ABSTRACT

CHOUDHARY, NIKET K. FabScalar: Automating the Design of Superscalar Processors. (Under the direction of Dr. Eric Rotenberg.)

Superscalar processors are at the heart of many computing platforms – servers, desktop and laptop computers, and even cell phones. Moreover, superscalar processors have remained successful in general-purpose computing for many years because they exploit parallelism transparently. Several computer architecture trends suggest that now is the time to automate their development. These trends are (1) the growing interest in single-ISA heterogeneous multi-core processors, which are comprised of microarchitecturally diverse cores to attain higher performance and lower power consumption compared to a single generic core design, and (2) the demand for rapidly-designed, diverse, superscalar-based Application Processors (APs) in future mobile computing devices. A key barrier in the development of heterogeneous multi-core processors is the higher design and verification effort that comes with multiple core designs. Superscalar processor design automation helps in this respect, by raising the design abstraction to the level of diverse cores. The recent introduction of superscalar-based APs in smart phones and tablets is driven by increasingly complex software stacks (operating systems, virtual machines, just-in-time compilers, web browsers, and sophisticated “apps”). Superscalar processor design automation opens up APs to microarchitectural diversity within and across products, while maintaining rapid time-to-market.

This work proposes framing superscalar cores in a canonical form, so that it becomes feasible to quickly design many cores that differ in the three major superscalar dimensions: superscalar width, pipeline depth, and sizes of structures for extracting instruction-level parallelism (ILP). From this idea, we develop a toolset, called FabScalar, for automatically composing the synthesizable register-transfer-level (RTL) designs of arbitrary cores within a canonical superscalar template. The template defines canonical pipeline stages and interfaces among them. A Canonical Pipeline Stage Library (CPSL) provides many implementations of each canonical pipeline stage, that differ in their superscalar width and depth of sub-pipelining. An RTL generation tool uses the template and CPSL to automatically generate an overall core of desired configuration. Validation experiments are performed along three fronts to evaluate the quality of RTL designs generated by
FabScalar: functional and performance (instructions-per-cycle (IPC)) validation, timing validation (cycle time), and confirmation of suitability for standard ASIC flows. With FabScalar, a chip with many different superscalar core types is conceivable. Moreover, FabScalar makes sophisticated cores more accessible to designers and researchers, fueling more innovation.

Fabricating a core requires reducing its RTL description to a layout, a step called physical design. Arguably, physical design is a significant portion of overall chip design cost. We present a detailed physical design study of FabScalar-generated cores. In keeping with FabScalar’s virtue of increasing accessibility through automation, we make a point of heavily relying on automated synthesis and place-and-route (SPR).

Finally, this work proposes a new heterogeneous multi-core design strategy: design-effort alloy, or DEA. In DEA, the processor is comprised of multiple core types: one is the flagship superscalar core or high-effort (HE) core and other types are low-effort (LE) cores. The HE core is mandatory for commercial viability of a high-end processor: maintaining leading-edge and robust performance that can only be reliably achieved with a highly optimized generic core. The LE cores target the HE core’s inherent compromises on outlier program phases, accelerating these outliers to further boost single-thread performance where possible. Much less effort is expended on microarchitecture tuning and physical design of the LE cores, however, to ensure that the overall enterprise is profitable.
FabScalar: Automating the Design of Superscalar Processors

by

Niket K. Choudhary

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering
Raleigh, North Carolina
2012

APPROVED BY:

Dr. Gregory Byrd

Dr. James Tuck

Dr. Vincent W. Freeh

Dr. Eric Rotenberg
Chair of Advisory Committee
DEDICATION

To my parents...and wife Tulika...
BIOGRAPHY

Niket Kumar Choudhary was born in November, 1982, in Patna, India. He received the bachelor's degree in Information and Communication Technology from Dhirubhai Ambani Institute of Information and Communication Technology (DAIICT), India, in 2005, and the master's degree in Computer Engineering from North Carolina State University (NCSU), USA, in 2009. During his undergraduate studies he worked as an intern at Cadence Design System, Bangalore, in the logic synthesis group. After graduating from DAIICT, he held an engineering position at ARM Private Ltd, Bangalore (August 2005 - July 2007), in the processor division.

In Fall 2007, Niket started working towards his Ph.D. in Computer Engineering at NCSU, under the guidance of Dr. Eric Rotenberg. Niket’s research interests include computer architecture, heterogeneous multicore processors, 3D IC architecture design, dynamic binary translation and optimization, and emerging technologies and their interaction with architecture. He was a graduate intern at Microsoft Research (May 2011 - August 2011) and Intel (May 2010 - August 2010).
ACKNOWLEDGEMENTS

I believe graduate school for me was more like a roller-coaster ride than a marathon. Besides my patience and will-power, there has been constant support from countless people whom I offer my sincerest gratitude.

Foremost, I would like to thank my parents, Ajay and Neera, for instilling in me the value of higher education and giving me their unambiguous support throughout my life. I would like to thank my wife, Tulika, and my sisters, Pallavi and Parul, for their continuous encouragement and affection.

It has been a privilege to work with Dr. Eric Rotenberg, my advisor and a wonderful researcher. His passion for teaching and research has made immense impact on me. I have learned many things from him throughout the Ph.D. program – computer architecture, presenting and quantifying a research idea, and writing to name a few. Moreover, thanks to him for providing financial support for all the years.

I would like to thank Drs. Greg Byrd, James Tuck, and Vincent Freeh for serving on my committee and providing constructive comments to make this dissertation better.

During the Ph.D. program, I was fortunate to work with talented and hard-working fellow graduate students. Hashem Hashemi was instrumental in developing many key ideas related to this dissertation. It is still fun brainstorming with Hashem and I wish him good luck in his endeavors. This dissertation was part of a big research project involving multiple students. It was indeed a great pleasure working with Brandon Dwiel, Sandeep Navada, Salil Wadhavkar, Tanmay Shah, Jayneel Gandhi, and Hiran Mayukh. I will cherish and feel proud about this project for all my life. L’chaim to the FabScalar team!! Thanks to Elliott Forbes for organizing ABPs (Architecture, Beer, and Pizza). I always felt ABP was more about BP than A. Elliot was always a patient listener to my random ideas and a perfect critic as well. I would also like to thank other CESR and NCSU colleagues for giving their input and feedback: Muawya Al-Otoom, Ahmad Samih, Siddharth Chhabra, Mark Dechene, George Patsilaras, Rami Al Sheikh, and Rajeshwar Vanka.
Special thanks to Abhishek Dhanotia and Shivam Priyadarshi, my close friends since undergraduate days, for making life fun and smooth outside the lab. I wish them best wishes in their professional and personal life.

