that the head μ-op in the MOP should not be a live-out for the region, and neither should its value be consumed by any other μ-op. As a result, the head μ-op of a MOP is not sent to any ALU.

This has an additional side-benefit of reducing pressure on microarchitecture resources such as read-ports, issue queue entries, forwarding network among others. Figure 5.8 shows the code coverage of a pair of fused μ-op is nearly 30%. This gives an estimate of reduction in dynamic instruction count, leading to efficient processor resource utilization.

5.3.4 Vector Load Units

Vector Load units enable execution of data parallel loads as a combined instruction. Only those loads that have same size, same operand, but access different banks are fused. x86 ISA consists of many loads that have same start address but different offset.

Such a kind of vector loads have been proposed already in Altivec [37]. However, there are several differences between the vector loads we propose and that proposed in Altivec. Firstly, in the context of atomic superblocks more such pairs of independent loads can be found and fused. This is because load-hoisting enables reordering loads not only within

a basic block but also across basic block boundaries. Secondly, the vector loads that we propose need not be to comprising of loads to contiguous memory location.

Thirdly, our vector loads write to separate registers, whereas the vector loads in Altivec write the merged data into a single register. Moves are needed to extract data from these registers for Altivec vector loads. Adding moves impacts the back-to-back execution of the dependents of the loads. So in a sense a vector load is like a VLIW instruction where independent μ-ops are bundled together and they execute in their respective FUs and writeback to their own registers.

A vector load is very similar to two independent loads executing in parallel in a superscalar pipeline. However, by fusing the loads together the issue width is effectively increased, as two μ-ops are issued in place of one. However, the number of write ports and number of read ports in the Register File is kept unchanged, which limits the potential, at the benefit of low complexity.

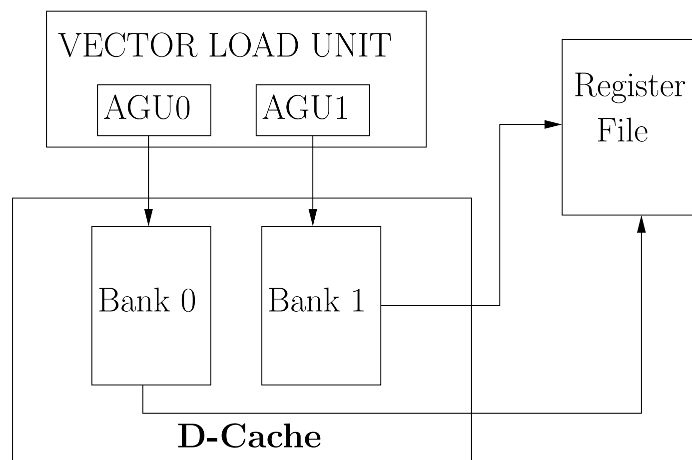


Figure 5.4: Illustration of VLD execution. Replicated address generation units generate the addresses of the fused load pair in parallel and sends to translation unit and issues load request to the D-cache in parallel.

Figure 5.4 illustrates how a VLD generates address and reads data from D-cache for both the fused loads. A VLDU consists of a pair of Address generation unit in order to generate address for both the loads in parallel. No scalar loads could be issued in the same cycle in which the VLD is being issued.

Request is sent to the respective banks. In case of a bank conflict, processor stalls the back-end for a single cycle while the bank conflict is being resolved. Data for both the loads are then available in next cycle. Once the data is received it is written to the respective registers in the writeback stage. Note that our baseline in-order processor also uses multi-banked D-Cache.

Scalar loads, on the the other hand, are sent to LDU for their address to be generated and translated, which is followed by an access to the Gated Store buffer or the D-cache in parallel. If the data is available in Gated Store Buffer, then the data is forwarded and written back.

Moreover, in case the load partially aliases with a store in the store buffer, then the superblock is rolled-back. The unoptimized Basic Blocks are fetched. An alternative solution would be to read the data partially from the store buffer and partially from the D-Cache. However, such cases were quite infrequent.

Handling Cache Miss

Cache-miss for scalar loads results in stalling of the back-end of the pipeline. Similarly, on a cache-miss for any load fused in the vector load only the back-end pipeline is stalled. The usage of an in-order core with non-blocking cache makes handling of vector loads much simpler.

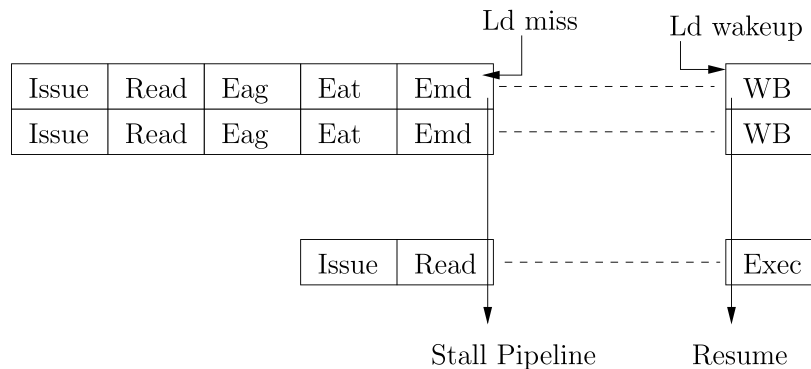


Figure 5.5: VLD Miss Handling. In case any of the load misses the pipeline is stalled. When the data arrives both the loads are woken up and the pipeline is resumed. Note that only the back-end pipeline is stalled.

Figure 5.5 illustrates how a VLD miss is handled. If any of the load in the VLD misses, the back-end pipeline is stalled. Only when the data arrives the activity is resumed, and loads write-back to the registers.

5.3.5 Instruction Encoding

Figure 5.6 shows the instruction encoding of a macro-op (MOP) and a vector load (VLD). We use an instruction encoding similar to [52], but extended to encode instructions that use VLD accelerator. The first bit “F” indicates whether the corresponding μ-op is fused

with the μ-op immediately following it. If F bit is set μ-ops are fused prior to decode, and are held as a single entry in reservation station.

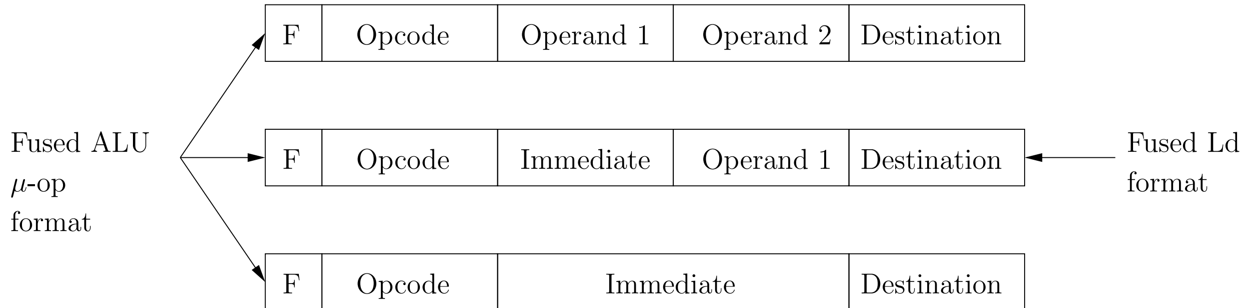


Figure 5.6: MOP/VLD instruction encoding. All the three encoding format is supported for fused μ-ops that form a MOP, whereas only the second format is applicable for loads that are part of VLD. If the "F" bit is set then the μ-op forms a pair with the following μ-op.

For vector loads only the second instruction format is valid. Our internal μ-ops support $register + offset$ instruction encoding for loads. Since, the opcode is redundant the μ-op pair when fused will only have a single opcode.

5.4 SoftHV Binary Optimizations

Co-designed virtual machine (Cd-VM) plays an important role in dynamically optimizing the code for an efficient use of ICALU and VLDUs. The optimization process implemented by the proposed Cd-VM is shown in Figure 5.7.

Few of the code generations steps, such as superblock formation, code optimization, dataflow graph generation and register allocation, are standard steps in most dynamic compiling systems. These steps have already been discussed in great details in Chapter 3. Hence, we will only discuss the steps that are relevant in the context of SoftHV.

5.4.1 μ-op Fusion

A single pass forward scan fusion algorithm similar to [52] is proposed. However, our proposed heuristic not only fuses a pair of dependent simple ALU μ-ops but also a pair of parallel loads. The algorithm is greedy and tries to find the first suitable μ-op to fuse with the current μ-op as described in Algorithm 4 below.

The heuristic checks whether a μ-op is already fused, as shown in Line 2. Then depending on whether a μ-op is load or a simple ALU operation, a fusible μ-op is searched for within

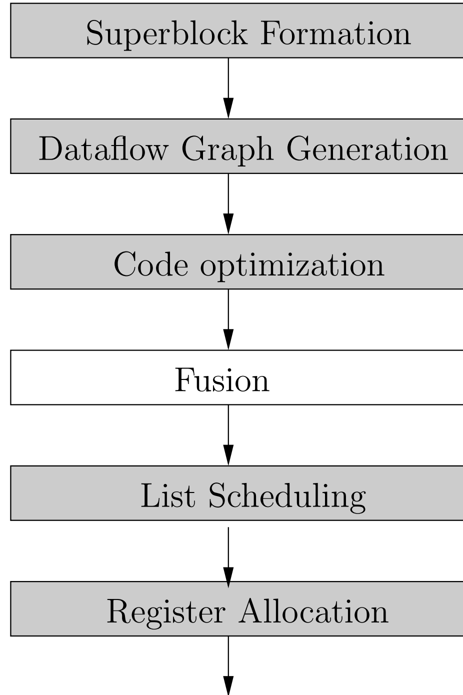


Figure 5.7: Code Generation Flow Chart

the superblock, as shown in Lines 5 and 7. If a fusible pair found then the pair of μ-ops are fused 10.

A pair of dependent ALU μ-ops are executed in the ICALU. ICALU considers only a specific pair of simple μ-ops such as arithmetic operations, logical operations (except shifts), register transfer operations, address generation and branch outcome determinations. For complete details on the pair of ALU μ-ops that are allowed the readers are referred to [78].

On the other hand, a pair of independent loads are executed in the VLDU. Only certain pair of independent loads are considered, which have a same base register but differ in the immediate offset. Loads with multiple source register operands are not considered for fusion.

Such a fused pair of μ-ops is called a macro-op. After considering a macro-op, the algorithm proceeds by considering μ-ops which were not included in the previous macro-ops. The algorithm proceeds in the similar fashion until all the μ-ops of the superblock have been considered.

Algorithm 4 Fusion Algorithm

```
1:  for all μ-op do
2:     if μ-op already fused then
3:        continue
4:     end if
5:     if μ-op is a load then
6:        scan forward for fusible parallel load
7:     else if μ-op is a simple arithmetic/logic op then
8:        scan forward for a dependent simple μ-op
9:     end if
10:    if pair found then
11:       mark as a new fused pair, by setting the fused bit of the head μ-op
12:    end if
13: end for
```

5.5 Performance Evaluation

In this section, the benefits in performance due to vertical and horizontal fusion is measured, first in isolation. Their combined benefit is studied as well. We report results both for a 2-way and 4-way in-order processor core that is described earlier in Chapter 3, Section 3.5.2.

5.5.1 Code Coverage of Fused Instructions

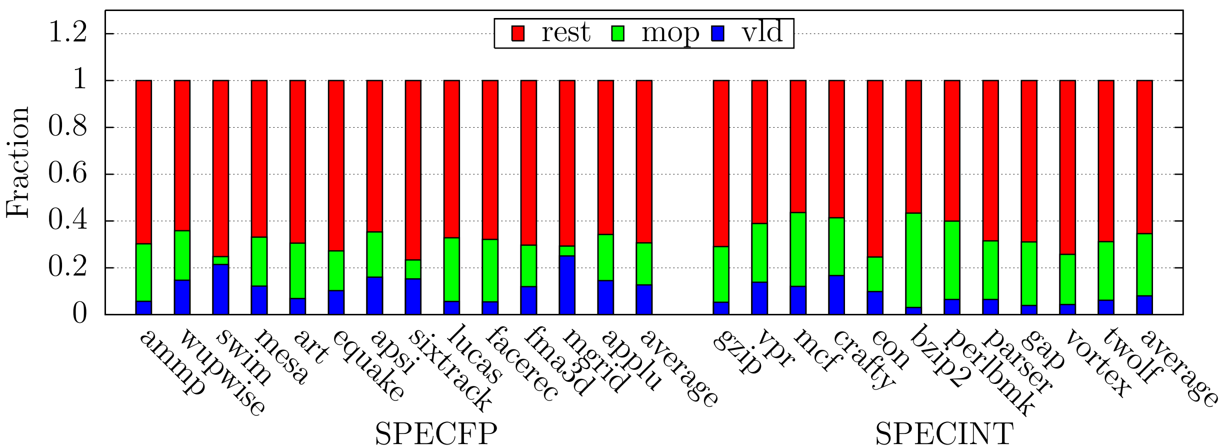


Figure 5.8: μ-op Code Coverage. *mop* stands for the code coverage provided by a pair of dependent instructions and *vld* stands for the code coverage provided by a pair of independent instructions. *rest* stands for the code coverage provided by rest of the instructions.

Figure 5.8 shows the code coverage provided by the instructions that are paired together

(*mop* and *vld*, in the bottom of stacked histogram). As described earlier, in Section 5.4.1, we use a heuristic that finds pair of independent loads or a pair of dependent ALU μ-ops in a superblock and pair them.

Together the μ-op pairs provide a good coverage for both SPECFP and SPECINT benchmarks. However, the coverage provided by these μ-op pairs are very different for SPECINT and SPECFP. SPECINT consists of 28% of μ-ops fused as a pair of dependent μ-ops (mop) and 7% of parallel loads (vld). SPECFP, however, has 12% of loads fused as a pair and 20% (mop) for dependent μ-ops.

5.5.2 Performance benefit due to horizontal and vertical fusion

In this section, we study the benefits of ICALU and VLDU. First, we study their benefits in isolation and then their combined benefits. The experiments are run by using the optimized and scheduled superblocks and by adding ICALU and VLDU first in isolation and then together.

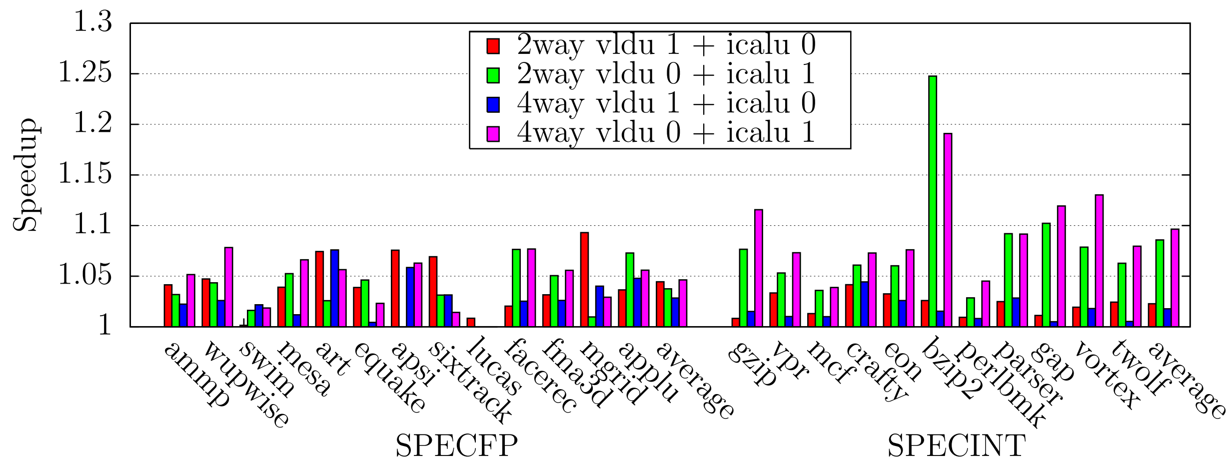


Figure 5.9: The Effect of adding ICALU and VLDU in Isolation. The first two bars measures the effect of adding ICALU and VLDU separately for a 2-way in-order processor. The next two bars repeats the experiments for a 4-way in-order processor. Speedups are normalized to the respectively n-way in-order processor configuration.

Figure 5.8 shows the weighted distribution of the fused μ-ops. It shows that nearly 32% and 35% of μ-ops are fused as mop or vld in SPECFP and SPECINT, respectively. However, the proportion of μ-ops fused as mop or vld is different for SPECFP and SPECINT. A low ILP in SPECINT leads to many short dependent chains. As shown in Figure 5.8, 28% of μ-ops are fused as MOP, while SPECFP contains 20%.

As a result of which, SPECINT obtains more performance benefit from ICALU 10% additional speedup for a 4-way SoftHV configuration as shown in the Figure 5.9 (**4way vldu**

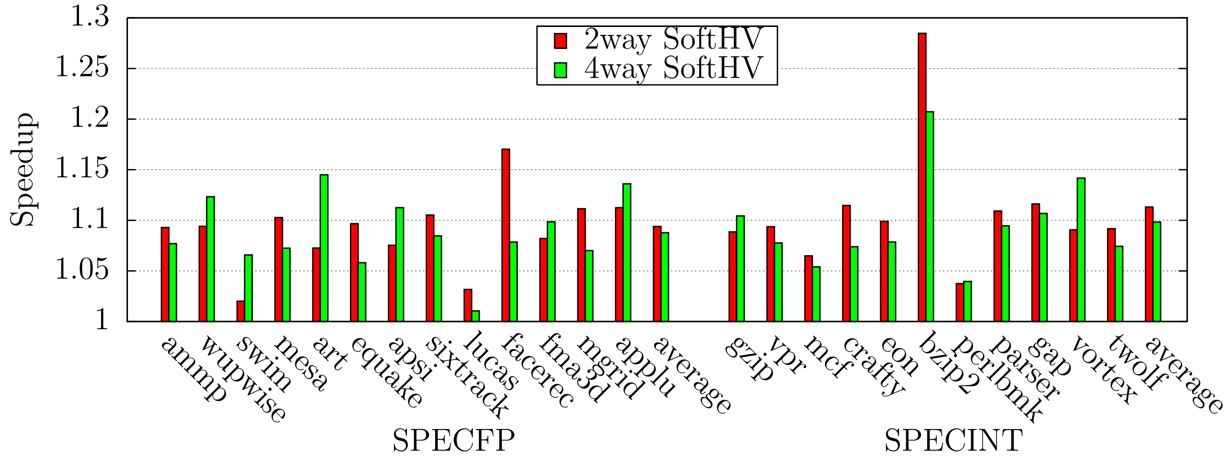


Figure 5.10: The combined Effect of ICALU and VLDU. First bar measures the speedup for a 2-way in-order processor and the section bar measures the same way for a 4-way in-order processor. Speedups are normalized to the respectively n-way in-order processor configuration.

0 + icalu 1). SPECFP, however, obtains 3% ICALU for a 2-way in-order configuration (**2way vldu 0 + icalu 1**).

The combined benefit of ICALU and VLDU, which is shown in the last row, is normally higher than the sum of their individual benefits. That clearly shows the synergistic effect of adding in the same design both accelerators which is one key contribution of this chapter. Bzip2 in particular obtains 28% speedup, as shown in Figure 5.10. A high proportion, 35%, of μ-ops are fused as MOPs that results this high speedup in bzip2.

Table 5.1 provides a summary of speedups obtained from optimizations and ICALU and VLDU for both 2-way and 4-way processor configuration. The combined benefit of ICALU and VLDU, which is shown in the last row, is normally higher than the sum of their individual benefits.

	SPECINT		SPECFP	
	2-way	4-way	2-way	4-way
VLDU	2.5%	2%	4.5%	2.5%
ICALU	8%	10%	3%	5%
ICALU + VLDU	11.5%	10%	9.5%	9%

Table 5.1: Breakdown of Speedup

5.5.3 Comparison with an Out-Of-Order processor

A modern out-of-order processor provides benefit by exploiting ILP using out-of-order execution and wider instruction issue. Our co-designed in-order processor exploits the

ILP by reordering the μ-ops. It provides higher effective issue bandwidth by instruction fusion.

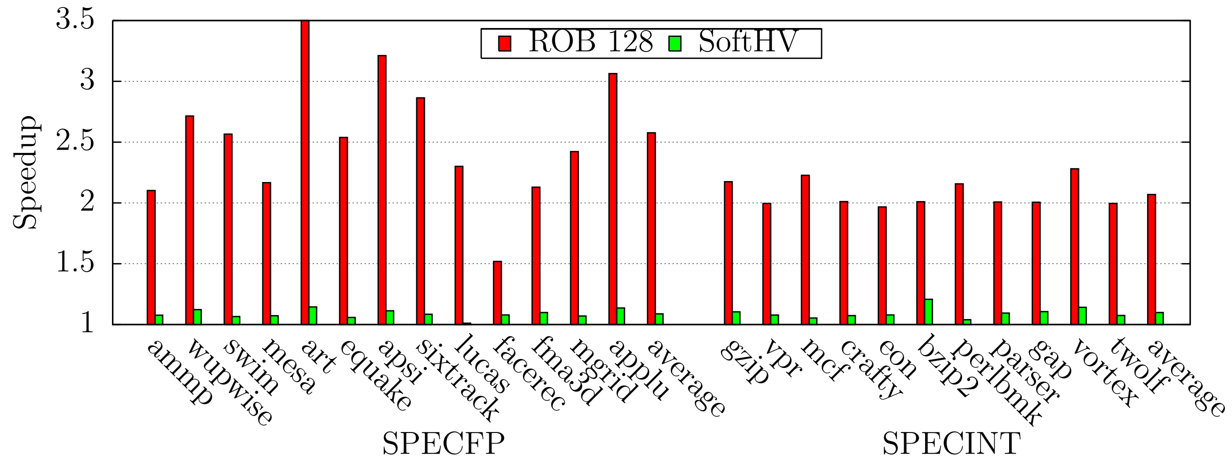


Figure 5.11: Comparison with out-of-order. *SoftHV* stands for 4-way in-order processor combined with code optimizations and FU accelerators. *ROB 128* stands for an out-of-order processor with instruction window of size 128. Speedups are normalized to 4-way co-designed in-order processor configuration with optimizations.

A comparison between the two design alternatives is worth a discussion. With a 128 entry ROB four-way out-of-order processor, which is described earlier in Table 3.3. Figure 5.11 shows the benefits of out-of-order processor. Note that, all the speedups reported are normalized to 4-way co-designed in-order processor with code optimizations.

Figure 5.11 suggests that the out-of-order execution with a large instruction window and memory disambiguation plays an important role in the performance of all the benchmarks. In the next chapter we will close this performance gap.

5.6 Related Work

Since in this chapter we have proposed general purpose application acceleration in a co-designed approach, most of the related work can be found in Chapter 2 and Chapter 4. The related work on Cd-VM can be found in Chapter 2. In Chapter 4 the related work on accelerators can be found. In this section, however, we choose to compare our proposal to the one that we proposed in previous chapter and to the one closest to our proposal.