Summer internships were an escape from Raleigh and to explore other research projects in computer architecture. I would like to thank Doug Burger and Daniel Lavery for giving me an opportunity to work in their groups at Microsoft and Intel, respectively.

I owe a big thanks to Sandy Bronson, Linda Fontes, Elaine Hardin, and Kendall Del Rio for taking care of all my paperwork and conference reimbursements. Special thanks go to our IT staff, Dan Green and Brian Carty, who made sure all the required machines are up and running during conference deadlines. I would also like to thank Erica Cutchins for reviewing this dissertation and providing useful edits.

I lived in Western Manor apartment complex, part of university housing, for all my years in graduate school. I would like to thank the housing staff for being cooperative and helpful all the time.

This thesis was supported in part by NSF grant No. CCF-0811707, Intel, and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

List of Tables ................................................................. viii

List of Figures ................................................................. x

Chapter 1 Introduction ....................................................... 1
  1.1 Automating the Generation of Synthesizable RTL Designs ........... 3
  1.2 Leveraging Automated Physical Design Flow to Reduce Physical Design Effort ..................................................... 5
  1.3 Design-Effort Alloy for Optimizing Multi-core Efficiency .......... 6
  1.4 Thesis Contributions .................................................... 7
  1.5 Thesis Organization ...................................................... 9

Chapter 2 Background ....................................................... 11
  2.1 Technology Trend ....................................................... 11
  2.2 Multi-Core and Application Diversity ................................ 14
  2.3 Application Processors ............................................... 17
  2.4 Related Work .......................................................... 21
    2.4.1 Open-source Verilog Model ........................................ 21
    2.4.2 Analytical Approaches to Model Timing, Area, and Power .... 22
    2.4.3 Automating Processor Design ..................................... 23
    2.4.4 Asymmetric Multi-core Architecture .............................. 24
    2.4.5 Dynamic Multi-core Architecture ................................ 26
    2.4.6 Low-Effort Processor Design Methodology ....................... 26

Chapter 3 FabScalar .......................................................... 28
  3.1 FabScalar .............................................................. 28
  3.2 Canonical Superscalar Processor and the CPSL ....................... 29
  3.3 Validation Methodology ............................................... 33
  3.4 Validation Results .................................................... 36
    3.4.1 Functional and IPC Validation .................................... 37
    3.4.2 Timing Validation .................................................. 41
    3.4.3 Suitability for Standard ASIC Flows ............................. 44
  3.5 Extensibility of CPSL ................................................. 45

Chapter 4 Studying Performance and Efficiency Advantages of Employing Microarchitecturally Diverse Cores .......................... 46
  4.1 FabScalar-PPA Framework .............................................. 47
  4.2 Application Space .................................................... 49
  4.3 A Generic Heterogeneous Multi-Core ................................ 50
  4.4 Heterogeneous Multi-core Analysis .................................. 53
4.4.1 Performance and Efficiency Results ........................................ 56
4.4.2 Result Summary .............................................................. 64
4.5 Comparing Heterogeneous Multi-core with DVFS ....................... 66

Chapter 5 Time-to-Market Sensitive Superscalar Processor Design Approach ................................................................. 70
5.1 Evaluation Methodology ......................................................... 71
  5.1.1 Synthesizable Cores ...................................................... 71
  5.1.2 ASIC Design Flow ....................................................... 73
  5.1.3 Measuring IPC ........................................................... 75
5.2 Results .............................................................................. 75
  5.2.1 Baseline: minimum design effort ...................................... 76
  5.2.2 Optimizing memory structures ........................................ 78
  5.2.3 Adjusting the microarchitecture ...................................... 82
  5.2.4 Optimizing the circuit .................................................. 90
5.3 Putting it all together .......................................................... 92

Chapter 6 Design-Effort Alloy: An Approach for Optimizing Multi-core Efficiency .............................................................. 97
6.1 Designing a Low-Effort Core .................................................. 99
6.2 Designing a High-Effort Core ................................................ 101
  6.2.1 Modeling Frequency and Energy of High-Effort Core .......... 101
6.3 Results ............................................................................. 103

Chapter 7 Summary .................................................................... 109

References .................................................................................. 111

Appendix ..................................................................................... 120
  Appendix A Detail Design of CPSL ............................................. 121
    A.1 Design of Fetch ........................................................... 121
      A.1.1 Fetch-1 ................................................................. 122
      A.1.2 Fetch-2 ................................................................. 122
    A.2 Design of Decode .......................................................... 126
    A.3 Design of Rename ......................................................... 131
    A.4 Design of Dispatch ....................................................... 131
    A.5 Design of Issue ........................................................... 134
    A.6 Design of Register Read ................................................ 137
    A.7 Design of Execute ........................................................ 140
    A.8 Design of Load-Store Unit .............................................. 143
    A.9 Design of Writeback ...................................................... 143
    A.10 Design of Retire ........................................................... 147
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Commercial application processors and their microarchitecture configurations</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>EDA tools used for ASIC design flow</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Original and adjusted CACTI device parameters (45nm)</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Cores used for functional and IPC validation experiments</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Delay comparisons of commercial processors with similarly configured FabScalar generated cores</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>Power and area of FabScalar versions of the three commercial processors</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>The SPEC CPU2000 SimPoints</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Microarchitecture design space for studying advantages of heterogeneous multi-core processor</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Frequency, Area and Power of cores in the design space</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance (BIPS) gain of various N-core-type heterogeneous multi-core processor</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Best-performing core (BIPS) for each benchmark in the application space</td>
<td>57</td>
</tr>
<tr>
<td>4.6</td>
<td>Efficiency (BIPS²/Watt) gain of N-core-type heterogeneous multi-core processors</td>
<td>58</td>
</tr>
<tr>
<td>4.7</td>
<td>Best efficiency-oriented core (BIPS²/Watt) for each benchmark in the application space</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Efficiency (BIPS³/Watt) gain of various N-core-type heterogeneous multi-core processors</td>
<td>62</td>
</tr>
<tr>
<td>4.9</td>
<td>Best efficiency-oriented core (BIPS³/Watt) for each benchmark in the application space</td>
<td>65</td>
</tr>
<tr>
<td>4.10</td>
<td>Voltage-frequency pairs used for the core in Design-A</td>
<td>68</td>
</tr>
<tr>
<td>5.1</td>
<td>Microarchitecture configurations of two reference cores</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Major memory structures in reference cores</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Baseline results for Core-2W and Core-4W designs using 45nm cell library</td>
<td>76</td>
</tr>
<tr>
<td>5.4</td>
<td>Baseline results for Core-2W using 130nm cell library</td>
<td>77</td>
</tr>
<tr>
<td>5.5</td>
<td>Design effort for the different memory implementations of Rename, Register Read, and Issue</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>Experiments for pipelining timing critical stages</td>
<td>85</td>
</tr>
<tr>
<td>5.7</td>
<td>Issue Queue partitioning schemes in Core-2W and Core-4W</td>
<td>88</td>
</tr>
<tr>
<td>5.8</td>
<td>Results for Core-2W and Core-4W designs using optimized 45nm cell library</td>
<td>92</td>
</tr>
<tr>
<td>Table 5.9</td>
<td>Key for decoding the labels in Figure 5.11</td>
<td>94</td>
</tr>
<tr>
<td>Table 5.10</td>
<td>Key for decoding the labels Figure 5.12</td>
<td>96</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Low-effort core types used for evaluating Design-Effort Alloy approach</td>
<td>104</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>The preferred LE core type in Design-B yielding efficiency gains</td>
<td>108</td>
</tr>
<tr>
<td>Table A.1</td>
<td>Fetch-1 input/output signals</td>
<td>125</td>
</tr>
<tr>
<td>Table A.2</td>
<td>Fetch-2 input/output signals</td>
<td>127</td>
</tr>
<tr>
<td>Table A.3</td>
<td>Decode input/output signals</td>
<td>127</td>
</tr>
<tr>
<td>Table A.4</td>
<td>Instruction-buffer input/output signals</td>
<td>128</td>
</tr>
<tr>
<td>Table A.5</td>
<td>Rename input/output signals</td>
<td>132</td>
</tr>
<tr>
<td>Table A.6</td>
<td>Dispatch input/output signals</td>
<td>135</td>
</tr>
<tr>
<td>Table A.7</td>
<td>Issue input/output signals</td>
<td>137</td>
</tr>
<tr>
<td>Table A.8</td>
<td>Register Read input/output signals</td>
<td>140</td>
</tr>
<tr>
<td>Table A.9</td>
<td>Execute input/output signals</td>
<td>141</td>
</tr>
<tr>
<td>Table A.10</td>
<td>Load-Store input/output signals</td>
<td>147</td>
</tr>
<tr>
<td>Table A.11</td>
<td>Writeback input/output signals</td>
<td>148</td>
</tr>
<tr>
<td>Table A.12</td>
<td>Retire-ActiveList input/output signals</td>
<td>150</td>
</tr>
<tr>
<td>Table A.13</td>
<td>Retire-ArchMapTable input/output signals</td>
<td>151</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Canonical superscalar template.</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Impact of technology scaling on transistor performance and transistor integration growth</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Clock frequency of Intel processors over the generations of Intel’s process node</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Dynamic instruction-level characteristics of bzip benchmark</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Dynamic instruction-level characteristics of gcc benchmark</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Block diagram of system-on-chip used in current generation mobile devices.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Evolution of application processors used in TI OMAP SoCs.</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>High level view of FabScalar toolset</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Overview of canonical pipeline stage designs available in the CPSL</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Results of executing 100 million instruction SimPoints of six benchmarks on the twelve cores</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Physical design of a 4-way superscalar processor</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>FabScalar-PPA framework for measuring performance, power, and area</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Summary of performance (BIPS) gain</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Performance (BIPS) gain achieved by benchmarks on their optimal cores</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Distribution of benchmarks falling in different BIPS-gain ranges</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Summary of efficiency (BIPS²/Watt) gain</td>
<td>58</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Efficiency (BIPS²/Watt) gain achieved by benchmarks on their optimal core</td>
<td>59</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Distribution of benchmarks falling in different BIPS²/Watt-gain ranges</td>
<td>60</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Summary of efficiency (BIPS³/Watt) gains</td>
<td>62</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Efficiency (BIPS³/Watt) gain achieved by benchmarks on their optimal core</td>
<td>63</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Distribution of benchmarks falling in different BIPS³/Watt-gain ranges</td>
<td>64</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Performance gain comparison between heterogeneous multi-core and DVFS-enabled homogeneous multi-core</td>
<td>69</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>The Core-2W and Core-4W superscalar pipelines</td>
<td>73</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>IPCs and timing slack of Core-2W and Core-4W</td>
<td>78</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Superscalar processors form the heart of computing for many of the current personal computers and server platforms. Superscalar processors have remained successful for many years, compared to alternative execution models like Dataflow [30] and VLIW [25] architectures, because they exploit parallelism within a program transparently. Despite performance and transparency advantages, the design complexity of implementing a superscalar processor is enormous. Processor companies spend thousands of engineer-years for designing, verifying, and physically implementing a processor [35]. This is evident from the fact that the proliferation of superscalar has been primarily fed by a relatively small number of elite processor design teams. Furthermore, the widespread proliferation could be constrained by intensive design-effort required to design superscalar processors.