In Chapter 4, we had proposed a PFU, which is a grid of functional units consisting of simple arithmetic units. A set of dependent or independent instructions could be mapped to the PFU. Moreover, an internal register file was proposed to hold the input operands. A ld-set and mv-set was proposed to bring data from the conventional register file to the internal register file. The fused instruction was generated by the dynamic binary

optimizer, same as in this chapter.

However, in this chapter we consider fusing dependent and independent instructions separately. VLDU executes a pair of independent load instructions, whereas ICALU executes a pair of dependent ALU instruction. Moreover, these fused instructions can read their input operands directly from the conventional register file.

Moreover, in this chapter we do not require a configuration cache in order to hold the control signals. The control signal corresponding to the fused pair of instructions reside together all along the pipeline.

The work closest to ours is that proposed by Hu et. al. [52]. They had proposed a complexity-effective out-of-order processor. In their work they use ICALU [78] to fuse a pair of dependent ALU instruction and execute in a single cycle. They also propose a co-designed VMM to find and fuse instruction pair within a superblock. However, they focus only on SPECINT benchmarks as it has abundance of dependent ALU instruction pair. They compared their proposal, which is a two-way out-of-order processor with ICALU, with a four-way out-of-order processor.

We, however, do not limit ourselves to SPECINT only. For SPECFP, we have showed vector loads provide major performance benefits. Moreover, we took a radical approach by moving to an in-order processor as the baseline. We have shown our proposed processor outperforms a small instruction window out-of-order processor.

Moreover, Hu et. al. [52] had proposed a co-designed out-of-order processor using ICALU. They send the head of a MOP in parallel to a normal ALU. However, our experiments have showed us that most of the times the ALU μ-op that is the head in the pair produces value only for the μ-op it is paired with. As a result, we do not send the head μ-ops into another ALU.

5.7 Conclusions

In this chapter, we have presented SoftHV, whose main novelty is the right combination of hardware and software to implement horizontal and vertical fusion of instructions. This helped in obtaining performance improvements on different types of workloads over a conventional in-order processor.

We have shown that SoftHV is implemented by means of a Cd-VM that reorders and optimizes a superblock. Pairs of dependent simple arithmetic/logic μ-ops are fused in a vertical fashion to be later executed in ICALU. Independent loads are fused in a horizontal fashion and are executed in a VLDU. These accelerators when combined together provide more performance benefit than when applied individually.

Overall SoftHV results in an interesting co-design approach that obtains an average

speedup of 9% for SPECFP and 10% for SPECINT over a conventional 4-way in-order processor. Hence, SoftHV provides an interesting design point for low-complexity high-performance processors for different types or workloads. Therefore, this can be a good alternative to small out-of-order processors for the low-end consumer electronics domain.

Chapter 6

A Power-efficient Co-designed Out-of-Order Processor

In the previous two chapters we had looked into instruction fusion in the context of Co-designed Processors. In this chapter, we cut down the complexity of out-of-order logic using FIFO based issue logic. Furthermore, we have co-designed the commit logic in order to bulk commit atomic superblocks efficiently.

First, we propose an enhanced steering heuristic and an early release mechanism to increase the performance of a FIFO based out-of-order processor. We obtain performance improvement of nearly 25% and 70% for a four FIFO and for a two FIFO configurations, respectively. We also show that our proposed steering heuristic based processor consumes 10% less energy than the previously proposed steering heuristic.

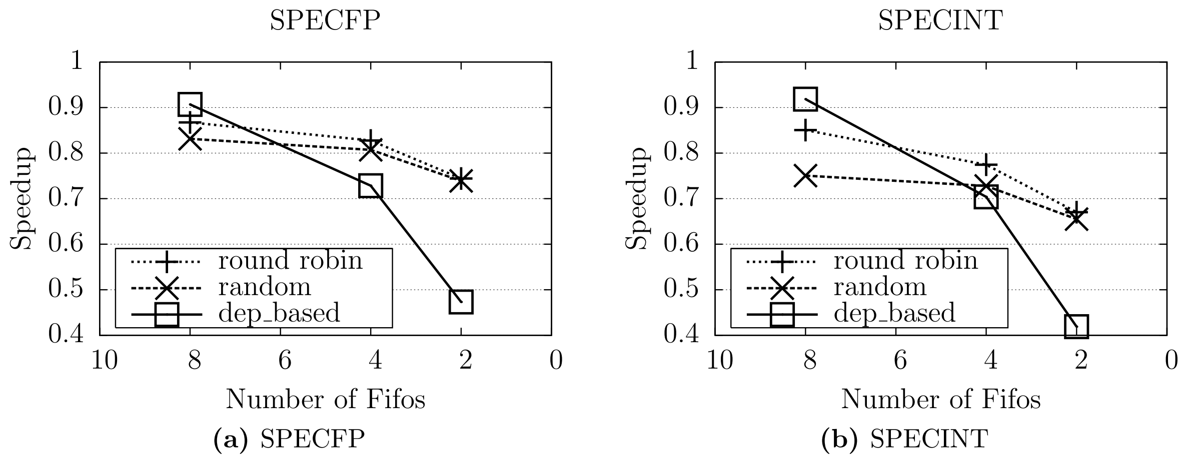
Moreover, we also show by choosing the order in which ready μ-ops from the head of the FIFO are issued also has a significant impact in performance. For instance with eight FIFOs and issuing the oldest μ-op the performance gap between CAM based issue logic and FIFO based issued logic is nearly closed.

Furthermore, we propose a bulk commit logic that is able to commit the program state at the granularity of the superblock. This enables us to get rid of the Reorder Buffer (ROB) entirely. Instead to maintain the correct program state, we propose a four/eight entry Superblock Ordering Buffer (SOB). We also propose the per superblock Register Rename Table (SRRT) that holds the register state pertaining to the superblock. Moreover, when compared to ROB based processor executing normal μ-ops we achieve a 40% reduction in power and 25% reduction in energy with four FIFOs.

6.1 Introduction

In previous chapter we have seen how fusion and good code scheduling techniques provides important performance improvements over a conventional in-order processor that puts the design in a level closed to a small out-of-order design. However, in order to achieve extremely good single-thread performance we still have to consider out-of-order processors.

Conventional out-of-order processors uses a CAM based issue logic that enables out-of-order execution of μ-ops, using a wake-up and select logic. This mechanism helps in exploiting the ILP, but it comes at the cost of higher complexity [76] and power dissipation [46].



(a) SPECFP **(b)** SPECINT

Figure 6.1: Performance of Steering Heuristics. The sharp decline in Performance observed for different Steering Heuristics when number of FIFOs are decreased. Speedup normalized to CAM based issue logic.

Some researches have considered alternatives to CAM-based designs including the use of FIFOs. A FIFO based issue logic, helps in reducing both the complexity and the power. Palacharla et al. [76] proposed a dependence-based steering heuristic that steers each μ-op to a chosen FIFO at the dispatch stage. As the number of FIFOs are decreased the performance of the dependence-based heuristic declines sharply, for both integer and floating-point benchmarks, as shown in Figure 6.1. A similar observation was also made by Canal et. al. [18]. In fact a policy as simple as round-robin or random performs better with fewer number of FIFOs compared to dependence-based heuristic.

In this chapter we consider a HW/SW co-designed out-of-order processor that uses multiple, but a small number of FIFOs, in order to achieve high performance with a low-complexity design. Our steering heuristic is based upon the dependence-based heuristic, and we enhance the heuristic by analyzing the stalls at the dispatch stage. This modified steering heuristic reduces the stalls at dispatch leading to significant performance benefit.

On the other hand, modern out-of-order processors, for example Alpha 21264 [58], speculate on the loads to hit in the L1-Cache. In order to guarantee back-to-back execution, dependent μ-ops of a load are issued speculatively. Issue Queue entries of all the issued μ-ops are held for a certain number of cycles[1]. In case of a miss-speculation, the μ-ops from the execution pipeline are squashed and issue queue entry is re-validated.

This holding of issue-queue entries for a few number of cycles adds pressure on the issue queue. For FIFO based issue queues this has an even more dramatic impact on the performance, especially for the dependence-based steering heuristic.

One could argue that this pressure could be reduced by increasing the size of Issue Queues. For a CAM based Issue Queue, though this might be true, but it comes at an additional cost both in complexity and power. For a FIFO based Issue logic, on the other hand, increasing the size of a FIFO, merely adds entries to its tail.

The performance of dependence-based scheme such as the one proposed in this chapter depends on the availability of entries at the head of a FIFO, as μ-ops are issued from the head. Therefore, in addition to the new heuristics for steering we propose an early release mechanism, that releases issue queue entries, at the issue stage, for all the μ-ops issued in that cycle; given its safe to do so. The proposed early release mechanism is not just applicable to FIFO based issue logic, but can be applied to CAM based Issue logic, as it is.

Finally, the ability to execute out-of-order requires ROB-like structures to retain the original program order to allow commit in order. However, in a design such as the one considered in this chapter, where a HW/SW co-designed out-of-order processor is used that uses a Virtual machine monitor (VMM) as described in Chapter 3 that builds atomic superblocks, we must also enforce atomic commit of the superblocks.

The atomic commit constraint of the superblock implies that if all the μ-ops of the superblock have not executed then the μ-ops need to wait in the ROB to commit. This puts pressure on the ROB and related structures and leads to stalls. In order to mitigate these problems in this chapter we have also proposed a ROB-less bulk commit logic. Two structures, namely a Superblock Ordering Buffer (SOB) and Superblock Register Rename Tables (SRRTs) are proposed that together maintain the correct program state. Each superblock is allocated a SRRT and an entry in a Superblock Ordering Buffer (SOB). The entry at the head of SOB is considered for commit at every cycle. A SOB entry contains various fields; that not only indicate whether a superblock is ready to commit, but also locates the program state associated with the superblock.

Overall in this chapter we have considered a HW/SW co-designed out-of-order processor based on FIFOs and with new steering, issue and bulk commit improvements. All these improvements results in a design that is within 20% of performance of a conventional out-of-order design, with a reduction in energy by 25% for a four FIFO configuration.

[1] Two cycle in our case.

The key contributions of this chapter are as follows:

- Superblock Ordering Buffer (SOB) is proposed in Section 6.4.2, that commits the program state in the original program order. As a result of the SOB and related structures, the need of conventional Reorder Buffer (ROB) is eliminated.
- Per Superblock Register Rename Table (SRRT) is proposed in Section 6.4.2, that holds the register of the corresponding superblock and is committed atomically.
- Enhanced dependence-based steering logic is proposed in Section 6.3.2, which reduces various stalls at dispatch stage due to the unavailability of empty FIFOs. This provides significant performance benefit, by increasing the decoder throughput.
- Early release logic is proposed in Section 6.3.3, that releases few issue queue entries at issue time. This reduces the pressure on FIFO based issue queues and provides major improvement in performance in a FIFO constrained scenario.
- We have shown the effect of *start policy* in Section 6.6.1, which determines the order of issuing the ready μ-ops.

The rest of the chapter is structured as follows. First, the overview of the proposed microarchitecture is given in Section 6.2. In Section 6.3 the steering logic and the early release logic are discussed in greater detail. Next, the co-designed commit logic is proposed in Section 6.4. The performance and the power results are discussed in Section 6.6. The related work is discussed in Section 6.7 and the conclusions are drawn in Section 6.8.

6.2 Overview of the Proposed Microarchitecture

In this section we provide an overview of our proposed co-designed out-of-order microarchitecture. In Figure 6.2 we provide an overview of our proposed microarchitecture. The gray blocks were added on top of the baseline microarchitecture. Moreover, we get rid of the ROB, which is not shown in the figure.

As shown in the figure the FRRT and the BRRT are maintained to hold the speculative and the committed register mappings. Additionally, a SOB and multiple SRRTs are added in order to support the bulk commit mechanism. The issue logic consists of multiple FIFOs, from which μ-ops could be steered to any FU.

Our FIFO based out-of-order issue logic cuts down both the complexity and power consumption; yet retains the benefits of out-of-order issue. Such a middle ground solution is more power-efficient and aligns with the co-designed paradigm.

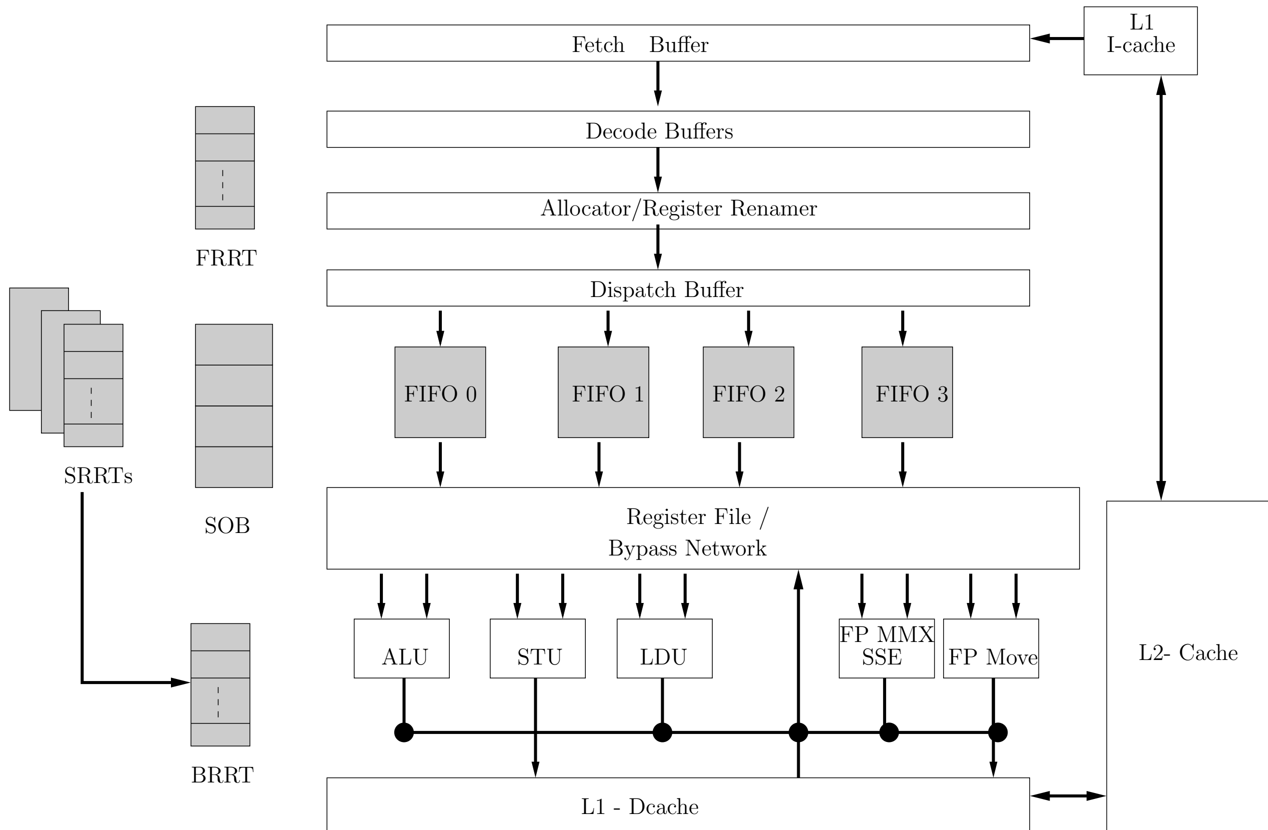


Figure 6.2: Block diagram of the Microarchitecture of SOB based processor.

However, the existing FIFO based dispatch steering heuristics do not perform well for low number of FIFOs. By making few key observations on dispatch stall behavior we propose enhanced steering heuristics that outperform the state-of-the-art dependence-based steering heuristic.

Moreover, we also propose an early-release mechanism that helps in releasing the FIFO heads sooner resulting in fewer stalls at dispatch. This early release mechanism is based on the observation that μ-ops are held in the issue queues for two cycles after they are issued. This is required in order to take corrective action in case of a load-hit misspeculation. However, holding issue queue entries for all the μ-ops is conservative. We exploit this fact and release the issue queue entries for those μ-ops for whom it is safe to do so.

The use of out-of-order issue logic requires a need for an in-order commit mechanism in the processor back-end. However, the conventional ROB based in-order commit mechanism is more suitable for conventional out-of-order processors. In conventional out-of-order processors, such as X86 based, the commit is at the granularity of an X86 instruction and not μ-ops.

In our proposed co-designed processors we execute atomic superblocks, which requires a bulk commit mechanism, as explained earlier in Chapter 2. In a bulk commit mechanism all the μ-ops corresponding to a superblock is committed together in a bulk. This would require the μ-ops be held in the ROB. However, holding μ-ops of a superblock in ROB has several disadvantages, which were discussed earlier in Chapter 4, in Section 4.3.1.

Our bulk commit mechanism is based on the key observation that it is the change in the program state– the register state and the memory state – is to be held, rather than the μ-ops. The second observation is that since the commit is at the granularity of the superblock, it is more natural to also maintain the state at that granularity. As a consequence of these observations, we get rid of the ROB; rather, we propose a SOB.

Each entry of the SOB is allocated to exactly one superblock, when the first μ-op of the superblock is being renamed. Only when all the μ-ops of the superblock have successfully executed, the superblock is ready to commit. The SOB entry at the head of the SOB is checked every cycle to determine whether the corresponding superblock is ready to commit. If it is the case then the program state is committed by making it architecturally visible.

As mentioned above, the program state consists of the register state and memory state. The register state is held in the SRRT, which holds the mappings from architected registers to the physical registers. Moreover, only the mappings of the live-outs of the superblock are held in the SRRTs. Each SRRT belongs to exactly one superblock, and the number of SRRTs is equal to the entries in the SOB[2]. An entry in the SRRT is allocated when a live-out of a superblock is being renamed.

When all the μ-ops of the superblock have successfully executed and the superblock's SOB entry is at the head then the superblock's program state is committed. The register state of the superblock which is held in the SRRT is copied to the BRRT. Since register state contain only valid mappings of live-outs they are committed. The memory state is held in the Gated Store Buffers [95], which is committed to the memory hierarchy in bulk.

6.3 Out-of-Order Logic

We use a FIFO based out-of-order logic that has been shown as a complexity-effective [76] and a power-efficient design. A steering policy selects a FIFO for the μ-op to be steered to at the dispatch stage, as shown in the Figure 6.3 .

μ-ops are issued from the head of a FIFO. Multiple μ-ops could be issued from the same FIFO if the FIFO supports that. For instance, if each FIFO has built-in dual-issue ability,

[2] We found four or eight entries to be reasonable. This will be shown experimentally later.

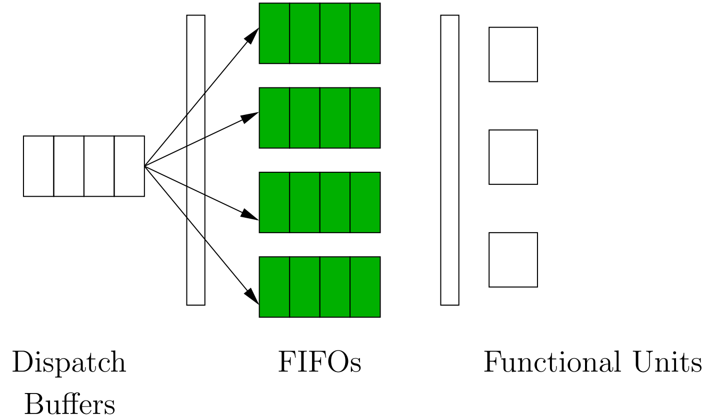


Figure 6.3: A FIFO based OoO Logic. μ-ops are steered to FIFO at the dispatch stage. μ-ops can only issue from the head of a FIFO.

then the second μ-op immediately after the head will be issued if both the head and it itself is ready to issue.

μ-ops from different FIFOs could issue in parallel as long as dependencies are respected. Every cycle μ-Ops at the head check if their operands are ready in a small table. This table stores just one bit per physical register indicating it is available.

Another way of implementing this table could be by using per physical register counters that are set to the latency of producing μ-op when it is issued. Every cycle the counter is decremented; dependent μ-ops are issued when counters corresponding to its source operands are zero. This mechanism is similar to the one proposed in ARM Cortex A8 [94].

In this section, first we will describe the baseline dependence-based steering heuristic proposed by Palacharla et al. [76]. Next, we will describe the reasons behind the poor performance of the heuristic. Specifically, we will provide the breakdown of conditions for stall at the dispatch. Next we will propose our enhanced steering heuristic that reduces these stalls and obtains a better performance. We will describe the logic required to enhance the steering heuristic. Next, we will describe the Early Release mechanism, followed by a brief description of the *start policy*. In the end, we will describe the Memory Disambiguation logic we used for our co-designed environment.

6.3.1 Dependence-based Steering Heuristic

Since our proposed steering logic depends partially on the one proposed by Palacharla et al [76], we first describe Palacharla's approach and later point out its drawbacks.

Let I be the μ-op that is ready to be dispatched. Depending upon the availability of I's

operands, the steering decisions made are:

- If all the operands of I are ready, then steer I to an empty FIFO.
- If only one source operand of I is available, then steer I to a FIFO whose tail produces the required operand. If no such FIFO found steer I to an empty FIFO.
- If both the source operands of I is unavailable, then steer I to a FIFO whose tail is the operand producer of either of the operands, giving priority to left source operand.

If the desired FIFO is full or an empty FIFO is not available then dispatch is stalled. The steering logic required in the dispatch stage consists of a SRC_FIFO table. This table is indexed by physical register, and contains the identity of the FIFO buffer that contains the μ-op that produces the architected register value.

Figure 6.1 shows the performance of the above mentioned dependence-based dispatch policy when implemented in our co-designed processor. The performance degrades sharply as the number of FIFOs are decreased; an observation also made by Canal et al. [18].

As the number of FIFOs are decreased, stalls at dispatch stage increases, reducing the throughput of the dispatch stage. This reduction impacts the overall performance of the processor. We quantify these stalls at dispatch stage and overcome them by proposing enhanced steering heuristics.

6.3.2 Enhanced Steering Heuristic

As observed above, a major drawback of the dependence-based scheme is a high frequency of stalls incurred at the dispatch stage, as the number of FIFOs are halved. The three conditions that causes the dispatch to stall for the dependence-based scheme are:

- Rdy_no: Empty FIFO is unavailable for a μ-op whose source operands are ready at dispatch.
- Tail_no: Empty FIFO is unavailable for a μ-op, neither of whose source operands are ready nor are any of the producers a tail of a FIFO.
- Tail_FIFO: FIFO is available, but is full, for a μ-op whose producer is the tail of the FIFO.