Over the years, academic and industry practitioners have advanced the performance and efficiency of superscalar processors, refined our understanding of them, and classified them. It might be possible that the design of superscalar processor can be fully automated. Automation will free engineers from designing complex superscalar processors and redirect their energy and creativity on using them in new ways. It will also put superscalar processor technology into the hands of more people so that they too can innovatively apply it in new ways. Moreover, there are two potential trends emerging in computing platforms that further necessitate the automation of superscalar processors:

- *Technology is driving specialization+diversity:* In past decades, transistor scaling delivered faster and lower-energy transistors at an exponential pace. During those years, it was reasonable to overlook the fact that a single generic microarchitec-
ture leaves some performance and power on the table for diverse program phases. With technology no longer delivering exponential efficiency gains, specializing the microarchitecture to instruction-level behavior is necessary to get the most performance and energy efficiency from silicon. To achieve an overall general-purpose processor, specialization must be coupled with diversity: providing sufficiently many specialized superscalar cores to cover arbitrary program behaviors. Heterogeneous multi-core has emerged as a response to providing microarchitecturally diverse superscalar cores on a single chip [52] [56] [65].

- **Lucrative markets demand quick-turn superscalar processors**: Smart phones, tablets, and other mobile devices are emerging as the new change agents in client computing. Their mobility, cloud connectivity, and integration of sensors, computation and communication have inspired unique applications unforeseen just a short time ago. Increasing software complexity has led to high-performance superscalar-based Application Processors (APs) being used in platform architectures, e.g., TI’s OMAP-5 has the ARM Cortex-A15 [40], Qualcomm’s Snapdragon uses Scorpion and soon Krait [37]. Mobile devices are a growth market that means a fast pace of introducing new products. The richness of future applications enabled by pervasive computing and both the technology-push and market-pull for richer user experiences will accelerate the pace of innovation. This makes time-to-market extremely important.

Both of these trends suffer from one common challenge. In the first trend, employing diverse superscalar cores in a heterogeneous multi-core increases design-effort significantly. Designing a single superscalar core is very expensive, therefore, designing many superscalar core types is prohibitively expensive and impractical. With respect to the second trend, short time-to-market coupled with the complexity of designing a superscalar processor makes low design-effort approaches imperative. *Automation of superscalar processor design is a potential solution to the common problem faced by these two trends.* In this thesis, we propose and explore a low-design-effort approach to implement a superscalar-based core. Although our approach is generally applicable, we use heterogeneous multi-core and time-to-market sensitive APs as the use cases.

The rest of this chapter introduces our approach to automate the generation of synthesizable register-transfer-level designs of arbitrary superscalar cores (Section 1.1), introduces the notion of reducing physical design-effort by relying on unabated transistor
miniaturization and decades of investment in CAD tools (Section 1.2), proposes a heterogeneous multi-core composed of high-effort and low-effort cores (Section 1.3), and finally enumerates the key contributions made by this thesis (Section 1.4).

1.1 Automating the Generation of Synthesizable RTL Designs

This thesis proposes framing superscalars in a canonical form, so that it becomes feasible to quickly design many cores that differ in the three major superscalar dimensions: superscalar width, pipeline depth, and sizes of structures for extracting ILP (frequency depends on these three). The canonical form is at the level of logical pipeline stages: fetch, decode, rename, dispatch, issue, etc. We call this a canonical superscalar processor and its logical pipeline stages are called canonical pipeline stages. Within this framework, all superscalar processors have the same canonical structure, i.e., each has a complete set of canonical pipeline stages and the same interfaces among them (shown in Figure 1.1). Where they differ is in the underlying implementations of their canonical pipeline stages. A Canonical Pipeline Stage Library (CPSL) is populated with multiple designs for each canonical pipeline stage. A specific superscalar processor can be composed by selecting one design for each canonical pipeline stage from the CPSL and stitching together a complete set of canonical pipeline stages. This composition is automated due to invariant interfaces among canonical pipeline stages and the confinement of microarchitectural diversity within the canonical pipeline stages. Finally, microarchitectural diversity is focused along key dimensions that both define superscalar architecture and differentiate individual superscalar processors. Namely, the different designs of a given canonical pipeline stage vary along three major dimensions:

1. **Superscalar complexity**: The superscalar complexity of a canonical pipeline stage is a product of its superscalar width (number of pipeline ways) and the sizes of its associated ILP-extracting structures (e.g., issue queue, physical register file, predictors, etc.). Increasing superscalar complexity may contribute to extracting more ILP in the program but typically increases the logic delay through the canonical pipeline stage. The effect of increasing logic delay on overall performance ultimately depends on the next differentiating factor.
2. **Sub-pipelining**: A canonical pipeline stage is nominally one cycle in duration, but may be sub-pipelined deeper to achieve a higher clock frequency.

3. **Stage-specific design choices**: Often there are multiple alternatives for handling certain microarchitectural issues, such as speculation alternatives, recovery alternatives, and so forth. These alternatives present a range of costs and benefits. Moreover, the costs and benefits often depend on specific instruction-level behavior in the program.

Our approach has been implemented in a novel toolset called FabScalar. FabScalar consists of a definition of the canonical superscalar processor, a CPSL containing many synthesizable register-transfer-level (RTL) designs of each canonical pipeline stage, and a tool for automatically composing the RTL designs of arbitrary superscalar cores by referencing the CPSL. To evaluate the quality of RTL designs generated by FabScalar, validation experiments are performed along three fronts: functional and performance (instructions-per-cycle (IPC)) validation, timing validation (cycle time), and confirmation of suitability for standard ASIC flows. Validation results, described in Chapter 3,
demonstrate that indeed a good quality RTL design can be generated using our proposed method.

The extensibility of the CPSL is important for proliferating microarchitecture diversity. While it is difficult to generalize about extensibility, the thesis presents a few examples in Chapter 3 that extend the CPSL to achieve better overall clock frequency or better IPC on different workloads.

FabScalar is a novel approach because it represents the first attempt to automate the design of superscalar processors. FabScalar addresses the design-effort problem by boosting designer productivity through generating RTL designs of whole cores. A quality RTL design is the essential starting point of the chip design cycle: the starting point for design tuning, verification, and physical design. With FabScalar, a chip with many diverse superscalar core types is conceivable with lower design effort. Moreover, FabScalar can help in proliferating soft superscalar cores (IP) to further shorten the design cycle for APs, and allow system manufactures to offer mobile devices at different performance/price points.