Figure 6.4 illustrates the three above mentioned stalling conditions. In Figure 6.4a μ-op 4, which is **ready**, cannot be dispatched because there is no empty FIFO available. In

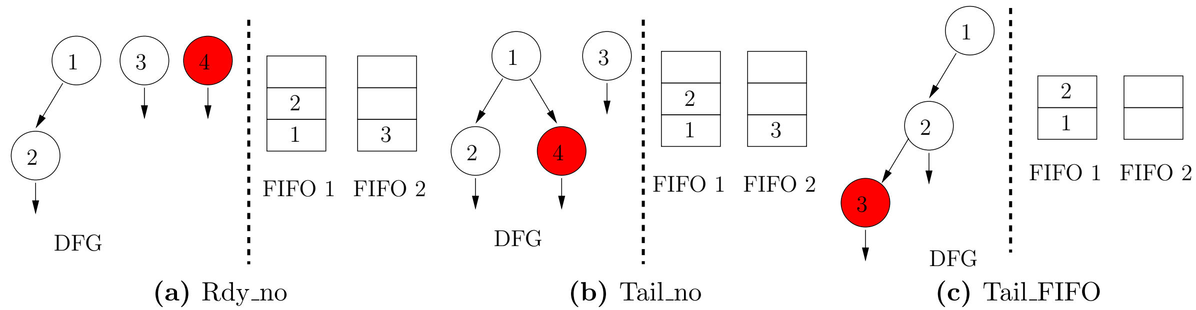


Figure 6.4: Dispatch Stalling Conditions. Each figure has the dataflow graph on the left and the contents of the FIFOs on the right of the dotted line. The μ-op in red cannot be dispatched.

Figure 6.4b μ-op 4, which is **not ready**, cannot be dispatched because neither is there an empty FIFO available nor is its producer, μ-op 1, tail of any FIFO. In Figure 6.4c μ-op 3, which is **not ready**, cannot be dispatched because FIFO 1, which holds μ-op 3's producer as a tail, is full.

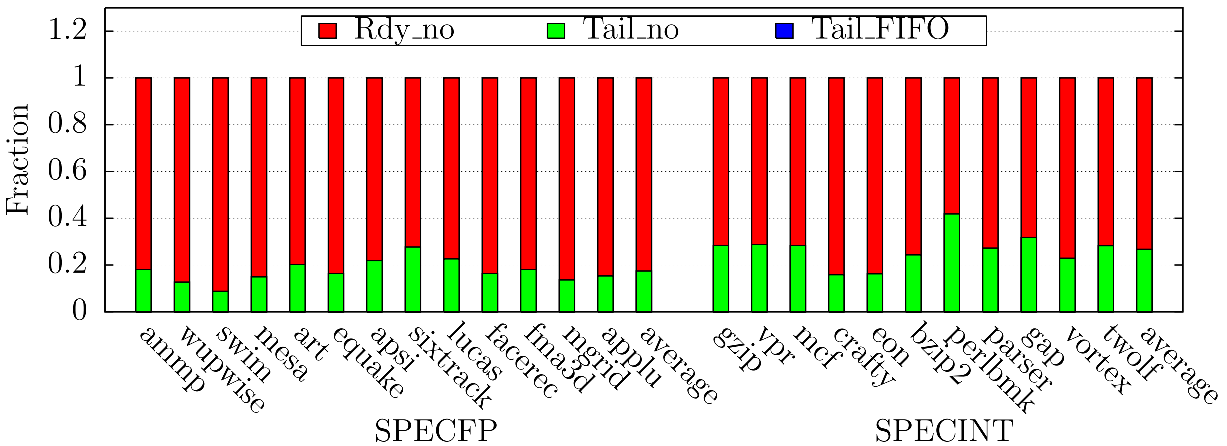


Figure 6.5: Dispatch Stall Condition Distribution for a four FIFO configuration.

Figure 6.5 provides distribution of the these three stalling conditions. Nearly 80% of stalls are due to Rdy_no, while Tail_FIFO hardly causes any stall. Tail_FIFO is hardly causing any stall because the current size of FIFOs, which is eight, is sufficient. For the stall Tail_FIFO to occur with eight FIFOs a long linear chain[3] of dependent μ-ops have to be present. However, in SPEC benchmark such a long linear chain of dependent μ-ops is hard to find. Moreover, even if there is such a chain, usually μ-ops from independent chains are present, which would result in other kind of stalls.

[3]for example 8

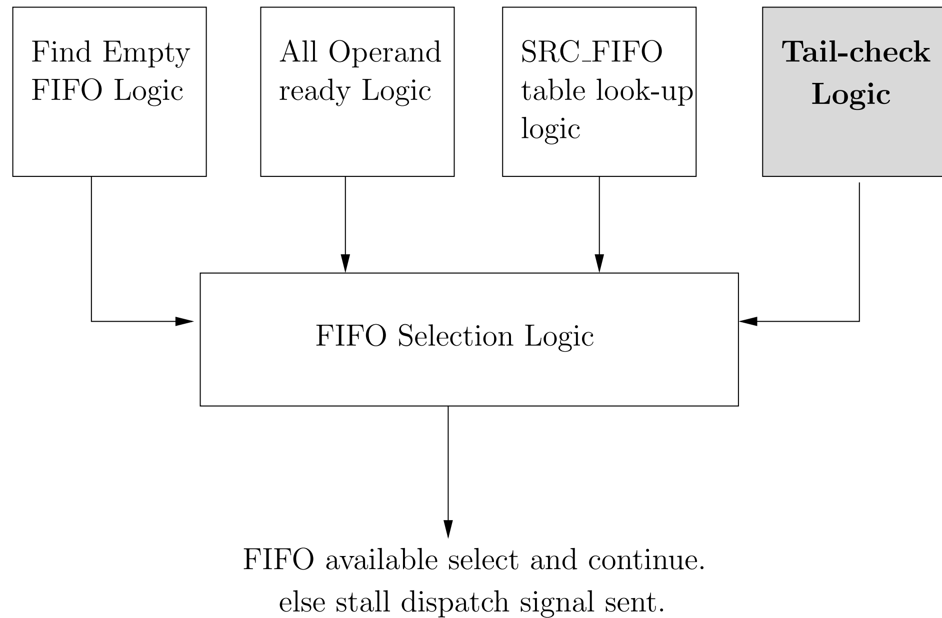


Figure 6.6: Steering Logic shows an additional Tail-check logic compared to the original dependence-based steering heuristic.

We exploit this observation by modifying the steering logic. Our modified steering logic builds upon the dependence-based steering logic. It, however, reduces the stalls due to Rdy_no by steering a μ-op, whose operands are ready, to a FIFO whose tail is ready. The FIFO whose tail is ready implies that all μ-ops ahead of it in the FIFO must be ready, as the μ-ops are steered based on their dependencies.

Figure 6.6 shows the enhanced steering logic that requires an additional tail-check logic. Shaded blocks are those that have been added. The Tail-check logic basically uses the Register ready table, which is also checked by the head of FIFO at the issue stage, as illustrated in Figure 6.7.

The Empty FIFO Logic is used to determine whether a FIFO is empty or not. Each FIFO maintains a counter to indicate the number of μ-ops that it holds. Every cycle at dispatch these counters are read to determine whether a FIFO is empty or not.

The SRC_FIFO table is used to determine which FIFO holds the producer of a physical register. This again is implemented using a SRAM structure and has number of rows equal to the number of physical registers, which is 128.

All Operand ready logic is essentially a score-boarding mechanism. We implement this using a Physical Register Ready Table. This table contains a bit indicating whether the Physical Register associated with the table entry is ready or not as shown in Figure 6.7. For each operand at dispatch a check is made to the Physical Register Ready Table to

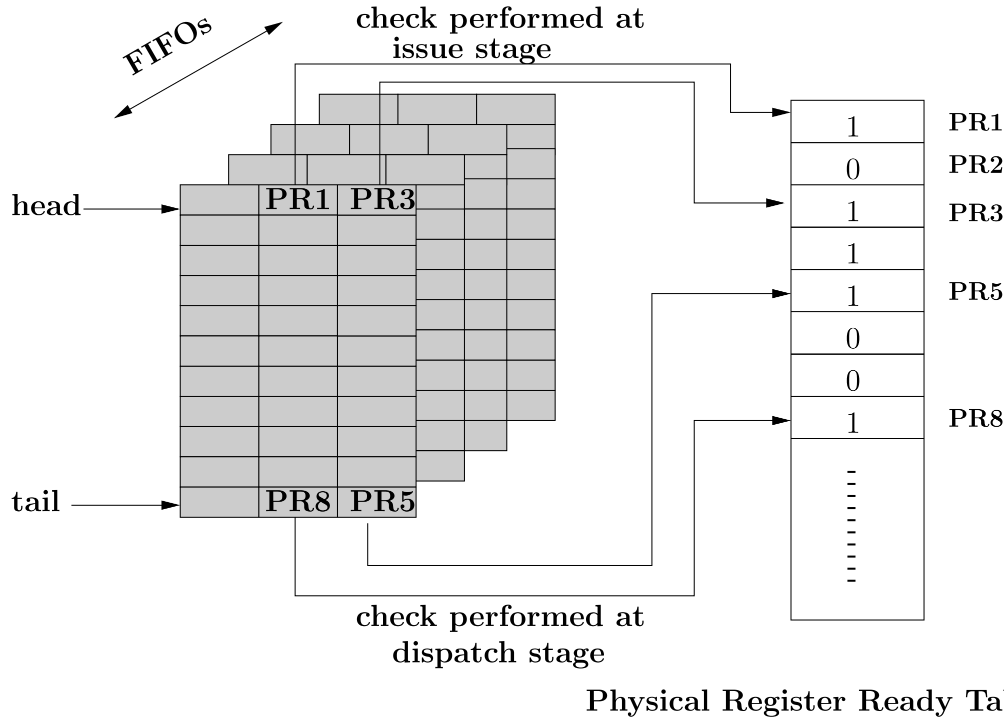


Figure 6.7: Illustration of head and tail check logic shows that the Physical Register Ready Table is accessed by the head and the tail at issue and dispatch respectively.

determine whether the corresponding physical register is ready.

The Physical Register Ready Table is implemented using a SRAM structure. The Physical Register Ready table has same rows as the SRC_FIFO table, which is 128. However, the size of a column is smaller than SRC_FIFO as it contains just a single bit. This implies the access delay to this table is smaller than that for the SRC_FIFO table.

As a consequence of the proposed enhancements to the steering logic, the likelihood of multiple ready μ-ops in a FIFO increases. This can be further exploited by increasing the issue width per FIFO. The μ-op immediately after the head will only be issue if the head is ready to issue. The dependence-based scheme is insensitive to the increase in per FIFO issue width. We validate this intuition in Section 6.6.

We further try to decrease the stalls at dispatch by reducing stalls due to Tail_no. We reuse the Tail-check logic by steering a μ-op, that encounters Tail_no, to a FIFO whose tail is ready. We use the same intuition, as stated above, that a FIFO whose tail is empty is as good as an empty FIFO. In other words, we would be better-off steering such a μ-op to a FIFO whose tail is ready than waiting for a FIFO to get ready. The impact of this enhancement is also validated in Section 6.6.

Since we reuse the Tail-check logic we do not add any additional logic. The all operand logic and the SRC_FIFO table look logic is always checked for each μ-op. These are the logic which determines whether a μ-op will incur a stall. Hence, no additional check is made.

6.3.3 Early Release

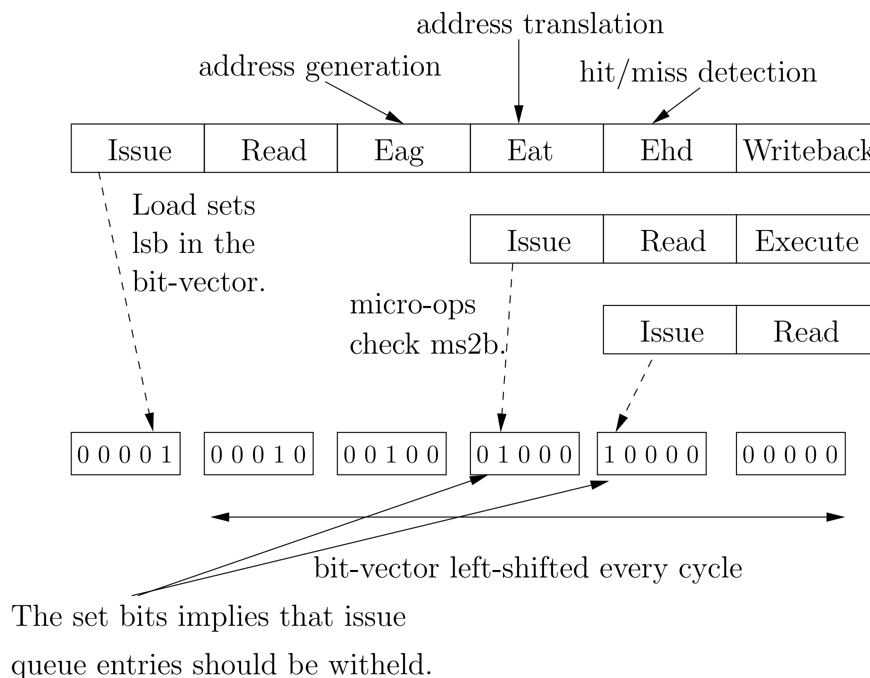


Figure 6.8: Illustration of early release. The lsb of the bit-vector is set when a load issues, the two most significant bits (ms2b) are checked by μ-ops when they issue to determine whether they could release issue queue entries or not.

We propose an early release mechanism, that releases issue queue entries, at the issue stage, for all the μ-ops issued in that cycle; given its safe to do so. The proposed early release mechanism is not just applicable to FIFO based issue logic, but can be applied to CAM based Issue logic, as it is.

Our proposal introduces two[4] bit-vectors that keep track of whether a load was issued in a given cycle for a small history. The length of these bitvectors is determined by the latency of the load μ-op, which is five in our case. At any given cycle, if a load issues the least significant bit is set. Both the bit-vectors are left-shifted, at every cycle, by one.

Figure 6.8 provides an illustration of the early release scheme. To determine whether a

[4] Since there are two Load Units.

load was issued X cycles back, the bit at position $X + 1$[5] is checked. Since the bit-vectors are five bits wide, the set bit at most significant or the second most significant position implies a load was issued four or three cycles back, respectively. This information is sufficient to determine whether the issue queue entries, of the μ-ops issuing in the current cycle, could be released.

Under branch miss-prediction the bit-vector is not cleared that might cause unnecessary occupation of FIFO entries. This, however, has a negligible impact in performance and has been validated experimentally. This check is done in parallel at issue along with other check and the logic involved is very trivial.

6.3.4 FIFO start policy

Every cycle μ-ops are selected from the head of the FIFOs to be issued. Only those μ-ops that are ready are considered for issue. However, only a subset of the ready μ-ops could be issued, because of limited resource constraints, such as issue width, functional units, write ports etc.

Moreover, the μ-ops that are ready are selected for issue in an order, which we refer to as the *start policy*. For instance, the order could be pre-determined such that μ-ops in the head of FIFO_0 gets priority over the μ-ops in the head of FIFO_1. We call such a start policy where order is fixed as *normal* policy.

Certainly such a policy is lot simpler to implement. However, older μ-ops which were dispatched to a FIFO with lower priority could issue later resulting in delaying the critical path. In order to overcome this potential performance issue we use another *start policy* that guides the selection based on the program order of μ-ops. We refer to this policy as *oldest first* policy, in which the issue order is determined every cycle by prioritizing older μ-ops over younger ones.

Such a policy could also be applied to multiple CAM based Issue Queues. The selection logic is implemented using priority encoder and is described in [76]. In Section 6.6.1 we have studied the impact of *start policy*, and found out that *oldest first* narrows the gap between FIFOs and CAM based logic further.

6.3.5 Memory Disambiguation Logic

Loads are issued speculatively, in a modern out-of-order processor, by issuing them before unresolved waiting stores. Miss-speculation is detected in the hazard detection stage, by searching in the Store Queue (a CAM based structure) for aliasing with older stores. The miss-speculated loads is stored into a Load Queue and a CAM based logic wakes up the

[5] With the least significant bit position being zero.

load when aliasing stores issue. Woken up loads are re-executed causing further activity, leading to power dissipation.

Our proposed design, however, does not have support for hardware memory disambiguation. Instead the VMM re-orders the memory μ-ops. Only those memory μ-ops that can be determined statically to not alias, within a superblocks, are reordered. Our proposed processor does not have support for hardware memory disambiguation. The order in which memory μ-ops are issued is explained below.

The stores are issued out-of-order, while the loads wait for all the older stores to issue. However, a bunch of loads, between two stores, can be issued out-of-order. This drastically simplifies the logic by eliminating multiple CAM wake-ups, miss-speculation and associated replay logic.

A single issued-bit is appended to each gated store buffer entry indicating, whether the store has issued. At the rename stage, a load locates the tail index of the gated store buffer. This index is appended along with other control signals of the load and moves along the pipeline. As a load reaches the head of a FIFO, the issued-bit of all the store buffer entries from head to the stored tail index is scanned to check whether all the older stores have issued. This check is done in parallel to the usual look-up into the table that indicates whether the Physical Register has been produced.

6.4 Co-designing the Commit Logic

As a consequence of out-of-order execution, reorder logic is required at the back-end in order to maintain a correct program state. Moreover, the atomic property of the superblock requires the program state be committed at bulk. The program state consists of the register state and the memory state.

In modern out-of-order processors front-end state is maintained by the Front-end Register Rename Table (FRRT), while the Back-end (committed) state is maintained by the Back-end Register Rename Table (BRRT), similar to the Netburst microarchitecture [51]. However, since the superblocks are atomic, the BRRT is updated only when all the μ-ops of the superblock have successfully written to the Physical Register File. Similarly, the memory state is committed to the D-cache when all the μ-ops of the superblock have completely executed.

In Chapter 4, Section 4.3.1 we had discussed a mechanism to implement bulk commit of atomic superblocks. We has also mentioned a couple of problems associated with bulk commit. In this section we will discuss yet another problem associated with bulk commit of atomic superblocks in the context of an out-of-order processor. In order to tackle all the bulk commit problems we will propose a novel bulk commit mechanism by co-designing the commit logic.

6.4.1 Bulk Commit Problem 3

This problem is related to the problem mentioned in Section 4.3.1. By using the SpecRRT as described in Section 4.3.2 we were able to commit μ-ops of the same superblock that are older to the μ-op that missed in L2-cache. However, the μ-ops that are independent and younger to the μ-op that missed in the L2-cache will still have to wait in the ROB.

For instance, in Figure 4.9 the μ-op 3 and μ-ops younger to it cannot retire. As a result, such a scenario will eventually lead to stall at the frontend, which can limit the ability to expose the ILP of an application.

6.4.2 A ROB-free Bulk Commit mechanism

In order to tackle all the bulk commit related problem described above we take an entirely different approach. We use the observation that the change in program state is the change in the register state and the change in the memory state. Hence, instead of holding μ-ops we hold the program state.

Moreover, since the program order has to be maintained at the granularity of the superblock, we precisely do that by introducing a ROB-free reorder logic. In our proposal, we get rid of the ROB entirely and introduce a new structure the Superblock Ordering Buffer (SOB) to maintain program order at the granularity of the superblock. A special commit operation updates the program state, both the register and the memory state.

6.4.3 Register State

In order to commit the register state atomically, we propose the per superblock register rename table (SRRT). The SRRT, like FRRT or BRRT, holds mappings from architected register to physical registers. However, unlike FRRT or BRRT, it holds mappings of only those architected registers that are live-outs of the superblock[6], as illustrated by the example in Figure 6.9.

A SRRT is assigned to a superblock when the head μ-op is being renamed. The source operands mappings are read from the FRRT and the destination operand is updated in the FRRT, in a conventional manner. However, if the μ-op being renamed is a live-out, then the SRRT, corresponding to the superblock, is updated. When all the μ-ops, of the superblock, have executed the register state is committed, by copying only the valid mappings from the SRRT to the BRRT; and the SRRT is made available.

Note that the primary difference between SRRT and the SpecRRT, as proposed in Section 4.3.2, is that the SRRT is per superblock, whereas there is only a single SpecRRT. More-

[6] Live-outs of the superblocks are marked by the VMM.

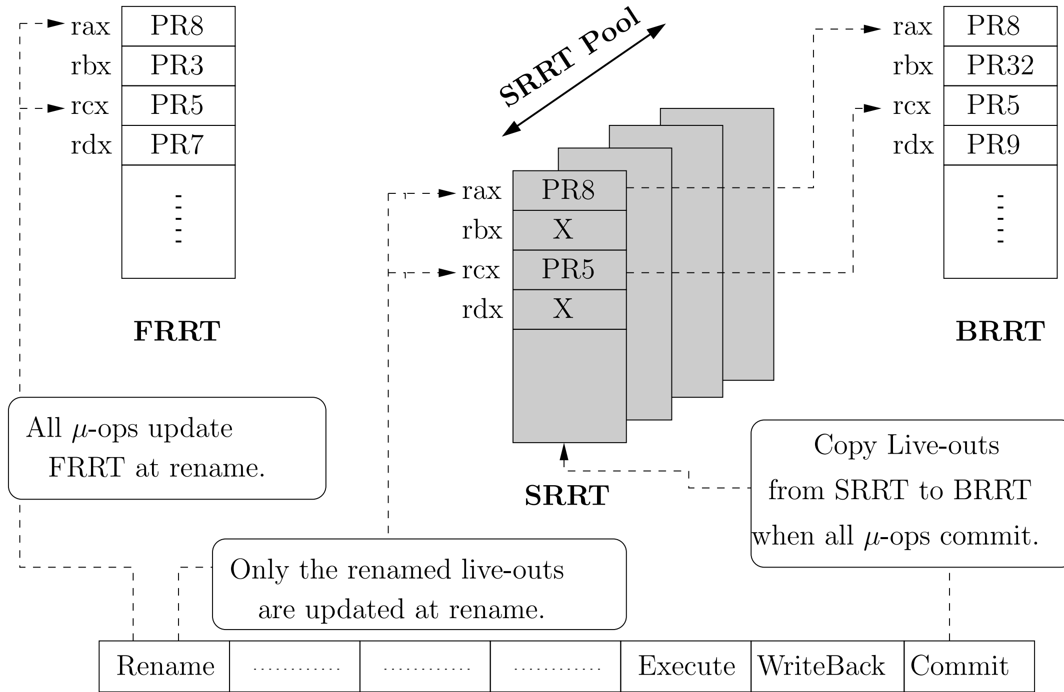


Figure 6.9: Register State Rename and Commit. rax and rcx are the live-outs of the current superblocks. Hence, rbx and rdx entries in the SRRT are invalid. As a result at commit only rax and rcx are updated in the BRRT. SRRT has four read and four write ports same as BRRT.

over, SRRT is allocated to the superblock when its μ-ops are renamed. The SpecRRT, however, is updated when the μ-ops of the superblock are being retired speculatively. Moreover, the contents from SpecRRT is copied to the BRRT when the tail of the superblock commits, whereas the contents from SRRT is copied when the last μ-op of the superblock to complete execution commits.

We have assumed four or eight SRRTs in order to support four or eight superblocks. We have assumed each SRRT has four read and four write ports.

Circuit Details of SRRT

The circuit level description of SRRT holds here for SpecRRT, which was described in Section 4.3.2 as well. SRRT like FRRT can be designed either using a RAM or a CAM based structure. We, however, use a RAM based structure as it is more scalable [76]. In a typical Register Rename Table a RAM cell consists of a shift register cell in order to shadow the mapping. However, the SRRT unlike FRRT does not need any shift register cell.

Moreover, unlike conventional rename tables, the SRRT is not read at the rename stage.

Its only written at the rename stage by the μ-ops that are live-outs of the superblock. The read ports are accessed only at the commit stage when the valid mappings from the SRRT are copied to the BRRT.

Total number of ports in a conventional FRRT for a four-way superscalar processor is twelve[7]. Whereas, six read and two write ports are sufficient for a SRRT. Hence if a superblock has more live-outs than six, the commit is split into multiple cycles. This has no consequence on the performance as commit is not in the critical path. Similarly, no more than two live-outs could be renamed in a given cycle. Our experiments have shown that the average number of live-outs per superblocks are nearly 8 and 4 for SPECFP and SPECINT, respectively.

The delay of a Rename Table is given by $T_{decode} + T_{wordline} + T_{bitline} + T_{senseamp}$ [76]. The T_{decode}, and $T_{bitline}$ depend upon the total number of ports and the number of entries. As shown above we reduce the number of ports to eight, while the number of entries are still the same[8]. This is in turn reduces these delays and the corresponding power.

Furthermore, the $T_{wordline}$ depends on the number of shift-register cells, the number of ports and the width of each entry. SRRTs do not need any shift register cells, has fewer ports, and the width is same[9]; and hence has a smaller delay. As a result, the overall delay of SRRT is lower than that of FRRT, and it does not fall into the critical path. However, as there are four/eight SRRTs, additional power is dissipated, which is quantified in Section 6.6.3.

6.4.4 Superblock Ordering Buffer

We propose Superblock Ordering Buffer (SOB) in order to commit the superblock register and memory state in program order. Since SOB is a circular buffer, a superblock is committed only when the entry corresponding to it in the SOB is at the head of SOB. For a four-way out-of-order processor a SOB could have four or eight entries, many-fold smaller than a conventional ROB for a four-way superscalar processor.

Each SOB entry consists of six fields as shown in Figure 6.10. The first field (mio_cnt) indicates the number of μ-ops in a superblock. The second field (exe_cnt) indicates the number of μ-ops that have successfully executed. The third field (p_srrt) is a pointer to the per superblock register rename table (SRRT). The fourth field (tail_rnm) is a bit indicating whether the tail μ-op of the superblock has been renamed. The fifth (st_bgn) and the sixth (st_end) fields indicate the beginning and the end index of the gated store buffer, respectively.

[7]Eight read ports and four write ports assuming two operand μ-ops.
[8]Number of Architected Registers.
[9]Size depends upon number of Physical Registers.

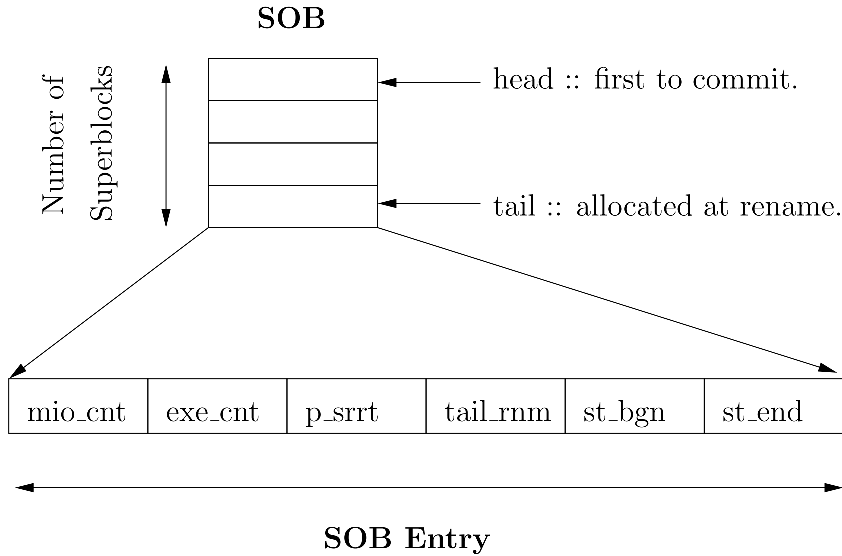


Figure 6.10: Superblock Ordering Buffer (SOB). Various fields of SOB are shown in the figure.

As the head μ-op is renamed, a SOB entry and a SRRT is allocated to the superblock. The p_srrt field of the SOB entry is made to point to the SRRT. As other μ-ops are renamed mio_cnt is incremented, and the tail_rnm is set when the tail is renamed. The st_bgn is updated when the first store is at rename, while st_end is updated when the last store is updated.

As a μ-op reaches the commit stage, the exe_cnt field of the SOB entry is incremented. The head of SOB is considered for commit when the exe_cnt equals the mio_cnt, and the tail_rnm (bit) field is set[10]. The register state held by SRRT is updated to BRRT; and the gated store buffer commits the memory state.

We have modeled SOB using a RAM array, and observed that the delay and power dissipated is lower than the ROB. Since the SOB consists of only eight or four entries, the $T_{bitline}$ and the T_{decode} are smaller[11]. The $T_{wordline}$ depends upon number of ports and the entry width. Since both the number of ports and the entry size[12] are equal for a SOB and a ROB, the $T_{wordline}$ delay is same. Hence, the overall delay and the power dissipated by SOB is lower than that by a ROB.

We have assumed a SOB of four or eight entries in order to support four or eight supperblocks. SOB has four read and four write ports.

[10] This implies the μ-ops have renamed and hence executed.
[11] Since the number of ports are eight in both SOB and ROB.
[12] Each SOB entry is four-five bytes wide, similar to a ROB entry

6.4.5 Physical Register Recycling

This Physical Register Recycling is applicable to all the bulk commit mechanisms proposed above. Physical registers are held in the conventional physical register file. The VMM marks the *non live-out* architected registers and finds the number of consumers for each one of them. This information is used to update the counter associated with the corresponding Physical Register.

As the consumer μ-ops execute the counter, associated with the source physical registers, is decremented. The Physical Registers whose counters decrement to zero are freed by the conventional Register Recycling mechanism [89].

On the other hand, the *live-out* architected registers of a superblock are handled similar to the conventional out-of-order processor. Even after the superblock is committed the counters associated with these physical register are not equal to zero. This is because they hold the program register state and at least the BRRT holds reference to them. Only when another superblock with same *live-out* architected register is committed the physical register is freed.

6.4.6 Memory State

We use gated store buffers [95] in order to hold the data, corresponding to a store μ-op, which is similar to our baseline described in Chapter 3. A buffer entry is allocated to the store at the rename stage. When all the μ-ops of the superblock have successfully executed a special commit operation commits the store buffer data to the cache hierarchy.

6.4.7 Handling Precise Exceptions

In an out-of-order processor one of the key role of the ROB, apart from updating the program state atomically, is to provide precise exceptions. Since we have proposed a ROB-free bulk commit logic we need to provide precise exceptions. So whenever a μ-op throws an exception, the exception information can be stored in the SOB entry corresponding to the μ-op's superblock.

Now when all the μ-ops corresponding to the superblock has been executed and the SOB entry is at the head then SOB checks whether any of the μ-op has set the exception information. If it is the case then a complete pipeline flush is triggered. The superblock is rolled-back and cold x86 code corresponding to the faulting superblock is fetched. Dual-mode decoders [52] decode the x86 instructions into μ-ops.

Since processor is in the exception mode only one of the FIFO is enabled. As a results μ-ops are executed in program order and the excepting instruction will be eventually

encountered, thereby resulting in a precise exception state.

6.5 Summary of Changes

Stage	**Baseline Processor**	**Proposed Processor**
Rename	FRRT is accessed for each μ-op. FRRT is a SRAM structure with eight read and four write ports. Physical Register corresponding to source operands are identified. Destination operand is renamed to a new Physical Register. A ROB entry is allocated for each μ-op, LSQ entry is allocated in case of a load/store. ROB is a SRAM structure with four read and write ports.	FRRT is accessed for each μ-op like in the baseline processor. However, SRRT is accessed only for those μ-ops that are live-outs of the superblocks. The physical register are updated in the SRRT corresponding to the destination operand of the live-out μ-op. SOB and SRRT are allocated in case the μ-op is head. Tail_rnm bit is set in the SOB entry if the μ-op is a tail. mio_cnt of the current SOB entry is incremented. SOB has eight entries and has four read and four write ports. ROB does not exist, hence not allocated. Gated Store buffer entry is allocated for a store.
Dispatch	μ-ops are dispatched to Issue Queue based on their type. There are four Issue Queues of sixteen entry each with four read and four write port.	μ-ops are dispatched to FIFOs based on the enhanced steering heuristic. There are two/four/eight FIFOs of eight entries each. Physical register ready table is accessed to determine whether the μ-op is ready. The physical register ready table has eight read and four write ports.
Issue	μ-ops that are ready to issue and have the functional units available are issued.	μ-ops are issued from FIFO head if they are ready to issue. Physical register ready table is accessed in issue stage as well.
Execute	μ-ops are executed in their respective functional units.	μ-ops are executed in their respective functional units. Moreover, the exe_cnt of the corresponding SOB entry is incremented.
Memory Disambiguation	For a Load a check is made to see if there exists an older store with unresolved address in the store queue or if there exists an older store with which the load aliases. If either of the two conditions are true then the load is stored in the load queue. The load queue is 48 entries and the store queue is 32 entries each with two read and write ports.	A check is made in the Gated Store buffer, 80 entries two read and write ports, to see if an older aliasing store exists. In case an older aliasing store is found the data is forwarded from the store buffer.
Commit	Commit a μ-op from ROB head if all the μ-ops corresponding to the x86 instruction are ready to commit.	For SOB head check if exe_cnt is equal to mio_cnt and whether the tail_rnm bit is set. If both the conditions are met, then copy SRRT contents to BRRT and Gated Store buffer contents are flushed to the memory hierarchy.

Table 6.1: Qualitative comparison between the baseline and the proposed processor

Table 6.1 provides a summary of changes for each pipeline stage that we had introduced in our proposal.

6.6 Evaluation

In this section, first we study the proposed out-of-order logic. We present the reasons behind performance improvement. Moreover, we show the performance benefit of our enhanced steering heuristic in detail normalized a CAM based issue logic.

Next, we study the benefits and behavior of our proposed Bulk Commit Mechanism. We show the impact of limiting the number of simultaneous superblocks in the pipeline. Next, we report results for various dynamic power related metrics for both the FIFO based out-of-order logic and the Bulk Commit Mechanism.

Finally, in order to put things in perspective we compare our co-designed processor, which consists of FIFO based out-of-order logic and Bulk Commit mechanism, to a conventional out-of-order processor. This comparison is done both in terms of performance and various dynamic power related metrics.

We use Wattch 1.02 [16] to quantify the dynamic power dissipated and energy consumed. The dynamic power results shown here are using the conditional clock gating[16]. We report results of various dynamic power related metrics such as dynamic power, energy, energy-delay product and energy-delay^2 product.

For devices that run on battery, energy is an important design constraint. If a processor finishes executing the same task by consuming lesser energy, it would be preferable[13]. The energy-delay product [17], on the other hand, is yet another dynamic power related metric that puts more weight on the performance.

The extra delay factor emphasizes performance and is appropriate for higher end systems (work stations). A lower value of the metric is preferred and it is inverse of $(MIPS)^2/(Watt)$. The energy-delay^2 product, on the other hand, puts further weight on performance. It is more suitable for high-performance domain such as work stations and servers.

[13]Given the dynamic power and execution time are reasonable.

6.6.1 Performance of new Steering Policy

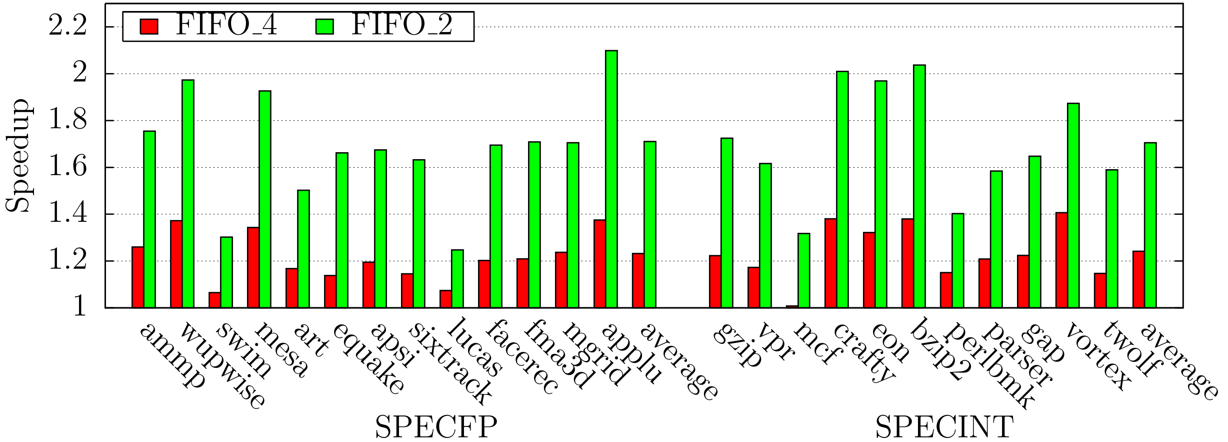


Figure 6.11: Performance of Enhanced Steering Heuristic when normalized to dependence-based steering heuristic. FIFO size = 8.

Figure 6.11 shows the performance of our enhanced steering heuristic. More specifically, in this figure we consider the HW/SW co-designed OOO with FIFOs discussed in this chapter but varying the steering heuristic. Two bars are presented, each referring to a configuration with different number of FIFOs: FIFO_4 refers to a 4 FIFO configuration design, whereas FIFO_2 refers to a 2 FIFO based design.

The two bars show the normalized speedup of our heuristic with respect to the with Palacharla dependence-based steering heuristic using equivalent number of FIFOs. As we can see from the figure, the proposed heuristic is better in performance than the dependence-based heuristic by nearly 70% and nearly 25% for a two FIFO and a four FIFO configuration, respectively. As we will see next, the main reason for the performance improvement is the reduction in stalls at dispatch.

As mentioned in Section 6.3 that our heuristic increases the dispatch throughput by reducing the unnecessary stalls. Figure 6.12 shows the stalls at the dispatch stage for a four FIFO configuration, when normalized with respect to the baseline dependence-based steering heuristic. Two bars are presented, the first one *Rdy_no*, is for the heuristic that reduces Rdy_no stalls[14], whereas the second one *Tail_no*, is for the heuristic that reduces both the Rdy_no and Tail_no stalls.

[14]See Section 6.3.2.

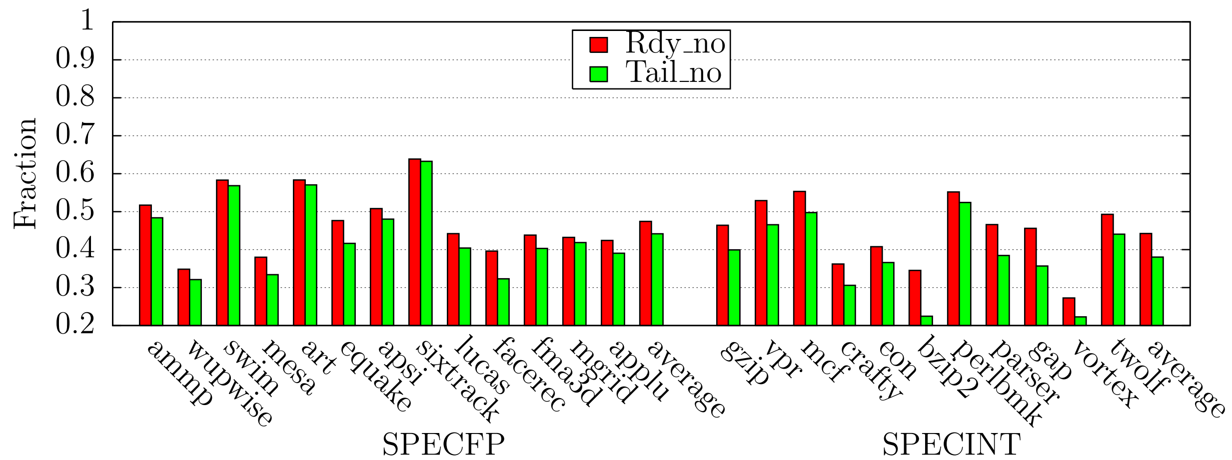


Figure 6.12: Stalls at Dispatch when normalized with respect to dependence-based heuristic. FIFO size = 8.

For SPECFP, we obtain a reduction in stalls at dispatch of 55%, whereas for SPECINT the reduction in stalls at dispatch is 62%. Since a two FIFO configuration obtains more speedup, the percentage reduction in stalls is even higher.

Earlier in Figure 6.5 we have shown the stall condition breakdown of Palacharla's dependence-based steering heuristic. Nearly 80% of the stall were due to Rdy_no and the remaining were due to Tail_no. Our enhanced steering heuristic reduces both of these stalls by steering μ-ops that have either encountered a Rdy_no or Tail_no stall to a FIFO whose tail is ready.

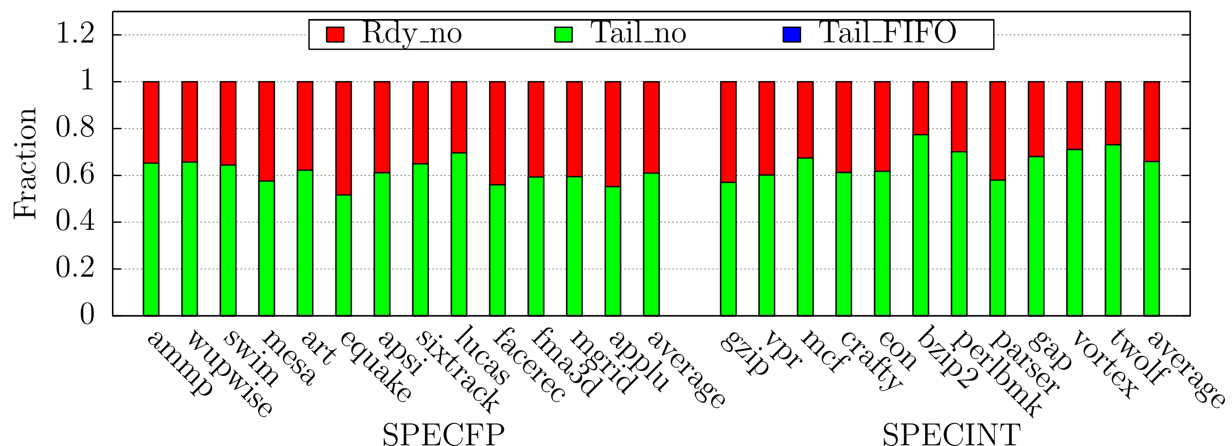


Figure 6.13: The breakdown of Dispatch Stall condition Enhanced Steering Logic. FIFO size = 8

Figure 6.13 shows the breakdown of stalls at dispatch, for our enhanced steering heuristic, for a four FIFO configuration. The percentage of stalls due to Rdy_no is reduced from

nearly 82% to nearly 40% for SPECFP, and 77% to 35% for SPECINT. Moreover, from Figures 6.5, 6.13 and 6.12 one can deduce that our enhanced steering heuristic obtains a reduction in Rdy_no stalls of 79%[15] and 82%[16] for SPECFP and SPECINT, respectively.

Steering Heuristic Benefits Characterization

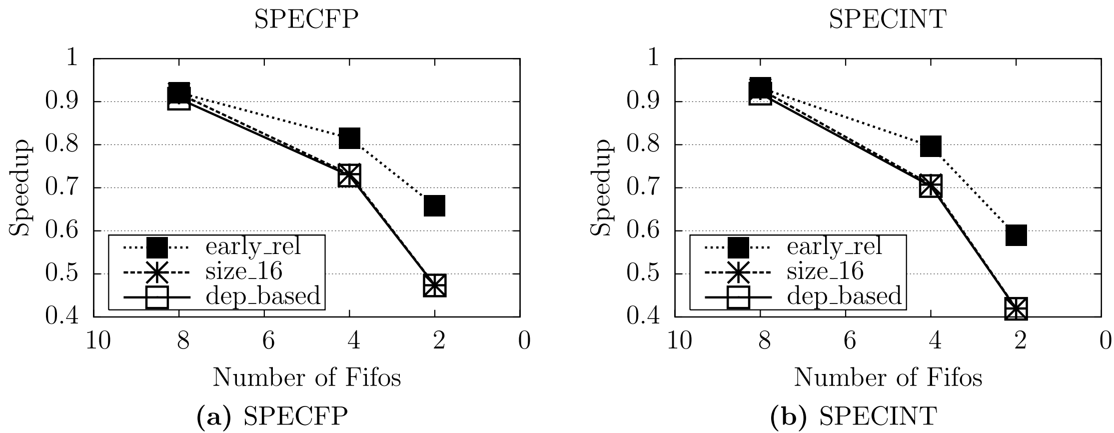
In order to better understand the performance benefits, we quantify them in this section. For this purpose we have chosen three FIFO configurations: two, four and eight.

It would also be interesting to compare FIFO based out-of-order processor to conventional issue logic based out-of-order processor. For this purpose we have replaced the FIFO logic with a CAM issue logic[17], in our out-of-order processor. All the numbers shown in this section are normalized to this CAM issue logic.

For the sake of readability, we have consistently used the same point-type (key) for a heuristic across different figures throughout this section. However, in order to differentiate between the same base heuristic with and without some microarchitectural features we use two different point-types. For instance, a base heuristic with and without early release are shown by two different point-types.

Effect of Early Release As described in Section 6.3.3, an early release mechanism releases some of the issue queue entries at the issue stage itself. As a result of which, issue queue entries are held only when a bit-vector indicates that a load was issued few cycles earlier.

[15] (100 - (40/82)*45)
[16] (100 - (35/77)*38)
[17] Details in Table 3.3.

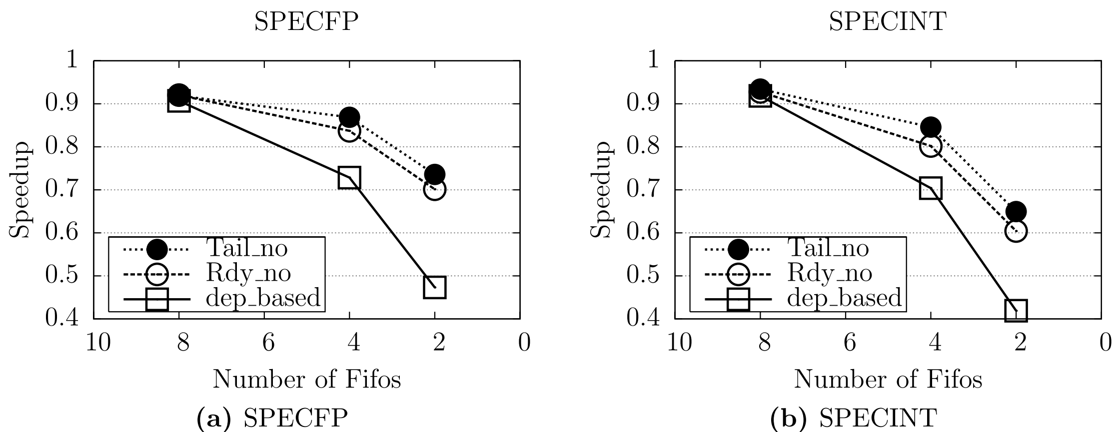


(a) SPECFP **(b)** SPECINT

Figure 6.14: The Effect of early release. Speedup normalized to CAM based issue logic. FIFO size = 8/16.