1.2 Leveraging Automated Physical Design Flow to Reduce Physical Design Effort

Traditionally, desktop- and server-class processors are implemented, starting from an initial RTL design, using semi-custom or full-custom approach [28] [35]. Superscalar processors in desktops and servers are aggressively optimized at the circuit level and in their floorplanning. Cycle-time critical circuits are optimized using high-performance dynamic logic and custom-designed memory structures [86]. Units are painstakingly placed and routed to optimize global wires. While custom design extracts the most performance out of silicon, it results in long design cycles and major design and verification effort (large design teams). Moreover, custom design may lower manufacturing yield (parametric yield loss), further increasing overall cost.

This thesis revisits the cost/benefit tradeoff of custom physical design of superscalar processors. In particular, the tradeoff for time-to-market sensitive superscalar designs
is explored. The context of this tradeoff has changed: (1) Frequencies of APs are not as high as desktop/laptop/server processors, due to form factor and power. (2) The intrinsic speed afforded by advanced technology can be leveraged to meet slightly less ambitious frequency targets. (3) Decades of investment in CAD has reduced the gap between automated and manual design. We explore the feasibility of heavily relying on automated synthesis and place-and-route (SPR), using RTL of several FabScalar-generated processors configured similarly to commercial APs. We explore a range of memory structure optimizations, microarchitecture adjustments, and circuit-level optimizations, in conjunction with SPR. The impact of each technique is carefully measured: its frequency contribution, its impact on instructions-per-cycle (IPC), power, and area, and its required design effort. This data enables constructing a frontier of combinations that minimize design effort for a given design quality. Chapter 5 presents detailed experiments and results.

1.3 Design-Effort Alloy for Optimizing Multi-core Efficiency

A significant amount of design effort goes into implementing a commercial flagship superscalar core. Logic partitioning to balance delay in pipeline stages, circuit-level optimization for IPC-critical stages, and routing global wires efficiently are some examples that stretch the design effort significantly. The effort is justified because of higher performance gains on a wide range of applications. Nonetheless, a single microarchitecture leaves some performance and efficiency on the table.

Alternatively, multiple diverse cores can be implemented using a high-effort flow to provide even larger performance and efficiency gains. However, the non-recurring engineering (NRE) cost of such a design would be insurmountable. This thesis proposes a new heterogeneous multicore design strategy: design-effort alloy, or DEA. In DEA, the processor is comprised of multiple core types: one is the flagship superscalar core or high-effort (HE) core and other types are low-effort (LE) cores. The HE core is mandatory for commercial viability: maintaining leading-edge and robust performance that can only be reliably achieved with a highly optimized single core, both its microarchitecture and physical design. The LE cores target the HE core’s inherent compromises on outlier program phases, accelerating these outliers to further boost single-thread performance where
possible. Much less effort is expended on microarchitecture tuning and physical design of the LE cores, however, to ensure that the overall enterprise is profitable. Notably, only crude floorplanning is performed, layout is achieved exclusively by automated SPR, and highly-ported memories are either fully synthesized or implemented by aggregating automatically generated non-custom SRAMs.

Chapter 6 presents detailed experiments and results.

1.4 Thesis Contributions

This thesis is part of a big project involving multiple graduate students, therefore, it is important to explicitly outline the contributions and co-contributions made by the author. In particular, major contributions are:

- The idea of framing superscalar processors in a canonical form to address the design-effort problem posed by single-ISA heterogeneous multi-core. This thesis is also the first work to bring solutions to bear on this problem.

- The FabScalar toolset which streamlines the design and verification of superscalar processors. The ability to automatically generate synthesizable RTL models of superscalar processors of different widths and depths is unprecedented, as these dimensions are not a simple matter of parameterization. Jayneel Gandhi contributed in pipelining the Fetch-1 stage [36], Hiran Mayukh contributed in pipelining the Issue stage [60], and Tanmay Shah contributed in pipelining the Register Read/Writeback stage [80].

- The development of a co-simulation environment in which a functional simulator written in C++ runs concurrently with verilog simulation of the superscalar processor. The two are independent and the functional simulator assists with checking and debugging the verilog simulation by comparing instructions’ results as they retire from the processor. The co-simulation environment also simplifies running standard benchmarks by not requiring full-system simulation.

- Validation of the quality of the generated RTL along three fronts: functional and IPC validation, cycle time validation, and suitability for standard ASIC flows.
• Exploration of an automated SPR flow to achieve acceptable quality physical design for time-to-market sensitive superscalar designs [22]. A range of memory structure optimizations, microarchitecture adjustments, and circuit-level optimizations, in conjunction with SPR are explored. The impact of each technique is carefully measured: its frequency contribution, its impact on IPC, power, and area, and its required design effort. This data enables constructing a frontier of combinations that minimize design effort for a given design quality. Brandon Dwiel assisted with memory structure optimizations [33].

• The idea of coupling a core designed using a traditional high-effort approach with cores designed using fully-automated (low-effort) approach, to deliver a lower NRE cost heterogeneous multi-core. The low-effort cores remove distinct microarchitecture bottlenecks in terms of IPC.

The author makes the following co-contributions:

• Using FabScalar, we proposed G21, a heterogeneous multi-core consisting of 21 diverse core types, that is not tuned to specific workloads but still achieves near-speediest execution on 59 SPEC SimPoints considering the entire design space [23]. The G21 design demonstrates that if the circuit-level tradeoffs are appropriately accounted for, then the design space for single-ISA heterogeneous multi-core can be reduced significantly for performance optimization only.

In addition, this thesis has the potential for large impact in computer architecture research:

• Technology-driven computer architecture research: Computer architecture research is increasingly driven by technology related problems (Moore’s law scaling, power, temperature, reliability, variability). Open-source synthesizable verilog of arbitrary superscalar processors is invaluable because it enables exploring architecture-technology interactions throughout the pipeline, enables sensitivity studies across different microarchitecture configurations, and increases result fidelity and detail.

• The next leap in processor evaluation methodology: FabScalar will enable the extensive use of whole-pipeline RTL models for processor research. While RTL modeling may seem constraining, constraints lead to discovery and innovation. It is enticing simply to revisit alternative processor architectures from past decades in the
context of RTL modeling; coupling these with heterogeneity offers even greater possibilities. This is not to mention the benefits of routinely quantifying the cycle time, area, and power costs/savings of new microarchitecture techniques. We will need a corresponding leap in RTL simulation, namely automatic mapping of arbitrary superscalar configurations to single FPGAs. The last several years have witnessed significant progress in this area. Recently, Dwiel et al. [33] developed a framework, referred to as \textit{FPGA-Sim}, that automates the mapping of FabScalar-generated cores to the FPGA. The FPGA prototypes are several orders of magnitude faster than verilog and C++ simulation.