Figure 6.14 shows the impact in performance of the early release mechanism if applied to a dependence-based heuristic, as shown by *early_rel*. Clearly, early_rel is 10% and 14% better than *dep_based* for a four FIFO configuration for SPECFP and SPECINT, respectively. Moreover, for a two FIFO configuration early release mechanism results in a gain of nearly 40% with respect to *dep_based* heuristic.

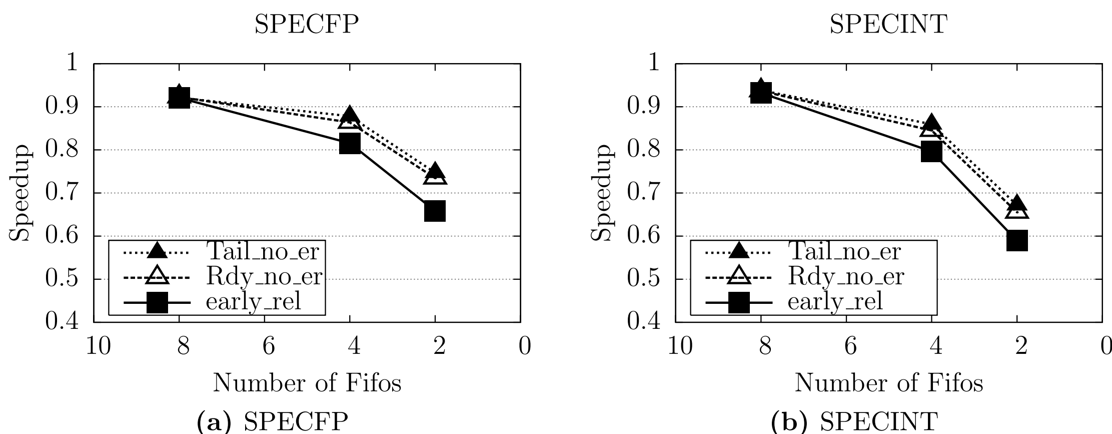
We also compare the early release to a FIFO with double the number of entries to sixteen, as shown by *size_16* in Figure 6.14. Increasing the size of a FIFO merely adds entries to the tail of the FIFO. Whereas, the early release mechanism releases entries from the head, which are critical for steering heuristic.



(a) SPECFP **(b)** SPECINT

Figure 6.15: Enhanced Steering Heuristic. Speedup normalized to CAM based issue logic. FIFO size = 8.

Enhanced Steering Heuristics Figure 6.15 shows the performance of our enhanced steering heuristics. *Rdy_no* is the one that reduces the stalls due to Rdy_no (see Section 6.3.2), whereas *Tail_no* is the one that reduces the stalls due to both Rdy_no and Tail_no.

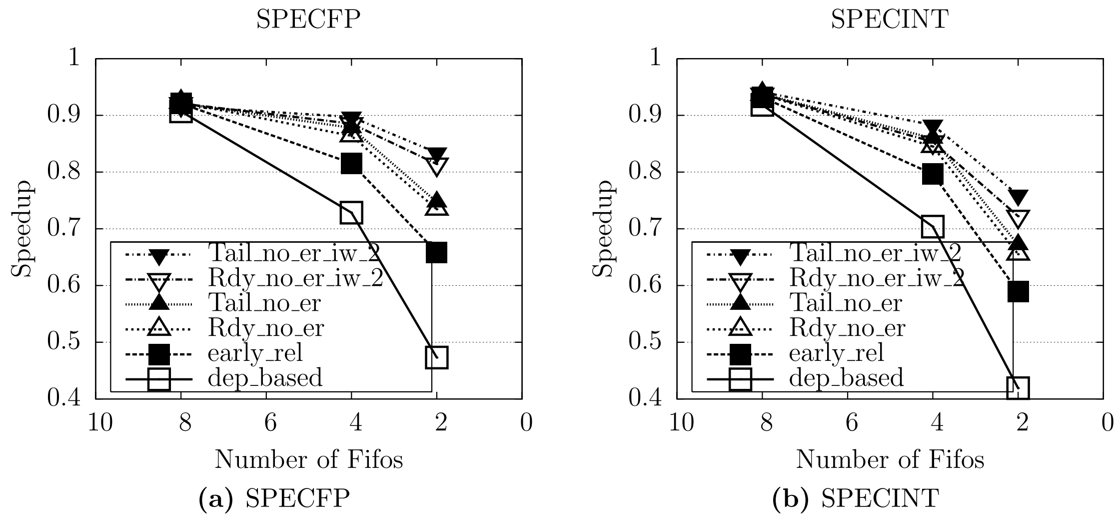


(a) SPECFP **(b)** SPECINT

Figure 6.16: The effect of Early Release when applied to Steering Heuristics. Speedup normalized to CAM based issue logic. FIFO size = 8.

We also show the performance of the enhanced heuristics with the early release in Figure 6.16. We append *er* to the existing heuristics in order to indicate them. For instance, *Rdy_no_er* is the performance of Rdy_no heuristic with early release.

In general, early release mechanism affects all the heuristics, by making the issue critical resources available. As the dependence based heuristic is more sensitive to empty FIFOs it obtains more performance benefit than our heuristic.



(a) SPECFP **(b)** SPECINT

Figure 6.17: The Effect of Per FIFO Issue Width when applied to enhanced steering heuristic. Issue width does not have any impact of dependence-based steering heuristic, hence it is not shown. Speedup normalized to CAM based issue logic. FIFO size = 8.

Effect of per FIFO Issue Width As mentioned in Section 6.3.2 our enhanced heuristic sends ready μ-ops to a FIFO whose tail is ready. This implies that the likelihood of finding multiple ready μ-ops in the same FIFO increases. We validate this intuition in the Figure 6.17 by increasing the per FIFO issue width from one to two. We append *iw_2* to the heuristic evaluated right above in order to indicate them. For instance, *Rdy_no_er_iw_2* is the heuristic that tackles Rdy_no stall and incorporate early release with an issue width of two for each FIFO.

The performance of enhanced heuristics with FIFOs that could issue two μ-ops is shown by *Rdy_no_er_iw_2* and *Tail_no_er_iw_2* in Figure 6.17. Clearly, the benefit of increasing per FIFO issue width is evident for a two FIFO configuration. For a two FIFO configuration in particular with Tail_no_er heuristic we obtain a speedup of 13% and 10% for SPECFP and SPECINT, respectively, with respect to the configuration where per FIFO issue width is one.

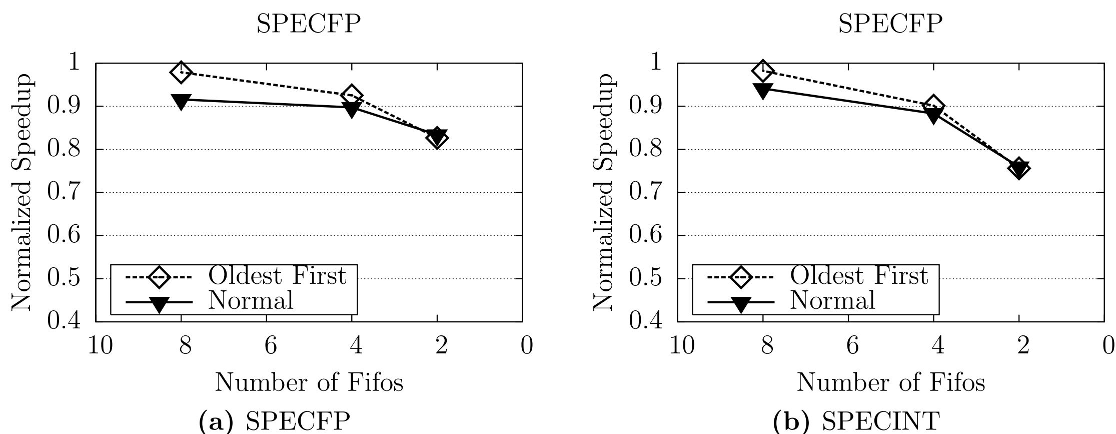


Figure 6.18: The Effect of Issue Queue Start Policy on FIFO based OOO logic with different FIFO sizes. Speedup normalized to CAM based issue logic with oldest first start policy. FIFO size = 8.

Effect of Start Policy Figure 6.18 measures the impact of *start policy* that was described in Section 6.3.4. For this purpose we measured the performance of *normal* policy with the *oldest first* policy.

As can be seen from the Figure 6.18 the performance gap between CAM based issue logic and FIFO based logic is closed further using *oldest first*. The higher the number of FIFOs the closer the performance gap gets; this is because a system with N FIFOs behaves like a CAM based out-of-order processor with 1 Issue Queue of size N.

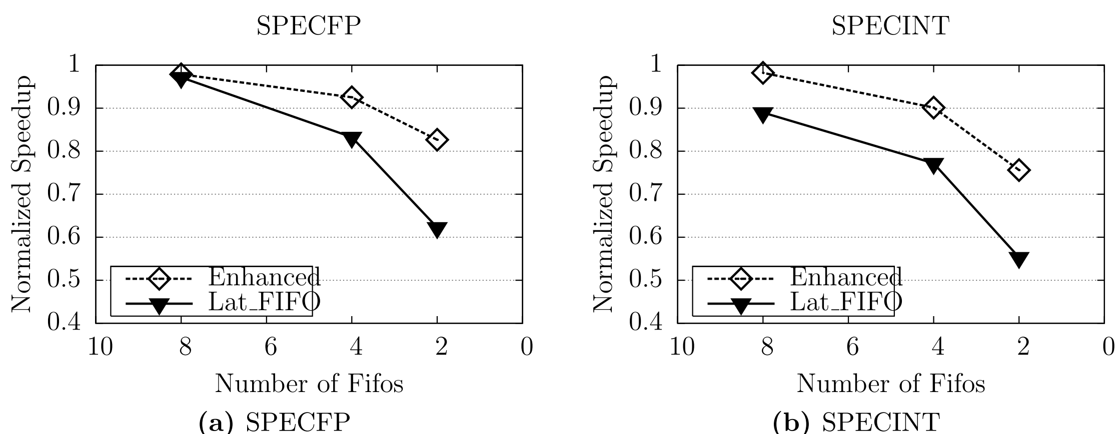


Figure 6.19: Comparison with Lat_FIFO scheme [5]. In order to be fair both of the heuristics use oldest first start policy. Speedup normalized to CAM based issue logic with oldest first start policy. FIFO size = 8.

Comparison with State-of-the-Art heuristic Abella et al. [5] have proposed a heuristic to improve the performance of FIFO based out-of-order logic for SPECFP. They refer to their heuristic as Lat_FIFO. For SPECINT they use the dependence-based heuristic.

In essence, their heuristic first tries to estimate issue cycle of an μ-op. In case according to dependence-based heuristic if no FIFO is available then instead of stalling μ-ops are steered to a FIFO whose tail would issue earlier than the μ-op being steered. The details of their heuristic can be found in their paper [5].

Figure 6.19 provides a comparison of our enhanced heuristic with one of the heuristic proposed by Abella et al. Our heuristic is shown by the line tagged by *Enhanced*, whereas the *Lat_FIFO* shows the performance of Lat_FIFO scheme.

Clearly both in SPECFP and SPECINT our heuristic performs better than the Lat_FIFO scheme. This is because first the Early Release mechanism makes more empty FIFOs available. The benefit is magnified with lower number of FIFOs. Secondly estimating the issue cycle of a μ-op is not always accurate. For instance, a μ-op that is being dispatched will have an issue cycle greater than that of the Tail of a FIFO in the Lat_FIFO scheme.

However, the tail of a FIFO might be waiting on a load that has missed. This would result the μ-op being dispatch to remain stuck. Whereas in our heuristic, we never steer to such a FIFO. A μ-op that is independent of the tail of a FIFO is steered to that FIFO only if the tail of the FIFO is ready.

6.6.2 Bulk Commit Mechanism Study

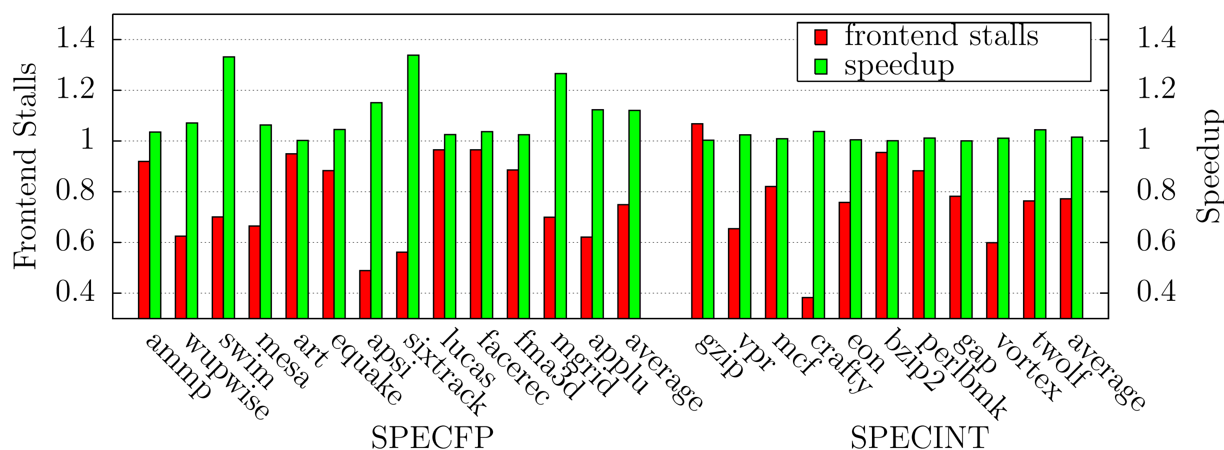


Figure 6.20: The first bar shows stalls at frontend observed by SOB based processor normalized to a ROB based processor. Both the processors are executing atomic superblocks and the size of superblocks are constrained to the size of the ROB. The second bar shows the speedup observed by SOB based processor with respect to the ROB based processor. Although in crafty frontend stalls are reduced the most but the speedup obtained is very low. This is due to the low ILP present in such benchmarks, i.e. the bottleneck is in issue not in frontend. SOB = 8, ROB = 128, 4-wide processor.

As explained earlier the atomic commit constraint of superblocks causes the conventional ROB based processor to stall. Figure 6.20 shows the reduction in stalls obtained by our SOB/SRRT commit logic with respect to conventional ROB based processor in the left y-axis *Frontend stalls*. Nearly 25% and 22% of resource related stalls are reduced in SPECFP and SPECINT, respectively.

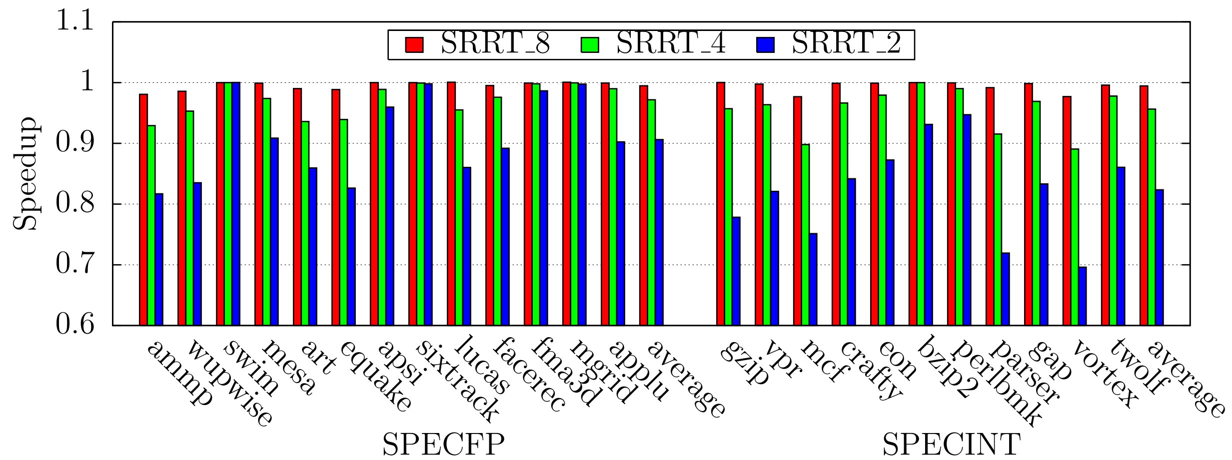


Figure 6.21: The Effect of reducing SRRTs. Results are normalized to unbounded number of SRRTs. Since number of SRRTs is equal to the number of entries in SOB. Three different SOB configurations were assumed. However, we assumed four FIFO configuration.

Figure 6.20 also shows the speedup obtained due to the reduction of these stalls in the right y-axis *Speedup*. In SPECFP we obtain a speedup of 12%, and 1.5% in SPECINT, with respect to conventional ROB based processor. The larger superblocks in SPECFP leads to more stalls, as it results in more μ-ops waiting in the ROB. Hence the performance improvement obtained in SPECFP is notably larger than that in SPECINT.

As mentioned earlier in Section 6.4 SRRTs are added to allow concurrent execution of several superblocks. In Figure 6.21 we show the impact in performance by limiting the number of simultaneous superblocks in the pipeline. In order to limit the number of simultaneous superblocks, we limit the number of SRRTs. This is exactly equivalent of limiting number of SOB entries. We show speedups of three different configurations with eight, four and two SRRTs. The speedups are normalized to the one with unbounded number of SRRTs.

As is evident from the figure that an eight SRRT configuration is as good as an unbounded SRRT configuration. Moreover, a four SRRT configuration provides performance that is within 3% of the unbounded case. Hence, a pool of four SRRTs and a SOB with four entries seems to be a good trade-off between performance and complexity.

6.6.3 Dynamic Power and Energy Results

The proposed enhanced steering heuristic does not need any additional structures with respect to the baseline dependence-based steering heuristic. However, the existing Physical Register ready table is accessed not only at the issue stage but also at the dispatch stage. This results in increase in the number of read ports on this table and , hence, the number of accesses per cycle. As a result, the dynamic power dissipated by the table

increases.

Moreover, since the enhanced steering heuristic reduces stalls and increases the IPC, the overall activity of all the processor units increases. This results in higher dynamic power dissipation from other units as well. In this section we quantify the dynamic power and energy of our proposed processor with different steering heuristics. All the results in this section are normalized to the CAM issue logic.

Dynamic Power Results of the Steering Heuristics

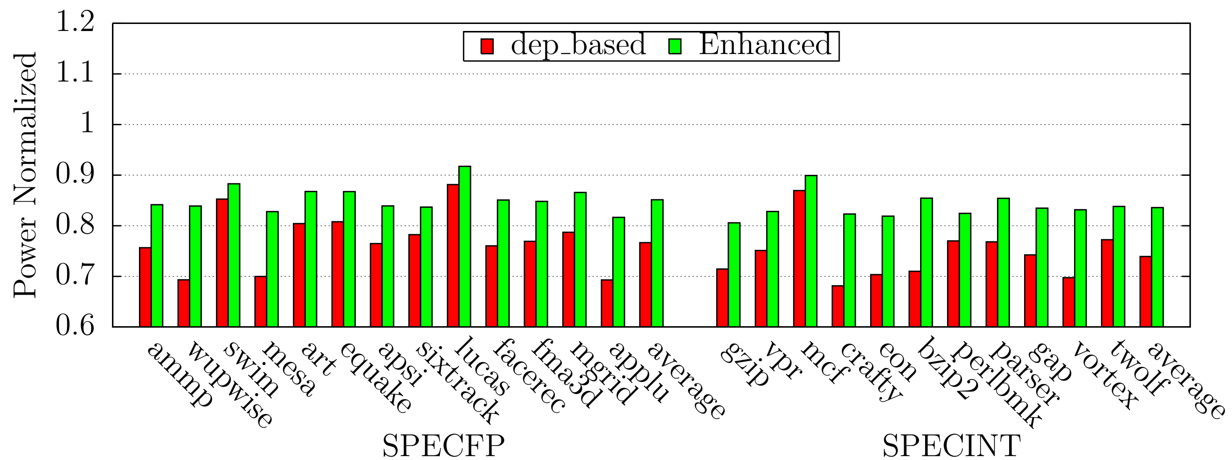


Figure 6.22: Dynamic Power Results of dependence-based steering heuristic and Enhanced steering heuristic normalized to CAM based issued logic.

Figure 6.22 shows the normalized dynamic power dissipated by the co-designed processor implementing different steering heuristics[18]. With respect to the processor with dependence based steering heuristic our enhanced steering heuristic dissipates 12% and 14% more dynamic power for SPECFP and SPECINT, respectively.

[18]Four FIFO configurations.

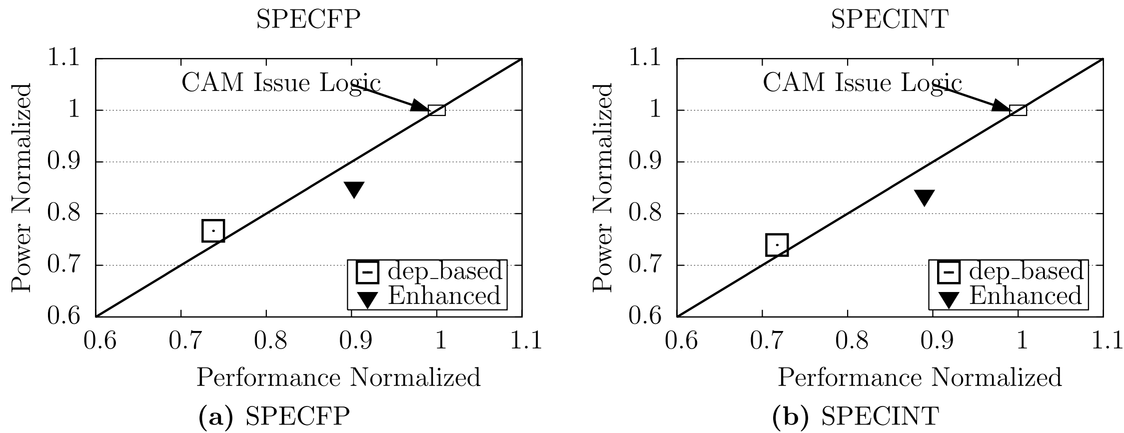


Figure 6.23: Normalized Power and Performance Results. The point 1,1 corresponds to CAM issue logic.

We also measured the dynamic power dissipated by the individual units. Together the issue and the dispatch logic of the proposed enhanced steering heuristic dissipates nearly 56% and 52% more dynamic power for SPECFP and SPECINT, respectively, compared to dependence-based heuristic. The dynamic power dissipated by these units is still nearly 2% of the dynamic power dissipated by the processor. To put things in perspective, our results show that a CAM issue logic based processor with four issue queues, each sixteen entries long, dissipates nearly 10% of the processor dynamic power[19].

Even though our heuristic dissipates more dynamic power, it is still more power-efficient than both the dependence-based heuristic and the CAM issue logic. By plotting dynamic power against performance normalized to CAM issue logic we quantify this claim in Figure 6.23. Any point that falls below the line $y = x$ is more dynamic power-efficient than CAM issue logic.

Dynamic Power Results of SOB and SRRT

We modeled both the SOB and SRRT using RAM array, to quantify the dynamic power dissipated. We assumed an eight-entry SOB and eight SRRT configuration[20]. The SOB has eight ports and each entry is five bytes wide.

We observe that our proposed processor dissipates 6% less dynamic power compared to a ROB-based processor. This main gain is due to the fact the SOB is multiple-folds smaller than a ROB. Even though there are multiple rename tables, at any given cycle at

[19]Four Issue Queues of thirty-two entries each dissipates 17% of the processor dynamic power.

[20]Four-entry SOB and four SRRTs are sufficient, but a bigger configuration provides an upper bound on dynamic power dissipation.

most two SRRTs will be accessed. A write access to a SRRT is made when a μ-op being renamed is a live-out of the superblock, while a read access to a SRRT is made when state is committed to BRRT.

Energy Results of the Steering Heuristic

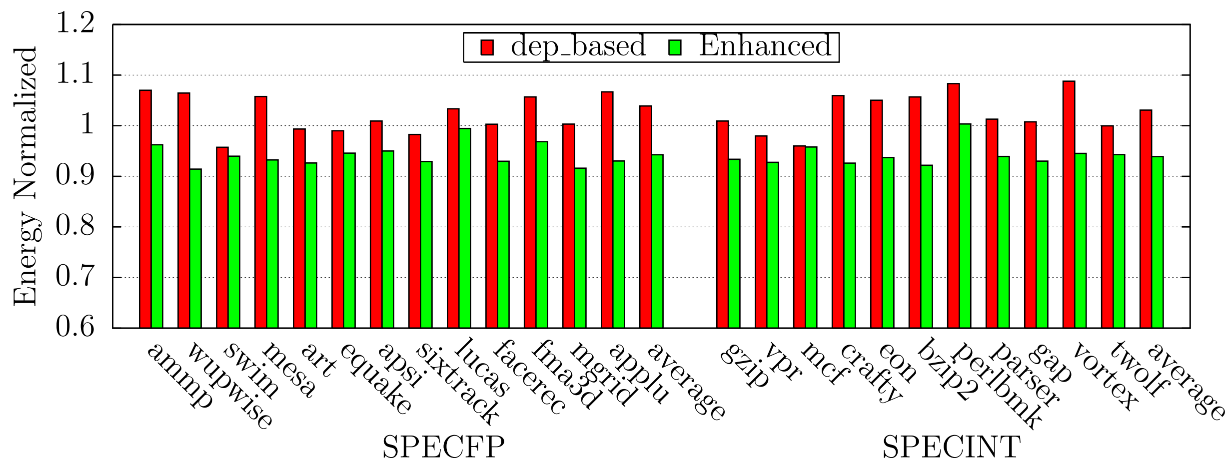


Figure 6.24: Energy Results of dependence-based steering heuristic and Enhanced steering heuristic normalized to CAM based issued logic.

Figure 6.24 shows that the enhanced steering heuristic consumes 7% less energy than CAM issue logic. The dependence-based heuristic, on the other hand, consumes 3-4% more energy.

We observed that the energy-delay product of the dependence-based heuristic is nearly 35% more than that of our heuristic. One can also deduce from the results above that our steering heuristic is more energy efficient than the dependence based one.

SpecRRT vs SRRT

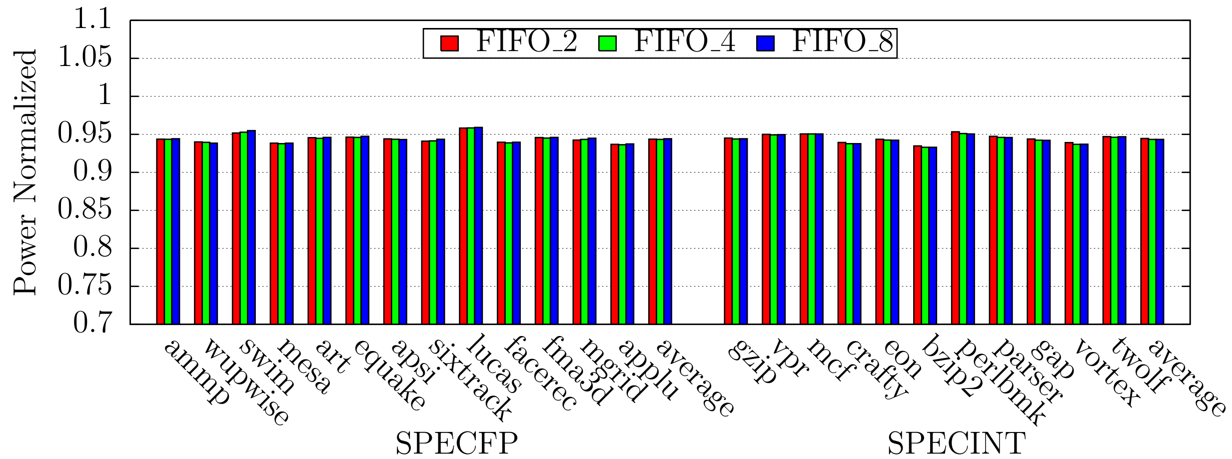


Figure 6.25: Dynamic Power Consumption of SOB and SRRT based processor normalized to ROB and SpecRRT based processor. Both the processors are executing atomic superblocks, and the issue logic is FIFO based.

We have also evaluated the difference in the two bulk commit mechanisms. SpecRRT based mechanism, which was proposed in Chapter 4 is compared to SOB and SRRT based bulk commit mechanism. In Section 6.4.1 we have shown qualitatively that SOB and SRRT based bulk commit mechanism addresses the problem when a Load encounters a miss.

However, in our experiments we had found that SpecRRT and SRRT obtain equivalent performance. This happens due to two primary reasons. Firstly, since our superblocks are smaller, the number of times a Load miss causes a ROB, in SpecRRT based solution, to stall is fewer. Secondly, the majority of our benchmarks have a small working set and generate fewer L1-cache misses and even lower L2-miss. As a result, the benefit of getting rid of the ROB and using SOB based solution was not fully exploited.

However, as the superblocks become larger or if the ROB is smaller and the applications with larger working set are evaluated we believe a SOB based solution would lead to a better performance.

Moreover, since the SOB is many folds smaller than a ROB, the SOB based processor consumes lower dynamic power. This is in spite of the fact that the SOB based solution requires multiple SRRTs to hold the register state for each superblock. In Figure 6.25 we have measured the dynamic power consumption of three FIFO configurations. The dynamic power is normalized to SpecRRT configuration.

As shown in the figure SOB based processor consumes 5% lower dynamic power in all the three different FIFO configurations, primarily because the SOB is manifolds smaller than

the ROB. Even though there are more SRRTs in comparison to a single SpecRRT, at any given point in time at most two SRRTs are being accessed.

Moreover, SRRTs and SpecRRT are both accessed only by live-outs. The difference is that in the SRRT design, the SRRTs are written in the rename stage, whereas the SpecRRT is written when μ-ops retire. Both of the structures are read and their contents are copied to the BRRT at bulk when the last μ-op commits. This indicates that dynamic power consumption due to SRRTs should be similar to that of the SpecRRT.

Hence, the key determining factor here is the reduction in dynamic power obtained due to a smaller SOB. This result can also be validated from Section 6.6.3, where we mentioned that SOB based processor executing superblocks obtains 6% reduction in dynamic power consumption compared to ROB based processor executing normal μ-ops. Notice that in this study we had assumed both these processor configurations use same issue logic.

6.6.4 SOB+FIFO Vs ROB+CAM processor

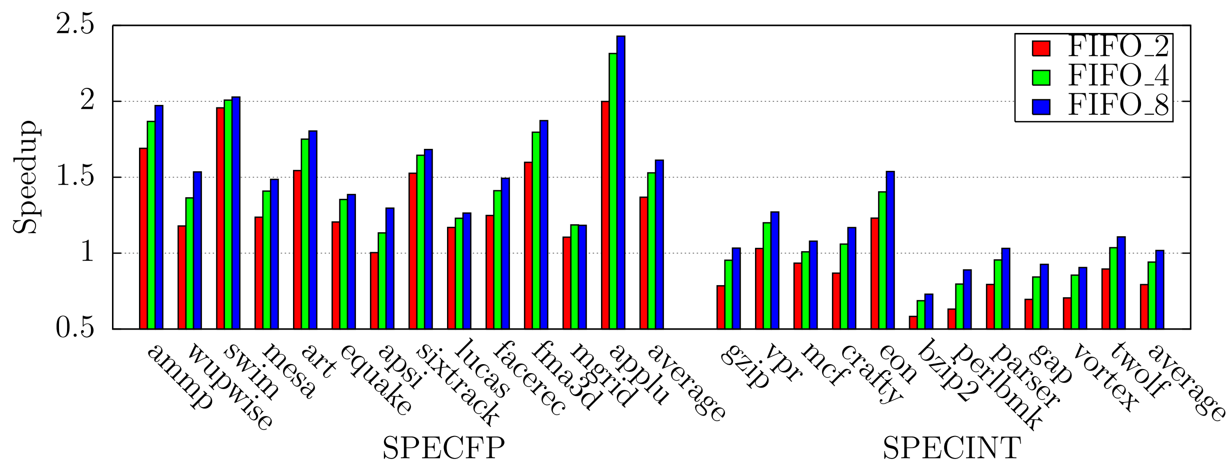


Figure 6.26: Speedup of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic executing normal μ-ops.

Finally, in this section we measure the performance of the co-designed out-of-order processor that we had proposed to a conventional ROB based processor. However, the difference here with respect to the study performed in Section 6.6.2 is described as follows.

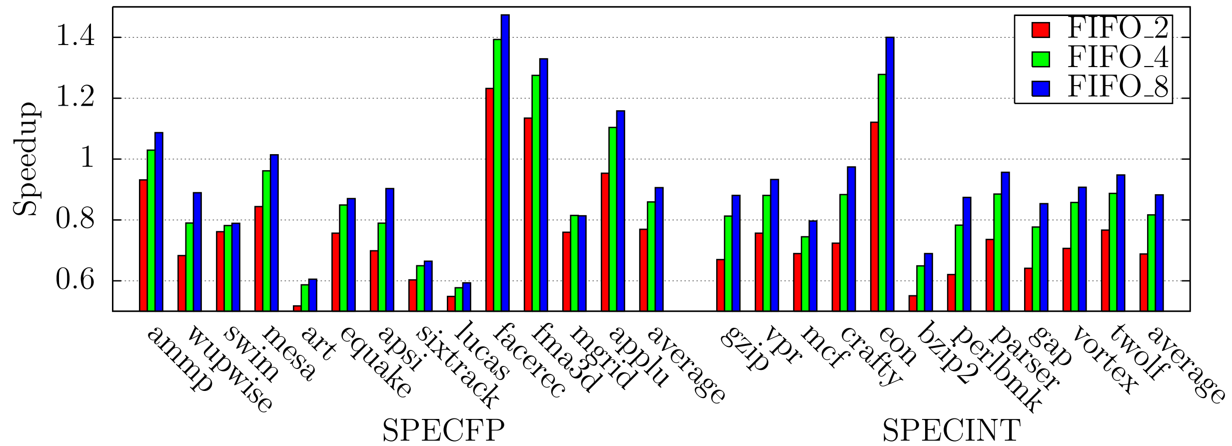


Figure 6.27: Speedup of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops.

We have compared a SOB based processor with FIFO based issue logic executing superblocks to a ROB based processor with CAM based issue logic executing normal μ-ops. None of the processor has support for HW memory disambiguation. Figure 6.26 shows the performance improvement of our proposed processor with three different FIFO configurations over the ROB based processor.

For SPECFP, as the superblocks are larger in size, as described earlier in Chapter 3, we obtain significant speedup. For instance, with four FIFOs we obtain speedup of over 50% on average. For SPECINT, on the other hand, the performance with four FIFOs is within 10% of the ROB based processor, whereas with 8 FIFOs we obtain equivalent performance.

Earlier in Chapter 3, we had shown that code optimization such as load hoisting and list scheduling narrows the gap between an in-order processor executing superblocks with an conventional out-of-order processor. Hence, in some ways dynamic compiler is applying memory disambiguation on the superblocks at the time of code optimizations.

In order to be fair we have enhanced the ROB based processor with hardware memory disambiguation. Figure 6.27 shows the results of this comparison. Similar to Figure 6.26 we have measure performance of our SOB based processor with three FIFO configuration. The only difference is that ROB based processor not only has CAM based issue logic but also CAM based LSQ.

As shown in the Figure 6.27 the performance of the SOB based processor with four FIFOs is within 20% of the ROB based processor with both CAM based issue logic and memory disambiguation. Moreover, with eight FIFOs the performance is within 15%.

Dynamic Power and Energy Results

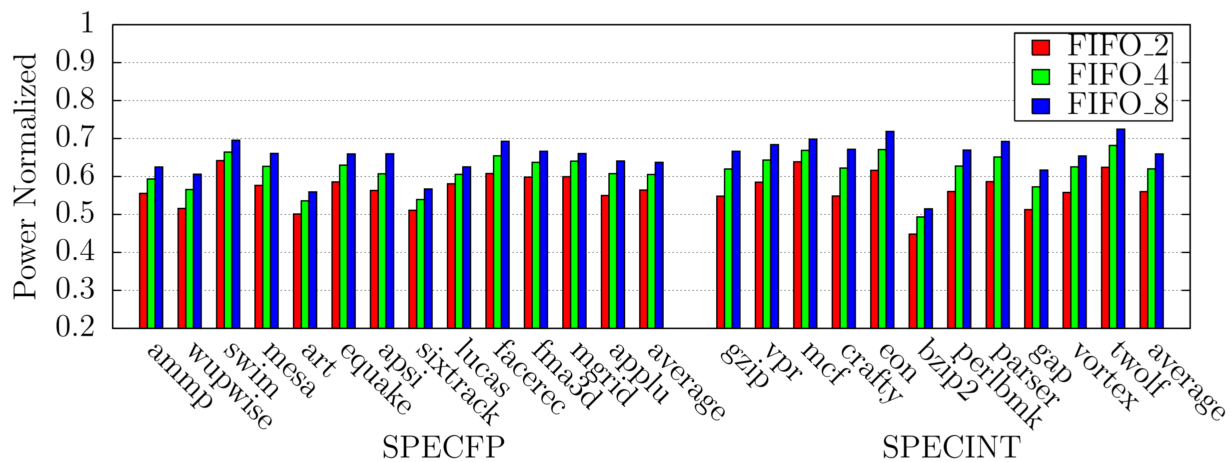


Figure 6.28: Dynamic Power consumption of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic running with memory disambiguation executing normal μ-ops.

Since the enhanced ROB based processor consists of CAM based issue logic and CAM based LSQ, it also consumes more dynamic power. Figure 6.28 shows the normalized dynamic power of the SOB based processor with respect to the enhanced ROB based processor. Both for SPECFP and SPECINT we achieve a 40% reduction in dynamic power with four and eight FIFOs on an average.

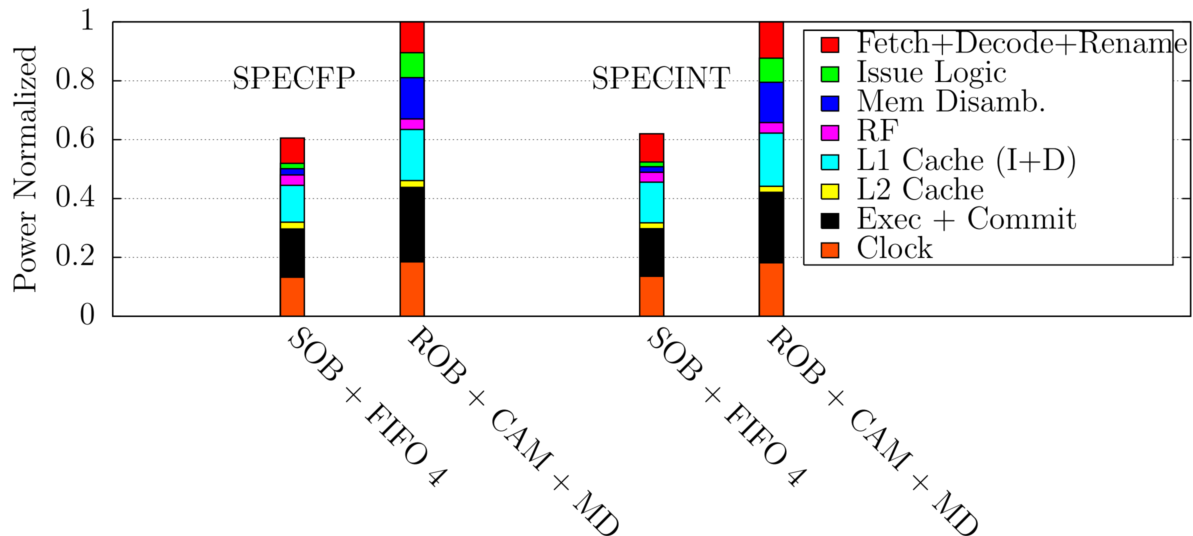


Figure 6.29: Dynamic Power consumption Distribution of SOB based processor with FIFO based issue logic executing optimized superblocks and that of ROB based processor with memory disambiguation and CAM based issue logic. Both are normalized to conventional ROB based processor with CAM based issue logic running with memory disambiguation executing normal μ-ops.

In Figure 6.29 we show the distribution of dynamic power consumption for a SOB based processor with four FIFO configuration. Moreover, the dynamic power consumption distribution of a conventional ROB based processor with CAM based issue logic and memory disambiguation is also shown.

As shown in the figure, our proposed co-designed processor obtains a 40% reduction in power, which comes from various sources. First of all, a 9% reduction is obtained in Exec + Commit. The majority of this reduction is because of lower activity in functional units. Moreover, we do not require a ROB, instead we use SOB, which is manyfolds smaller than the ROB. Similarly, due to low activity, we obtain a 8% reduction in total clock power.

One of the important source of power consumption for a conventional OOO processor is due to CAM based issue logic and memory disambiguation. FIFO based issue logic, on the other hand, hardly consumes any power. As a result of which, we obtain a 7% reduction in power in issue logic.

Moreover, another important source of power consumption in conventional out-of-order processors in due to memory disambiguation. The key structure that enables issuing of memory instructions out-of-order is a LSQ, which is a CAM based structure. As shown in the Figure 6.29, in the baseline processor the LSQ contributes nearly 14% of total power consumption. The power consumption due to Memory disambiguation in the Wattch paper [16] is reported as 11% for a Alpha 21264 like processor. The power consumption due to memory disambiguation is slightly higher due to larger LSQ in our baseline.

We obtain nearly 13% reduction in power in memory disambiguation. This is mainly because we do not have LSQ, instead we use Gated Store Buffers. With a conventional LSQ based processor, each time a store reaches the hit/miss stage[21] a CAM wake-up is performed on the queue, in order to wake-up all the waiting loads. Moreover, since stores are split into address and data μ-ops, a CAM wake-up is performed for both the μ-ops.

Moreover, when loads reach the hit/miss stage, a CAM lookup is performed to find unresolved older stores. In case an unresolved store is found, then the load is inserted in the load queue.

On the other hand, in our proposed co-designed processor, we do not have a LSQ. Instead a Gated Store Buffer is present, which holds the store data of uncommitted stores. However, for each load, a CAM lookup is performed on the buffer to find older aliasing stores.

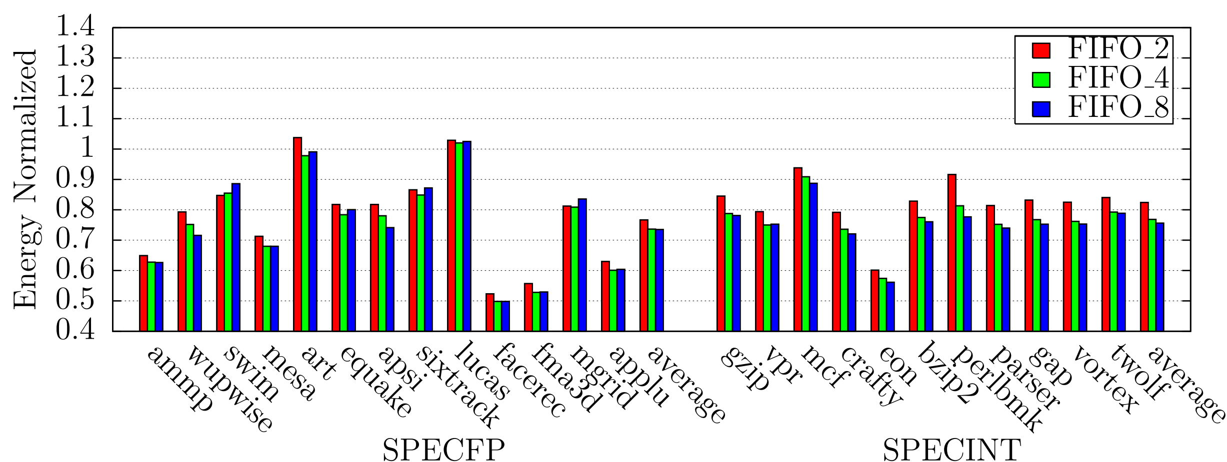


Figure 6.30: Energy Consumption of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic running with memory disambiguation executing normal μ-ops.

For handheld devices that operate on battery, energy is a very important metric. The lower the energy consumed to run an application, the longer the battery time could be provided. As shown in Figure 6.30, we achieve a 25% reduction in energy consumption with four FIFOs for both SPECFP and SPECINT.

[21] See Figure 3.2

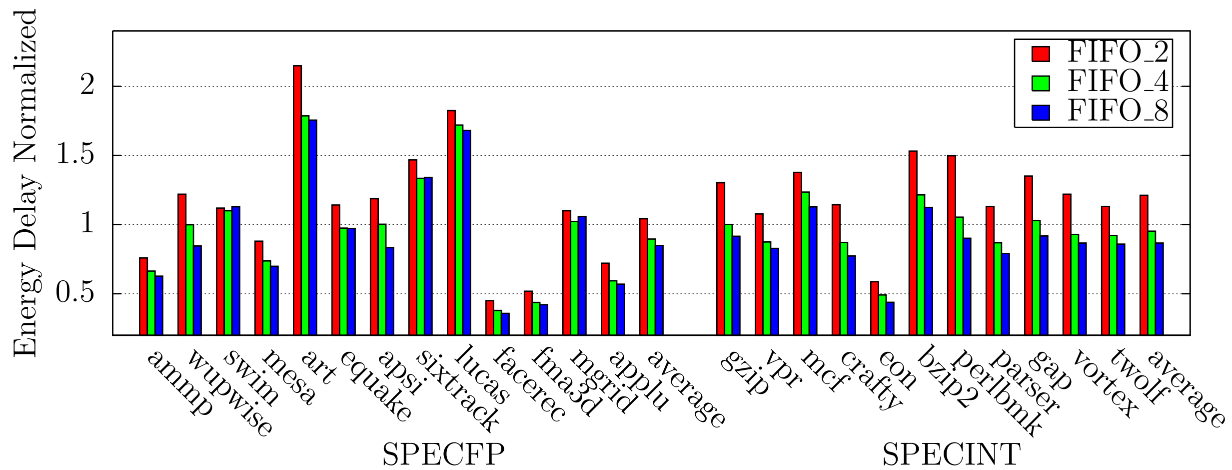


Figure 6.31: Energy-Delay of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic running with memory disambiguation executing normal μ-ops.