- \textit{Architecture Research Prototyping}: What could be better than prototyping a new design, to demonstrate its feasibility and worthiness? Recently, many computer architecture research groups in academia have started focusing towards prototyping. The 3D heterogeneous processor project at NC State University is one such example. Academic groups may benefit from our work directly, if a FabScalar-generated core is being used for prototyping, or indirectly, by applying the techniques explored in this thesis.

In one year, the beta-release of FabScalar has been downloaded by researchers at over 20 universities and 3 industry sites (GlobalFoundries and two Intel labs), from 9 different countries. Multiple publications that use FabScalar integrally are appearing in various IEEE and ACM sponsored conferences.

### 1.5 Thesis Organization

The rest of this dissertation is organized as follows.

Chapter 2 provides background information on topics related to this thesis. It discusses process technology trends and their impact on computer architecture, single-ISA heterogeneous multi-core processors, and APs used in mobile devices. Finally, related work is thesis is discussed.

Chapter 3 describes FabScalar, including the canonical superscalar processor, the CPSL, and the validation experiments. It also introduces the evaluation methodology
used throughout this thesis.

Chapter 4 studies performance and computation-efficiency advantages of a heterogeneous multi-core compared to a homogeneous multi-core using real designs generated from FabScalar. The chapter describes our FabScalar-PPA framework that is used to estimate power consumption at the architecture level.

Chapter 5 explores a low design-effort approach to design superscalar processors for fast time-to-market. It provides detailed quantification of design-effort required to achieve an acceptable frequency using an automated SPR flow. The experiments described in this chapter forms the motivation for Chapter 6.

Chapter 6 presents a heterogeneous multi-core composed of high-effort and low-effort cores. The chapter describes the design methodology used for implementing high-effort and low-effort cores. Finally, the results are presented to demonstrate the benefits of Design-Effort Alloy based heterogeneous multi-core.

Chapter 7 summarizes the contributions of this thesis and outlines future work.
Chapter 2

Background

This chapter provides background information on topics related to this thesis. Section 2.1 discusses the trend of process technology and its impact on computer architecture. Section 2.2 provides an overview of heterogeneous multi-core processors and the design challenges faced by this paradigm. Section 2.3 presents a brief survey of commercial application processors being used in current mobile devices, discusses the complexity of their microarchitecture and discusses how the context for designing APs differ from desktop and server processors. Finally, Section 2.4 presents related work.

2.1 Technology Trend

Processors have experienced unprecedented growth in performance, owing to innovative microarchitecture and technology scaling. Technology scaling has been the performance growth enabler, providing abundant transistors to add more hardware for complex microarchitectures, and still leaving room to optimize clock frequency because of faster transistors. Over the years microarchitecture techniques like pipelining, superscalar processing, out-of-order execution, and branch prediction successfully exploited the doubling of transistor density every technology generation. To put technology scaling and microarchitecture innovation into perspective: since the inception of the Intel 4004 in 1971, the world’s first commercially available single-chip microprocessor, processor performance has increased 5,000-fold [26]. The transistor minimum feature size has scaled from 10,000nm in 1971 to 45nm in 2008 and transistor count on a single chip has increased from 2,300 in 1971 to more than one billion in 2008.
However, the joyride of performance scaling gained by technology scaling and conventional microarchitecture has hit a wall. Primarily, there are two reasons for this trend. First, technology scaling trends are compounding the problem: a transistor’s dimensions and its physical characteristics are not scaling as well as in the past. Second, the dizzying array of microarchitecture techniques for extracted instruction level parallelism (ILP) in the past has already become very complex, and the complexity for extracting additional ILP is enormous.

In the past, every generation of transistor scaling followed the scaling theory propounded by Dennard in 1974 [29]: transistors become faster and more energy-efficient every generation by reducing transistor dimensions by 30% and keeping the electric field constant everywhere in the transistor for reliability. Device engineers translated Dennard’s scaling theory into 1) $2 \times$ increase in transistor density, 2) $1.4 \times$ increase in clock frequency, and 3) constant dynamic power consumption every technology generation [13]. Unfortunately, the classical scaling ended in the early 2000s due to exponentially increasing leakage currents in transistors. A MOS transistor is fundamentally a voltage-controlled switch, and it has an intrinsic threshold voltage ($V_T$) for operation. Decreasing supply voltage ($V_{DD}$) every generation needs decreasing $V_T$ of CMOS. Unfortunately, reducing $V_T$ after a certain physical limit increases the off-current (referred to as leakage current) in CMOS significantly and also makes CMOS highly unreliable [91] [12] [39]. The limitation on scaling $V_T$ also limits scaling $V_{DD}$ [91]. Unfortunately, dynamic power quadratically depends on $V_{DD}$ (shown in Equation 2.1). In Equation 2.1, $\alpha$ is a switching activity factor and it only depends on the workload characteristics, $C$ is total load capacitance, $V_{DD}$ is supply voltage, and $f$ is clock frequency.

$$Power_{Dynamic} = \alpha * C * V_{DD}^2 * f$$ (2.1)

In the current technology scaling regime, transistor density and performance still increase every generation. Figure 2.1(a) shows the transistor performance for Intel CMOS technology [12] and Figure 2.1(b) shows the transistor integration growth over a decade of technology scaling for Intel processors [3]. Note that the higher drive current leads to higher CMOS performance. However, the un-scalability of $V_{DD}$ in the traditional way
Figure 2.1: Impact of technology scaling on (a) transistor performance and (b) transistor integration growth over the generations of Intel’s process node. Source for transistor performance graph: *The New Era of Scaling in an SoC World* by Mark Bohr [12].

has created a *power wall* for modern microprocessor designs. Increasing clock frequency further exacerbates the power wall – dynamic power increases linearly with clock frequency. Higher power consumption leads to severe thermal design issues and increases packaging cost significantly [83]. It is evident from Figure 2.2 that the clock frequency of the Intel family of processors has saturated around 3GHz to 3.5GHz in the last few years.

Currently, computation-efficiency is at the forefront of the design constraints in general-purpose microprocessors. The power wall has already made a disruptive impact on the computing industry in the last decade – major processor companies shifted to multi-core based designs from conventional monolithic superscalar designs [54]. Designers have
already started considering power/performance trade-offs at the microarchitecture definition level and the logic design level. Micro-op fusion (in Intel Core Duo [96]), clock gating, power gating, and dynamic voltage-frequency scaling (DVFS) are some techniques that are being employed to counter the increasing power consumption problem at different levels. Energy-proportional design metrics like $EDP$ (energy-delay product) or $ED^2P$ (energy-delay-square product) have become more important than purely performance-driven metrics.

### 2.2 Multi-Core and Application Diversity

Despite the leveling-off of single-thread performance due to the power wall, transistor density has been increasing by a similar rate as in the past (shown in Figure 2.1). This abundance of transistors led a major transition from single-core to multi-core microprocessors as a computing platform. Multi-core design is an alternative to scaling the complexity of a monolithic processor. Multi-core designs are more thermal-friendly because execution of applications is spread over an entire die. A multi-core microprocessor with each core having the same microarchitecture is referred to as homogeneous multi-core. With multiple cores on a die, multiple applications in a multiprogramming environment
can execute simultaneously, leading to an increased throughput of the processor [68]. Moreover, applications with abundant thread-level parallelism derive significant performance benefits from multi-core designs.

A single-core or homogeneous multi-core design has a generic microarchitecture, therefore, it is oblivious to the ILP diversity that exists within and across applications. Multi-core designs present an exciting opportunity to exploit diversity within and across applications by employing microarchitecturally diverse superscalar core types, called single-ISA heterogeneous multi-core or asymmetric multi-core [51] [90]. Program phases differ in their instruction-level characteristics: the amount and distribution of ILP and memory-level parallelism (MLP), branch predictability, cache locality, etc. Figure 2.3 and Figure 2.4 show the differences in four dynamic instruction-level characteristics, branch misprediction rate, average basic-block size, level-1 instruction cache miss rate, and level-1 data cache miss rate, within and across the bzip and gcc benchmarks, respectively. The characteristics have been measured for every interval of 100 million instructions. It is evident from the figures that the characteristics change significantly across different intervals within bzip and gcc. Moreover, the difference in characteristics is also evident across bzip and gcc. The branch misprediction rate of bzip is very high in many intervals compared to gcc, instruction cache miss rate of bzip is very low compared to gcc, and many intervals of gcc incur a very high data cache miss rate compared to bzip.

Performance/power metrics can be improved by matching instruction-level behavior to differently-designed cores. The core types may differ in their superscalar fetch/issue widths, pipeline depths, instruction scheduling (in-order or out-of-order), sizes of units involved in exposing ILP and MLP (issue queue, load and store queues, physical register file, reorder buffer, etc.), function unit mix, and sizes of predictors and caches. Customizing each core to an application, class of applications, or class of application behaviors will yield higher performance on individual applications, and overall execution will be power efficient.

Prior works in this area project significant performance and power advantages for microarchitecturally diverse superscalar cores [52] [56] [65] [90] [70]. Yet, no prior research has addressed a crucial drawback of this paradigm: design-effort is multiplied by the
Figure 2.3: Dynamic instruction-level characteristics of *bzip* benchmark (from SPEC2000). The characteristics have been plotted for every interval of 100 million instructions by simulating the benchmark for 10 billion instructions. (a) Branch misprediction rate, (b) Average basic block size, (c) Level-1 instruction cache miss rate, and (d) Level-1 data cache miss rate.

Figure 2.4: Dynamic instruction-level characteristics of *gcc* benchmark (from SPEC2000). The characteristics have been plotted for every interval of 100 million instructions by simulating the benchmark for 9.3 billion instructions. (a) Branch misprediction rate, (b) Average basic block size, (c) Level-1 instruction-cache miss rate, and (d) Level-1 data-cache miss rate.
number of different core types. This factor limits the amount of microarchitectural diversity that can be practically implemented. Moreover, because core customization captures the interplay between application, microarchitecture, and technology, it is imperative to faithfully consider the impact of superscalar core complexity on propagation delay, and its factoring into cycle time and pipeline depth. Propagation delay must be considered comprehensively: neglecting it not only makes it difficult to draw comparisons between different heterogeneous and homogeneous designs, it also diminishes the effect of customization by hiding the intrinsic performance advantage of customizing the trade-off between latency and parallelism within a pipeline.

2.3 Application Processors

The last decade has witnessed the rapid growth of smart phones, tablet PCs, and other mobile devices. These devices, in conjunction with existing cellular networks, provide very flexible and ubiquitous computation to users. The future holds even greater promise, as applications fuse diverse inputs to deliver new levels of situation awareness and embedded intelligence.

The general-purpose processors that power mobile devices are commonly referred to as application processors (AP). APs are capable of running operating systems and sophisticated consumer applications. Due to increasing software complexity and technology scaling, many of today’s commercial APs are in-order or out-of-order superscalar processors. Figure 2.5 shows a block diagram of a system-on-chip (SoC) used in a mobile device. The diagram is derived from the TI OMAP series of SoCs developed for mobile devices [40]. The SoC typically contains a module to interface with wireless antenna, digital signal processors (DSPs) for baseband processing, graphics accelerators for multimedia, and application processors for general-purpose computation. An interesting trend is the growing complexity of application processors. The graph in Figure 2.6 shows the evolution of the TI OMAP series starting from OMAP 1 introduced in 2004 to the recently announced OMAP 5. The ARM926, used in OMAP 1, is an in-order, single-issue, 5-stage pipeline, with a maximum reported clock frequency of 220 MHz. On the other hand, the Cortex-A15, used in OMAP 5, is an out-of-order, 3-way superscalar (dispatch is 3-way, the narrowest stage), 15-stage pipeline, with a maximum reported clock frequency of 2
Table 2.1 shows the microarchitecture configurations of four commercial application processors, the AMD Bobcat [18], ARM Cortex-A9 [2], MIPS 74K [63], and ARM Cortex-A15 [1]. The disruptive growth in application processor complexity in mobile devices has been fueled by increasing software complexity. A current generation mobile device is capable of running operating systems, web browsers, global positioning system based services, etc.