Finally, yet another important metric for high end devices is the Energy-Delay product. Energy-Delay product takes both the energy consumption and the execution time into account. It is simply the product of Energy consumption and execution time of an application. Figure 6.31 shows the normalized energy-delay product for three FIFO configurations.

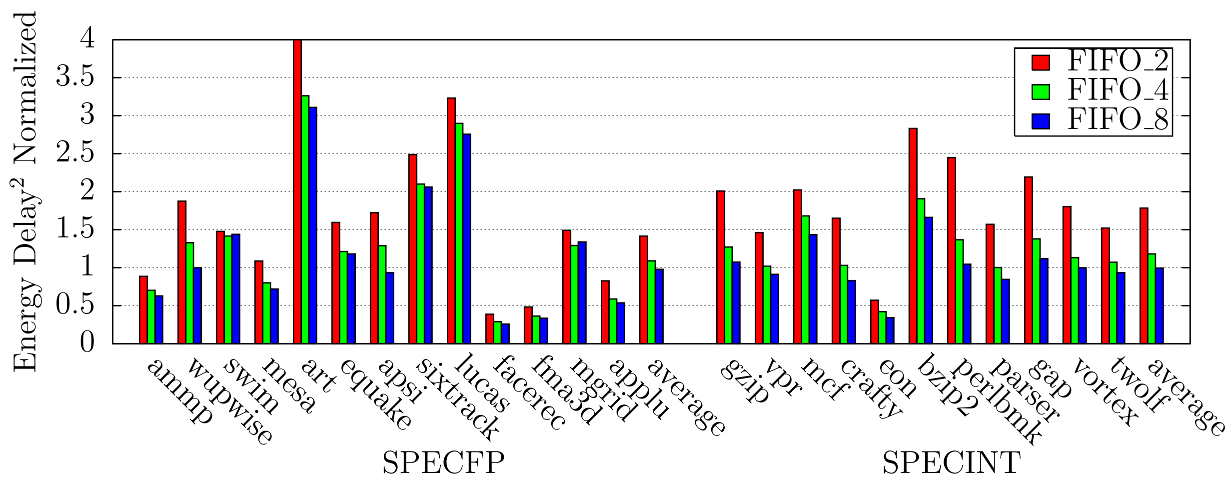


Figure 6.32: Energy-Delay^2 of SOB based processor with FIFO based issue logic executing optimized superblocks normalized to conventional ROB based processor with CAM based issue logic running with memory disambiguation executing normal μ-ops.

Low energy-delay corresponds to a preferable design point. As shown in the figure we have obtained a reduction in Energy-Delay product by 10% in SPECFP and 2% in SPECINT with four FIFOs. For some benchmarks such as lucas, sixtrack, art etc. the SOB based

processor obtains a worse energy-delay product. This is because from Figure 6.27 it can be seen that art and lucas have the worst performance.

In order to show the energy-efficiency of our proposal for high performance domain, we have also provided Energy-Delay^2 product in Figure 6.32. For high-end server machines Emery-Delay^2 product is a more suitable metric as it puts more weight on performance. Figure 6.32 shows that a SOB based processor with FIFO based issue logic also has lower Emery-Delay^2 product for four and eight FIFO configurations.

Comparison of SOB + FIFO + SB vs ROB + CAM + SB

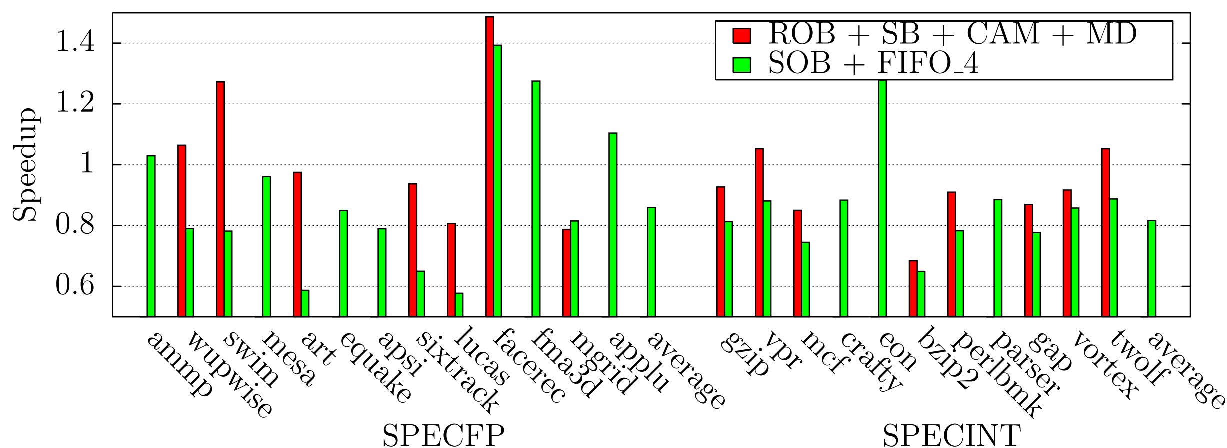


Figure 6.33: Speedup of SOB based processor with FIFO based issue logic executing optimized superblocks and that of ROB based processor with CAM based issue logic with memory disambiguation executing atomic superblocks. Both are normalized to ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops. However, executing atomic superblocks in a ROB based processor will result in deadlock for those benchmarks where superblocks are larger than ROB such as fma3d, crafty, eon etc. Hence, the results of these benchmarks are not shown.

In Figure 6.33 we compare the performance of the SOB based processor with FIFO based issue logic to ROB based processor with CAM based issue logic, both executing atomic superblocks. Moreover, the ROB based processor also has hardware memory disambiguation. Both are normalized to ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops. For the SOB based processor we have chosen a four FIFO configuration.

In most of the SPECINT benchmarks the ROB based processor executing atomic superblock encounters slowdown. This is mainly because of stalls at the frontend. In some of the benchmarks, however, such as swim and facerec the ROB based processor executing superblocks performs better than both the baseline and our SOB base processor.

This happens due to multiple factors. Firstly, CAM based issue logic and hardware memory disambiguation helps in obtaining better performance. Secondly, optimized superblocks further provides speedup. These two factors mitigate the affect of stalls due to unavailability of the ROB.

However, since atomic superblocks are executed in ROB based processor, deadlock is encountered in the case where superblocks is larger than the size of the ROB. In this experiment the size of the ROB is 128. As a result, performance of such benchmarks are not reported, such as fma3d, crafty, eon, mesa etc.

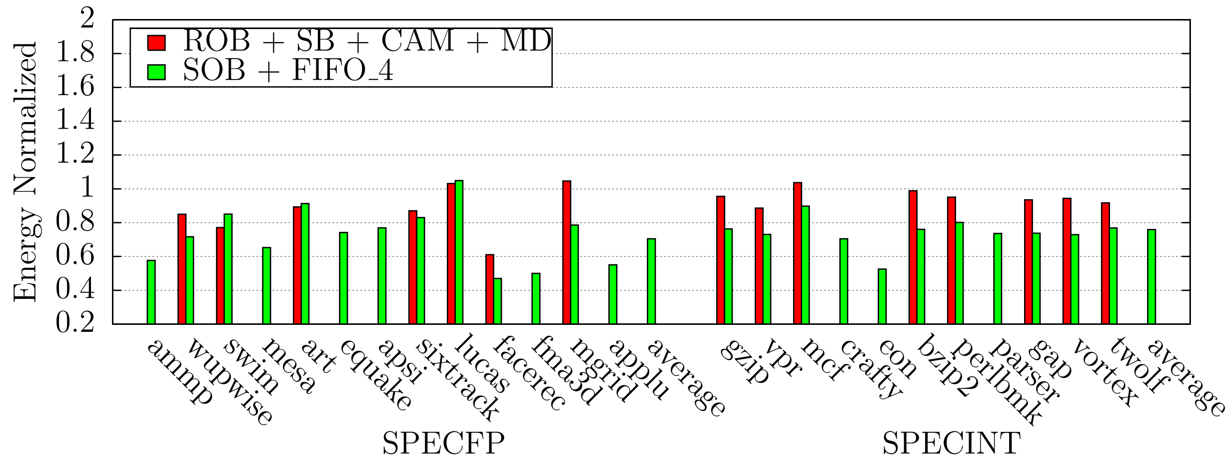


Figure 6.34: Energy of SOB based processor with FIFO based issue logic executing optimized superblocks and that of ROB based processor with CAM based issue logic with memory disambiguation executing atomic superblocks. Both are normalized to ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops. However, executing atomic superblocks in a ROB based processor will result in deadlock for those benchmarks where superblocks are larger than ROB such as fma3d, crafty, eon etc. Hence, the results of these benchmarks are not shown.

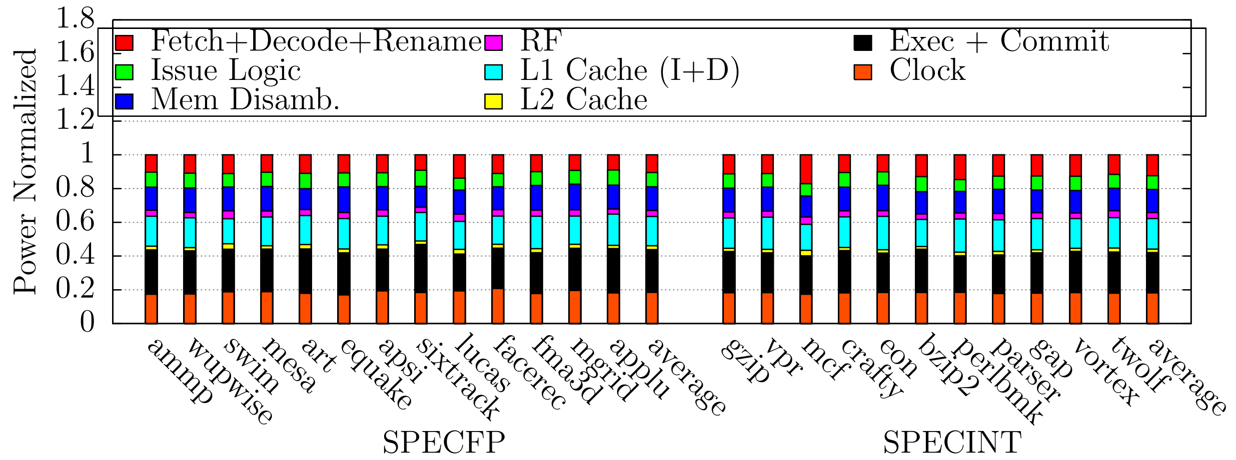


Figure 6.35: Dynamic Power consumption Distribution of ROB based processor with CAM based issue logic with memory disambiguation executing atomic superblocks. We show the distribution of those benchmarks as well that have deadlocked, upto the point where they executed.

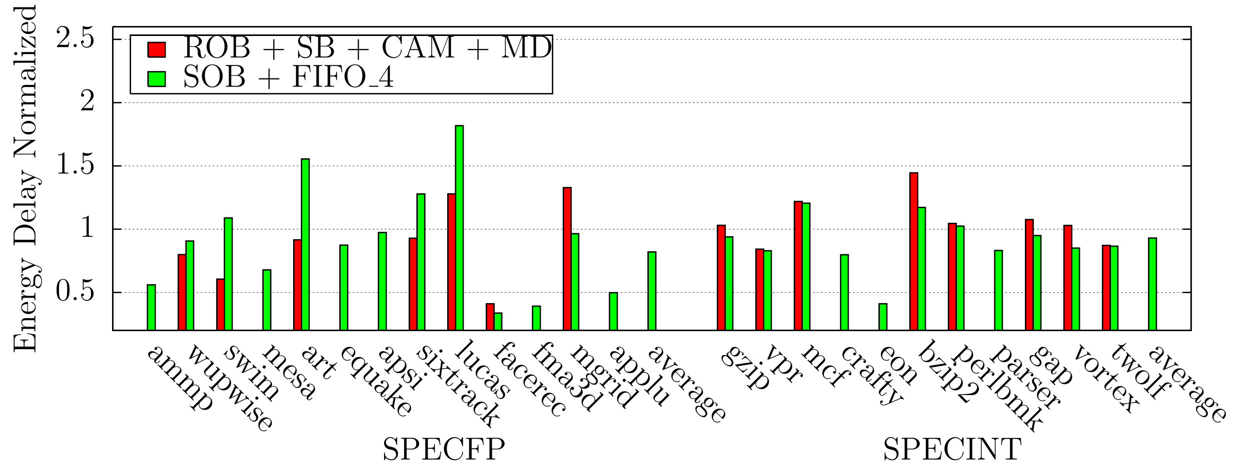


Figure 6.36: Energy-delay product of SOB based processor with FIFO based issue logic executing optimized superblocks and that of ROB based processor with CAM based issue logic with memory disambiguation executing atomic superblocks. Both are normalized to ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops. However, executing atomic superblocks in a ROB based processor will result in deadlock for those benchmarks where superblocks are larger than ROB such as fma3d, crafty, eon etc. Hence, the results of these benchmarks are not shown.

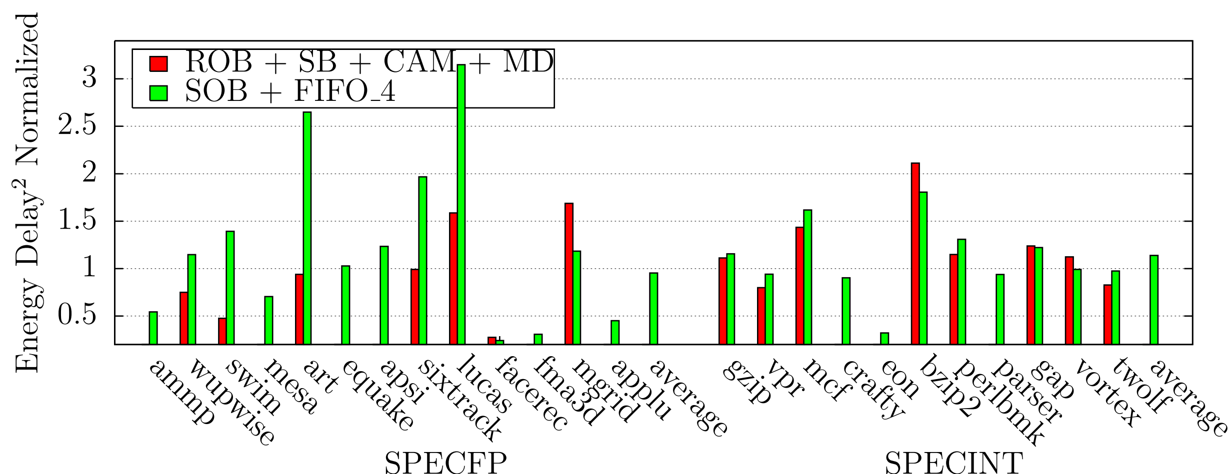


Figure 6.37: Energy-delay^2 product of SOB based processor with FIFO based issue logic executing optimized superblocks and that of ROB based processor with CAM based issue logic with memory disambiguation executing atomic superblocks. Both are normalized to ROB based processor with CAM based issue logic with memory disambiguation executing normal μ-ops. However, executing atomic superblocks in a ROB based processor will result in deadlock for those benchmarks where superblocks are larger than ROB such as fma3d, crafty, eon etc. Hence, the results of these benchmarks are not shown.

In Figures 6.34, 6.36, 6.37 we compare the energy related metric of the SOB based processor with FIFO based issue logic to ROB based processor with CAM based issue logic, both executing atomic superblocks. Moreover, the ROB based processor also has hardware memory disambiguation.

In Figure 6.35 the power consumption distribution of all the benchmarks are reported for the conventional ROB based processor. Although, as evident from the figure that the power distribution is similar for most of the benchmarks are similar, the absolute power consumption numbers vary. For instance, the total power consumption of mcf is 25% below average for SPECINT benchmarks. In mcf L2 consumes 20% more power with respect to SPECINT average. Contribution of power due to L2 for mcf is 3.3% and that for the rest of SPECINT is nearly 2%.

Figure 6.36, shows the energy-delay product. Mgrid has a higher energy-delay product when executing atomic superblocks on a conventional out-of-order processor. This is because mgrid encounters a slowdown when executing atomic superblocks on a ROB based processor. We found that one of the superblocks in mgrid has 101 μ-ops. Moreover, since the energy consumption is similar to the baseline. As a result, mgrid has 30% higher energy-delay product with respect to both the baseline and our SOB based processor.

6.7 Related Work

In Transmeta Crusoe [60] Shadow copy of Register File is used to check-point the register state before a superblock starts executing. Working copy, as the name suggests, holds the working register set of the superblock. Whereas, the memory state is held in gated store buffers [95]. A special commit operation updates the register and the memory state, at once. Since Crusoe uses VLIW processors -to cut down complexity and power-, the above bulk commit solution is sufficient.

Our microarchitecture is different since we use a FIFO based out-of-order logic. This gives us the flexibility of executing instruction out-of-order albeit in a power-efficient way. As a consequence of out-of-order execution a different bulk commit mechanism is required to commit the superblock program state.

We use SOB and SRRT based atomic commit and reordering mechanism. This allows concurrent execution of multiple superblocks and hence multiple SRRTs are providing to hold the register state. Moreover, the proposed SOB commits the program state in program order.

In Transmeta Crusoe the stores were held in Gated Store Buffers. This is similar to what we have used in our proposed co-designed processor. An alternative way to implement speculative commit of store instructions is Cherry [68] and Transmeta's Efficeon. In their scheme stores speculative commit to the L1-cache, know as the speculative cache mechanism. A speculative bit indicates whether the line has been speculatively written or not. In case of an exception the state is rolled-back by simply invalidating the speculatively written cache lines.

Both the solutions have their advantages and disadvantages. The size of gated store buffers could not scale very well due to associative searches. Furthermore, store buffers puts a limit on the number of stores that could be included in the superblock.

Speculative Caches, on the other hand, not only complicates the cache coherence protocol, but also could potentially loose performance as speculatively written lines are invalidated in case of an execution roll-back. Moreover, Speculative Caches also have to deal with a scenario of multiple stores conflicting in the same set. In our proposed co-designed processor we have used Gated Store Buffers to hold the memory state.

Akkary et al. [6] have proposed a large instruction window processor using checkpoint processing and recovery. They replaced the conventional ROB with a checkpoint buffer, which is similar to the SOB that we have proposed. However, they do not execute superblock, instead checkpoints are taken at branches that have low-confidence of being predicted correctly. We instead checkpoint when the first instruction of the superblock is renamed. Their proposal still requires a buffer to hold instructions in order to bulk commit. We instead propose per superblock map table, the SRRT, that holds the register state of the superblock.

Moreover, since we use VMM to form superblock, we get rid of confidence-based predictor. VMM also helps in providing early register recycling as number of consumer of non live-out register are determined. More importantly, our focus in this chapter is not to provide a large instruction window processor, but to provide a co-designed out-of-order processor which is more power-efficient.

Conventional Issue Queue logic is based on CAM and RAM structures [76], leading to high complexity and power dissipation [46]. Palacharla et al. [76] have proposed a multiple FIFO based issue logic, where instructions are issued from the head of FIFOs. Their proposed dependence-based steering heuristic was described in Section 6.3. As shown in the Section 6.1, the performance with this heuristic declines sharply as the number of FIFOs are halved.

Canal and González [18] proposed several schemes to issue instructions. In one of their schemes instructions are placed in a buffer that is indexed by the physical register identifier. It is based on the observations that nearly one-quarter of the dynamic instructions have one of their operands available at dispatch. Their another scheme is based on computing the issue cycle of each instruction at dispatch. The circuit required to estimate the issue cycle of an instruction is not presented, it may not be trivial to estimate the issue cycle in a single cycle. We instead use a simpler FIFO based issue logic with enhanced steering heuristic, which does not require CAM logic at all.

Michaud and Seznec [72] proposed a two-level issue queue such that the small first level works as a conventional CAM/RAM issue queue and the second level store instructions but has no wake-up capability. Few prescheduling stages are added that estimate the issue cycle of an instruction and place the instruction in the second level queue at an appropriate location. Instructions that reach the head of the second level queue are issued, if ready; are sent to first level queue otherwise. We instead use a simpler FIFO based issue logic with enhanced steering heuristic, which does not require CAM logic at all.

Abella and González [5] proposed a distributed issue queue design, with multiple chains in a single FIFO. As a result of which, the effect of long latency instructions can, somewhat, be mitigated, by issuing independent instructions from the same FIFO. Each queue has an associated selection logic, which picks up just one instruction from the queue, and a small table for the chain latencies. This latency information along with encoded age identifier is used to select the instruction to issue. The selection logic is not trivial as noted by the authors.

We have compared our proposed steering heuristic to the Lat_FIFO steering heuristic proposed by the authors. Our steering heuristic performs better than the Lat_FIFO steering heuristic for both SPECFP and SPECINT. Their alternate issue logic, as described above, is not a new steering heuristic but an alternate issue queue design with FIFOs using multiple heads. Since our goal was to compare steering heuristics on a FIFO based issue logic, we did not compare with their design.

A comprehensive survey of issue queue designs is provided in [4]. We encourage readers to read the survey which discusses various issue queue designs and the trade-offs involved.

6.8 Conclusion

This chapter presented a complexity-effective co-designed out-of-order processor. By analyzing the various stalls, we propose enhanced heuristic. μ-ops are steered to a FIFO that have its tail ready. For a dependence-based heuristic if the tail of a FIFO is ready, then all the μ-ops in front of the tail are ready as well. Such a FIFO is as good as an empty FIFO. The enhanced heuristic benefits, further, by the increase in the issue width of each FIFO; as the heuristic increases the likelihood of finding a ready μ-op.

We have also proposed an early issue queue entry releasing mechanism. Issue queue entries are released at the issue stage; given that no loads were issued, a fixed number of cycles, earlier. This helps in reducing the pressure on issue queues.

Our proposed steering heuristic, compared to the dependence-based heuristic, obtains speedups of 25% and 24% for SPECINT and SPECFP, respectively. We have also shown that our proposed steering heuristic based processor consumes 10% less energy than the previously proposed steering heuristic.

Moreover, we have also shown by choosing the order in which ready μ-ops from the head of the FIFO are issued also has a significant impact in performance. For instance with eight FIFOs and issuing the oldest μ-op the performance gap between CAM based issue logic and FIFO based issued logic is nearly closed.

In order to efficiently execute superblocks, we co-design the commit logic. We had first shown various kinds of problem associated with bulk commit of atomic superblocks. In one of our solution we have proposed two structures - Superblock Order Buffer (SOB) and Superblock Register Rename Tables (SRRT) - in order to achieve this. Such a processor dissipates 6% less power than a conventional ROB based out-of-order processor and performs 12% better over a conventional ROB based processor.

Overall in this chapter we have considered a HW/SW co-designed out-of-order processor based on FIFOs and with new steering, issue and bulk commit improvements. All these improvements results in a design that is within 20% of performance of a conventional out-of-order design, with a reduction in energy by 25% for a four FIFO configuration.

Chapter 7

Conclusions

The need for uniprocessors that provide the needed performance is always important. This thesis has focused on uniprocessors; increasing their performance using accelerators, and increased its energy-efficiency using middle-ground design. The research was performed in a HW/SW Co-designed environment, where the processor is partly implemented in software and partly in hardware.

A simpler microarchitecture when accompanied with a dynamic compiler and applied at a full-system level lead to the invention of the Co-designed Virtual Machines (Cd-VM) [88]. In such a scheme, a processor is a co-designed effort between hardware and software designers. In other words the ISA is implemented partly in software and partly in hardware. The software layer performs dynamic translation and optimization on the source code to adapt it to better exploit the capabilities of the underlying microarchitecture.

Primarily, Cd-VM enables microarchitects to migrate to a simpler core, thereby cutting power consumption and complexity. It also enables microarchitects to add accelerators in order to speedup parts of applications. Moreover, it enables microarchitects to choose from a wide spectrum of the processor design space ranging from low-end in-order processors to aggressive out-of-order processors.

In Chapter 4 we have proposed a novel co-designed programmable functional unit along with a novel execution model. We have also proposed code generation heuristic specific to μ-op fusion. The Cd-VM monitor selected μ-ops for fusion and generates them. The fused μ-op is known as a *macro-op*. By collapsing and executing a chain of simple ALU μ-ops with low latency, performance is improved.

Moreover, since we have used a co-designed out-of-order processor, we have proposed a novel bulk commit mechanism. We provided a solution towards *Bulk Commit Mechanism* using an additional Register Rename Table (RRT), the SpecRRT.

We have obtained average speedups of 17% in SPECFP and 10% in SPECINT. We have

also obtained speedups of up-to 33 % is some benchmarks. Moreover, with a slight modification in the proposed MOP model we have obtained improvements in performance of 29% in SPECFP and 19% in SPECINT.

In Chapter 5 we have proposed a co-designed in-order processor using two application specific accelerators. In this work, fusion of dependent and independent μ-ops are considered separately. Dependent μ-op fusion is called as *vertical fusion*, whereas independent μ-op fusion is called as *horizontal fusion*. The two techniques accelerates the application by combining the most commonly found pairs of μ-ops in applications. Overall SoftHV resulted in an interesting co-design approach that obtained an average speedup of 9% for SPECFP and 10% for SPECINT over a conventional 4-way in-order processor.

In Chapter 6 we have proposed a co-designed out-of-order with FIFO based out-of-order issue logic. We have proposed new steering heuristics to dispatch μ-ops to the FIFOs. We have shown that FIFO based out-of-order processor is a middle-ground solution between an in-order processor and a CAM based out-of-order processor.

Moreover, we have proposed another *Bulk Commit Mechanism* in the context of out-of-order processors. In this particular solution we get rid of the ROB entirely and instead maintain the order at the granularity of the superblock using Superblock Ordering Buffer (SOB). The register state of each superblock is held in per Superblock Register Rename Table, the SRRT.

Our proposed steering heuristic, compared to the dependence-based heuristic, obtains speedups of 25% and 24% for SPECINT and SPECFP, respectively. We have also shown that our proposed steering heuristic based processor consumes 10% less energy than the previously proposed steering heuristic.

Our proposed co-designed out-of-order processor results in a design that is within 20% of performance of a conventional out-of-order design, with a reduction in energy by 25% for a four FIFO configuration.

7.1 Future Work

7.1.1 Larger Regions

Table 3.1 in Chapter 3 shows the average size of the superblocks. The average size of superblocks in SPECFP is 32 and in SPECINT is 13. Since SPECFP has more regular code and the branches are highly predictable, it results in larger superblocks.

Larger superblocks provide a larger scope for the dynamic optimizer. This results in a better optimized code, by reordering the μ-ops. The same property is exploited by the large μ-op window of out-of-order processors.

Better superblock formation heuristics when accompanied with procedure inlining and/or loop unrolling can increase the size of superblocks. Moreover, other kind of regions that fuses both the paths of a branch can lead to better regions. For instance, hyperblocks [66] or tree regions [10] could be used to better provide the context of an application.

7.1.2 Coarse-grained Accelerators

In this thesis we had just considered FU based accelerators. These accelerators are added into the datapath of the processor. They read and writeback to the Register File. Coarse-grained accelerators, on the other hand, can execute operations equivalent to a large number of fused μ-ops. Such accelerators are usually kept outside the processor.

For kernel oriented applications, tasks could be offloaded to these accelerators. Many applications fall into this category. Coarse-grained accelerators can also be co-designed, where the runtime selects a large groups of μ-ops suitable enough to be offloaded to the accelerator. One novel co-designed approach of Coarse-grained accelerators is VEAL [24].

7.1.3 Alternate Issue Logic

In this thesis, we only considered FIFO based out-of-order issue logic. As observed in Chapter 6 we have narrowed the performance gap between FIFO based and the CAM based issue logic. However, in the presence of memory disambiguation this gap widens back again.

In the presence of hardware memory disambiguation CAM based issue logic provides more opportunity for μ-ops to execute. Whereas in FIFOs the μ-ops are issued just from the FIFO heads. As a result, the dependents of a load that are stuck would not let other μ-ops to execute.

Hence, an alternate issue logic such as a Priority Queue based one [72, 19] could be used to close this gap. In such a scheme μ-ops are placed in the Priority Queue based on their estimated issue cycle. However, there are issues with the existing designs. For instance, in case of a branch misprediction, it is difficult to remove μ-ops that are in the wrong path. This is because they have now been interleaved with older μ-ops in the Priority Queue. Moreover, the circuit required to estimate the issue cycle of an μ-op need not be trivial.

Furthermore, Abella et al. [5] had proposed an alternate issue queue design using FIFOs with multiple heads. This design can be combined in a co-designed fashion to further exploit out-of-order issue capabilities.

7.1.4 Co-designing the Steering Heuristic

In Chapter 6 we had proposed an enhanced steering heuristic. This steering heuristic was completely hardware based. Perhaps a co-designed approach could be taken where a dependent chain of μ-op could be marked by the VMM. This would result in most of the steering related decision being pre-determined. Moreover, a better scheduling heuristic could be devised which could arrange the μ-ops in such an order that would reduce stalls at dispatch.

7.1.5 Speculative Caches

In this thesis we have only considered Gated Store Buffers [95] in order to hold the memory state. Gated Store Buffers limit the number of stores that can be included in a superblock. Moreover, committing from gated store buffer to the memory hierarchy may take several cycles.

In order to overcome these limitations Transmeta's Efficeon uses a Speculative Cache. However, Speculative Caches have some issues in general and specific to out-of-order co-designed processors. For instance, there could be multiple conflicting stores to the same line. One way of handling them is to rollback the superblock whenever, such a situation is encountered.

Another solution could be using a write-through cache with no-write-allocate policy. But this would require the speculative state to be extended to the L2-cache. In a Chip multi-processor environment where the L2-cache is generally shared it may be a problem. Instead, of writing to L2-cache write-buffers can be introduced between L1 and L2 cache.

Moreover, in case of co-designed out-of-order processors stores from different superblocks could write to the same line. Now suppose if the second superblock were to roll-back, then all the cache lines written by its stores have to be discarded. However, the cache lines may hold values of stores corresponding to the older superblocks.

This requires a novel solution to discard only some of the state or to restore the cache line. It would be interesting to see if speculative caches when combined with out-of-order processor provides a better solution. This case, however, would not occur in co-designed in-order processors as the speculative cache is cleared before a new superblock executes.

7.1.6 More accurate Cd-VM modeling

In Chapter 3, Section 3.4 we had described the limitations of our implementation. We had implemented an unbounded JTLB, which implied that once a superblock its corresponding mapping is always present in the JTLB. This part can perhaps be handled such that a

bounded JTLB is modeled. A simple LRU based replacement policy can be modeled for the JTLB.

Moreover, the impact of a limited space in *Code Cache* can also be modeled by using an LRU based replacement policy. The overhead due to re-translation and re-optimization can then be taken into account while estimating translation overhead.

Furthermore, whenever a superblock has to be formed, there must be a context switch to the Virtual Machine Monitor. Or for that matter any trap must invoke the VMM first. Handling the trap requires a context switch mechanism and executing the VMM code.

An accurate implementation must then save the context in case of a trap and execute the VMM code in the microarchitectural simulator. Note that our microarchitectural simulator is inject inside the application memory image. The application is simulated by forking a process and the simulator reading and executing code from the application binary.

The context switch support can be extended in our simulator by keeping the VMM code in a separate segment in the memory of the application's address space. As a result in a context switch to VMM the code from this segment could be simulated in the microarchitectural simulator.

However, the above implementation still would not simulate a full system environment. Perhaps in order to support that the full system version of the simulator must be used.

7.1.7 Comparison with other Fine-grain Accelerators

There exists large differences between the proposals made in the prior works. Moreover, the frameworks used in the prior works is quite different from the one used in this thesis. As a result, a detailed comparison with the prior work is not an easy task. Therefore we have left that as a future work.

7.2 List of Publications

Following are the list of publications that resulted from this thesis.

- A Co-designed HW/SW Approach to General Purpose Program Acceleration using a Programmable Functional Unit. *(INTERACT'11)*, 2011.
- A HW/SW Co-designed Programmable Functional Unit. *(CAL'11)*, 2011.
- SoftHV : A HW/SW Co-designed Processor with Horizontal and Vertical Fusion. *(CF'11)*, 2011.

- A Power-efficient Co-designed Out-of-Order Processor. *(SBAC-PAD'11)*, 2011.

Acronyms

AGU Address Generation Unit
ALU Arithmetic and Logical Unit
AMD Advanced Micro Devices
ARM Advanced RISC Machines
BB Basic Block
BOA Binary-translated Optimized Architecture
BRRT Backend Register Rename Table
BTB Branch Target Buffer
CAM Content-Addressable Memory
Cd-VM Co-designed Virtual Machine
CMOP Computtation Macro Operation
D-Cache Data Cache
DAISY Dynamically Architected Instruction Set from Yorktown
DEC Digital Equipment Corporation
DRAM Dynamic Random Access Memory
FIFO First-in First-out
FRRT Frontend Register Rename Table
FU Functional Unit
HW Hardware
IBM International Business Machines
I-Cache Instruction Cache
ILP Instruction Level Parallelism
ISA Instruction Set Architecture
JIT Just-in-Time
JTLB Jump Translation Lookaside Buffer
LDU Load Unit
LRU Least Recently Used
LSB Least Significant Bit
LSQ Load Store Queue
macro-op Macro Operation
MIMD Multiple Instruction Multiple Data
MOP Macro Operation
MSB Most Significant Bit
MS2B Two Most Significant Bits
μ-op Micro Operation
OOO Out-of-Order
PFU Programmable Functional Unit
RAS Retrurn Address Stack
ROB Reorder Buffer
SB SuperBlock
SIMD Single Instruction Multiple Data
SOB Superblock Ordering Buffer
SoftHV A HW/SW co-designed processor with horizontal and vertical fusion
SPC Source PC

- **SPEC** Standard Performance Evaluation Corporation
- **SPECFP** SPEC Floating Point Benchmarks
- **SPECINT** SPEC Integer Benchmarks
- **SpecRRT** Speculative Register Rename Table
- **SRAM** Static Random Access Memory
- **SRRT** Superblock Register Rename Table
- **SSA** Single Stage Assignment
- **STU** Store Unit
- **SW** Software
- **TLB** Translation Lookaside Buffer
- **TPC** Target PC
- **VLIW** Very Long Instruction Word
- **VMM** Virtual Machine Monitor

Index

AtomicSuperblocks, 39

BulkCommit, 71, 123

CMOP, 62
cmop, 61
CMS, 27
Co-DesignedVirtualMachine, 16
CodeCache, 21
CodeOptimizations, 23
CompletionRate, 40
Configuration, 67
ConfigurationCache, 67
ControlSignals, 66

FIFO, 114
Fusion, 100
FusionPerformanceObjectiveFunction, 77

GatedStoreBuffers, 127

HierarchicalIssueQueue, 68
HotSpotThreshold, 40

ICALU, 95
InternalRegisterFile, 65

ld-set, 61
ListScheduling, 43
LoopScheduling, 77

mv-set, 61

optimizer, 28

PhysicalRegisterCycling, 126
PreScheduling, 73
ProcessingElement, 66
ProgrammableFunctionalUnit, 64

RMOP, 62

SOB, 125
SpecRRT, 72
split-mop, 61
SRRT, 124
SuperblockTranslationOverhead, 48

TranslationOverhead, 22

VLDU, 95
VMM, 17

Bibliography

[1] Arco (architectures and compilers) research group. 29

[2] Arm cortex-a15, 20. 6

[3] Standard performance evaluation corporation (spec), 2000. 1, 7

[4] J. Abella, R. Canal, and A. González. Power- and complexity-aware issue queue designs. *IEEE Micro*, 23, 2003. 154

[5] J. Abella and A. González. Low-Complexity Distributed Issue Queue. In *Intl. Symp. on High Performance Computer Architecture*, 2004. 134, 135, 153, 157

[6] H. Akkary, R. Rajwar, and S.T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003. 152

[7] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The Architecture of a Binary Translation Processor. Technical report, IBM, 1999. 24, 25

[8] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010. 2

[9] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *ACM SIGPLAN Intl. Conf. on Programming Language Design and Implementation*, 2000. 24, 39, 46, 50

[10] S. Banerjia, W.A. Havanki, and T.M. Conte. Treegion Scheduling for Highly Parallel Processors. In *European Conf. on Parallel Processing*, 1997. 157

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Intl. Symp. on Operating Systems Principles*, 2003. 24

[12] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. 26

[13] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, 2006. 1

[14] Marc Berndl and Laurie Hendren. Dynamic profiling and trace cache generation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 276–285, 2003. 27

[15] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *IEEE Intl. Symp. on Microarchitecture*, 2004. 87

[16] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, 2000. 34, 127, 145

[17] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20, 2000. 127

[18] R. Canal and A. González. A low-complexity issue logic. In *Intl. conference on Supercomputing*, 2000. 108, 114, 153

[19] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, 2001. 157

[20] J.E. Carrillo and P. Chow. The Effect of Reconfigurable Units in Superscalar Processors. In *ACM/SIGDA Intl. Symp. on Field Programmable Gate-Arrays*, 2001. 86, 87

[21] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, 1997. 27

[22] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S.B. Yadavalli, and J. Yates. FX!32 A Profile-Directed Binary Translator. *IEEE MICRO*, 1998. 24, 50

[23] C. Cifuentes and M. Van Emmerik. Uqbt: adaptable binary translation at low cost. *Computer*, 33(3):60 –66, mar 2000. 27

[24] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *IEEE Intl. Symp. on Computer Architecture*, 2008. 157

[25] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. In *IEEE Intl. Symp. on High-Performance Computer Architecture*, 2007. 88

[26] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction set customization. In *IEEE Intl. Symp. on Microarchitecture*, 2004. 9, 58, 63, 79, 80, 82, 85, 86, 87

[27] R.P. Colwell. The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips. *IEEE Computer Society Press*, 2006. 24

[28] T. Conte, A.B. Patel, and J.S. Cox. Using branch handling hardware to support profile-driven optimization. In *IEEE Intl. Symp. on Microarchitecture*, 1994. 54

[29] K.D. Cooper, P.J. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. Technical report, Dept. of Computer Science, Rice University, 1998. 41

[30] D. Cronquist, P. Franklin, S. Berg, and C. Ebling. Specifying and compiling applications for RaPiD. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, 1998. 86

[31] A. Deb, Josep M. Codina, and A. González. A Co-designed HW/SW Approach to General Purpose Program Acceleration using a Programmable Functional Unit. In *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architecture (INTERACT'11), held in conjunction with the 17th International Symposium on High-Performance Computer Architecture (HPCA'11)*, 2011. 12

[32] A. Deb, Josep M. Codina, and A. González. A HW/SW Co-designed Programmable Functional Unit. In *IEEE Computer Architecture letters (CAL'11)*, 2011. 12, 61

[33] A. Deb, Josep M. Codina, and A. González. A Power-Efficient Co-designed Out-Of-Order Processor. In *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'11)*, 2011. 12

[34] A. Deb, Josep M. Codina, and A. González. SoftHV : A HW/SW Co-designed Processor with Horizontal and Vertical Fusion. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*, 2011. 12

[35] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *IEEE Intl. Symp. on Code Generation and Optimization*, 2003. 17, 25, 50

[36] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96. ACM, 2009. 1

[37] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 2000. 90, 95

[38] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35:202–211, November 2000. 27

[39] K. Ebcioglu and E.R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *IEEE Intl. Symp. on Computer Architecture*, 1997. 8, 9, 21, 24, 25, 40, 46

[40] SSE extension. : Intel IA 64 and IA-32 Architectures Software Developer's Manual, 1997. 9, 82

[41] E. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully bypassed six-integer datapath and register file on the itanium-2 microprocessor. In *IEEE Intl. Journal of Solid-State Circuits*, 2002. 58, 66, 85

[42] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478 –490, july 1981. 27

[43] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, New York, NY, USA, 1983. ACM. 7

[44] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, 1998. 1

[45] S.C. Goldstein, H. Schmidt, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *IEEE Intl. Symp. on Computer Architecture*, 1999. 86

[46] M.K. Gowan. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*, 1998. 1, 6, 9, 108, 153

[47] M. Gschwind. Method and apparatus for determining branch address in programs generated by binary translation. *IBM Research Disclosures*, 1998. 20

[48] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber. The Machine Paradigm of Xputers and its Application to Digital Signal Processing Acceleration. In *Intl. Conf. on Parallel Processing*, 1990. 86

[49] J.R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a Reconfigurable Coprocessor. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, 1997. 86

[50] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, 1993. 1

[51] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousell. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 2001. 6, 31, 52, 120

[52] S. Hu, I. Kim, M.H. Lipasti, and J.E. Smith. An approach for implementing efficient superscalar CISC processors. In *IEEE Intl. Symp. on High-Performance Computer Architecture*, 2006. 9, 26, 79, 82, 86, 87, 90, 92, 93, 97, 98, 104, 125

[53] S. Hu and J.E. Smith. Reducing startup time in co-designed virtual machines. In *IEEE Intl. Symp. on Computer Architecture*, 2006. 15, 17, 20, 26, 46, 54

[54] Shiliang Hu and James E. Smith. Reducing startup time in co-designed virtual machines. In *In Proc. of the 33rd Annual International Symposium on Computer Architecture*, 2006. 20, 39

[55] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 1993. 16, 27

[56] AMD K8. Software Optimization Guide for AMD64 Processors. In *http://support.amd.com/us/Processor_TechDocs/25112.PDF*, 2005. 6, 31, 33, 52

[57] E. J. Kelly. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed, 1998. 21, 25

[58] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 Microprocessor Architecture. In *IEEE Intl. Conf. on Computer Design*, 1998. 6, 31, 52, 109

[59] H-S. Kim and J.E. Smith. Hardware support for control transfers in code caches. In *IEEE Intl. Symp. on Microarchitecture*, 2003. 20

[60] A. Klaiber. The technology behind Crusoe Processors, 2000. 8, 9, 10, 25, 54, 152

[61] Peter M. Kogge. An architectural trail to threaded-code systems. *IEEE Computer*, pages 22–32, March 1982. 26

[62] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, 1996. 5

[63] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, PACT '96, 1996. 7

[64] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John Ruttenberg. The multiflow trace scheduling compiler. *J. Supercomput.*, 7:51–142, May 1993. 27

[65] C. Madriles, C. Garcia-Quinones, J. Sanchez, P. Marcuello, A. Gonzalez, D.M. Tullsen, Hong Wang, and J.P. Shen. Mitosis: A speculative multithreaded processor based on precomputation slices. *Parallel and Distributed Systems, IEEE Transactions on*, 19:914 –925, 2008. 1

[66] S.A. Mahalke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Intl. Symp. on Microarchitecture*, 1992. 157

[67] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 77–84, 1998. 1

[68] J.F. Martinez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early recycling in out-of-order microprocessors. In *IEEE Intl. Symp. on Microarchitecture*, 2002. 152

[69] C. May. Mimic: a fast system/370 simulator. In *Papers of the Symposium on Interpreters and interpretive techniques*, SIGPLAN '87, pages 1–13, 1987. 26

[70] Scott McFarling. Combining branch predictors, 1993. 5

[71] Matthew C. Merten, Andrew R. Trick, and Ronald D. Barnes. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 2001. 15, 23, 46

[72] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, 2001. 153, 157

[73] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997. 40

[74] S. Oberman, G. Favor, and F. Weber. AMD 3Dnow! Technology: Architecture and implementations. *IEEE/MICRO*, 1999. 9, 82

[75] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3, September 2005. 1

[76] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *IEEE Intl. Symp. on Computer Architecture*, 1997. 1, 6, 9, 58, 66, 85, 108, 112, 113, 119, 122, 123, 153

[77] S.J. Patel and S.S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 2001. 22, 27, 37, 87

[78] J. Phillips and S. Vassiliadis. High performance 3-1 interlock collapsing ALUs. *IEEE Transactions on Computers*, 1994. 58, 85, 87, 90, 92, 94, 99, 104

[79] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 1999. 43

[80] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, 1994. 7

[81] R. Razdan and M.D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *IEEE Intl. Symp. on Microarchitecture*, 1994. 86, 87

[82] Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, and Avi Mendelson. Power awareness through selective dynamically optimized traces. In *IEEE Intl. Symp. on Computer Architecture*, 2004. 27, 83

[83] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, 1998. 86

[84] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, 1997. 5

[85] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 2011. 6

[86] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36:69–81, February 1993. 26

[87] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, 1981. 5

[88] J.E. Smith and R. Nair. *Virtual Machines: A Versatile Platform for Systems and Processes*. Elsevier Inc., 2005. 2, 8, 14, 19, 20, 23, 24, 25, 50, 155

[89] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609 –1624, 1995. 125

[90] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, 1995. 1

[91] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11, January 1967. 5, 6

[92] VMware Inc., Palo Alto, CA. *VirtualCenter VMware Virtual Center Users Manual, version 1.2*. 24

[93] M.V. Wilkes. The genesis of microprogramming. *IEEE Ann. Hist. Comput.*, 1986. 24

[94] D. Williamson. Arm cortex a8 : A high performance processor for low power applications. *http://www.arm.com/files/pdf/A8_Paper.pdf*. 5, 113

[95] M. J. Wing and G. P. D'Souza. Gated store buffer for an advanced microprocessor, 2000. 10, 22, 25, 53, 54, 112, 125, 152, 158

[96] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *IEEE Intl. Symp. on Computer Architecture*, 2000. 9, 79, 86, 87

[97] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, 1992. 5

[98] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *IEEE Intl. Symp. on Computer Architecture*, 2004. 9, 79, 86, 87

[99] M.T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2007. 29, 31