The context of APs is quite different from desktop- and server-class superscalar processors, different enough to warrant revisiting the cost/benefit trade-off of custom design of superscalar processors. Several factors suggest automated synthesis and place-and-route (SPR) may provide a better trade-off:

- **Time-to-market**: Mobile devices are a growth market, which also means a fast pace of introducing new products. The richness of future applications enabled by pervasive computing and both the technology-push and market-pull for richer user experiences will accelerate the pace of innovation. This makes time-to-market very important.

- **IP trends**: The licensing of soft IP of processors is a significant trend. Using soft
Table 2.1: Commercial application processors and their microarchitecture configurations. The configuration only reflects the integer pipeline. NA means information is not available in the literature. FU mix: S=simple ALU, C=complex ALU, B=branch, Ld/St=load/store pipeline. Note: Bobcat fetch width is reported in terms of bytes as Bobcat implements variable-length x86 ISA; its other widths are in terms of uops.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Width</td>
<td>32-bytes</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Issue Width</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Functional Unit Mix</td>
<td>2S/C, 1Ld, 1St</td>
<td>1S/C, 1S, 1Ld/St</td>
<td>1S/C, 1Ld/St</td>
<td>2S, 1C, 1B, 2Ld/St</td>
<td></td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>12</td>
<td>NA</td>
<td>12</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Issue Queue(s)</td>
<td>Int:16, Agen:8</td>
<td>NA</td>
<td>Int:8, Agen:8</td>
<td>Others:NA, Agen:16</td>
<td></td>
</tr>
<tr>
<td>Load / Store Queues</td>
<td>26 / 22</td>
<td>NA</td>
<td>8 / 8</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Phys. Register File Size</td>
<td>64</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>32</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>56</td>
<td>NA</td>
<td>32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>L1 I-cache / L1 D-cache (KB)</td>
<td>32 / 32</td>
<td>32 / 32</td>
<td>32 / 32</td>
<td>32 / 32</td>
<td>32 / 32</td>
</tr>
<tr>
<td>fetch-to-execute pipeline depth (simple / load-store)</td>
<td>13 / 15</td>
<td>8 / 11</td>
<td>13 / 14</td>
<td>14 / 17</td>
<td>14 / 17</td>
</tr>
<tr>
<td>Frequency (tech. node)</td>
<td>1.6GHz (40nm)</td>
<td>830MHz (65nm)</td>
<td>1.1GHz (65nm)</td>
<td>Up to 2GHz (28nm)</td>
<td></td>
</tr>
<tr>
<td>Area (mm2)</td>
<td>4.9</td>
<td>1.5 (no caches)</td>
<td>2.5 (1.7 no caches)</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Power (mW/MHz)</td>
<td>NA</td>
<td>0.482</td>
<td>0.65</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>
IP shortens the design cycle and reduces design effort by leveraging an existing microarchitecture and software ecosystem. At the same time, there is still flexibility to adjust the microarchitecture to tailor it for a specific product, improve performance or power, and integrate it into a specific system-on-chip. Soft IP and SPR are highly compatible. First, soft IP is often used by fabless companies using generic foundry-provided standard cell libraries and basic memory compilers. Second, extensive custom design counteracts the design-effort savings of using off-the-shelf IP. Custom design cannot be summarily discounted, nor should it be without careful consideration, but the soft IP trend does tend to dovetail with SPR.

- **Form factor**: The form factor of mobile devices limits their packaging and cooling technology. Consequently, frequencies of APs are not as high as desktop/server processors.

- **Closing the gap with technology and CAD**: In future technology nodes, frequency is constrained more by thermal/power limits than by the intrinsic speed of logic. What this means is that, faster logic afforded by new technology nodes can be used to recoup performance losses due to automation. Additionally, decades of investment in CAD has reduced the gap between automated and manual design.
2.4 Related Work

This section describes the related work for this thesis. We primarily focus on important research works that have made a constructive influence on this thesis.

2.4.1 Open-source Verilog Model

The Illinois Verilog Model (IVM) [95] provides the Verilog for a semi-parametrizable 4-issue, dynamically-scheduled superscalar processor. The processor executes a subset of the Alpha instruction set. Floating point instructions and other miscellaneous instructions are not implemented. The pipeline is 12 stages deep and includes microarchitectural features like a branch predictor and a memory dependence predictor. The detailed microarchitecture is described in [95]. The drawbacks of the current IVM are its unsynthesizable or poorly synthesizable (low frequency) verilog modules. More importantly, IVM’s superscalar width and pipeline depth are inflexible. These aspects are not easily parametrized and require FabScalar’s approach: an RTL generator that uses the canonical superscalar template and CPSL to compose a core of desired width and depth. Finally, FabScalar runs SPEC benchmarks out-of-the-box, and has been validated in terms of IPC, cycle time, and synthesizability via standard ASIC flows.

Sun’s OpenSPARC T1 and T2 are open-source Verilog models of the UltraSPARC T1 and T2, respectively [5]. Both designs are single-chip multi-threaded processor and implement the 64-bit SPARC V9 architecture. OpenSPARC T1 has eight homogeneous processor cores on the same die. Each core is an in-order, six-stage, scalar pipeline, and has hardware support to run four threads. OpenSPARC T2 also has eight homogeneous processor cores on the same die. Each core is an in-order, eight-stage, scalar pipeline, and has hardware support to run eight threads. Sun’s open-source Verilog models also include on-chip memory hierarchy, including on-chip DRAM controller. OpenSPARC T2 has 4 Mbyte shared L2 cache (banked eight ways) and cores are connected to L2 cache through a crossbar. The OpenSPARC memory hierarchy complements FabScalar by enabling full system-on-chip integration of complex superscalar cores with its validated cache system.

OpenRISC 1200 (OR1200) is an open-source Verilog model of a processor available from OpenCores [4], and it implements the 32-bit ORBIS32 architecture. OR1200 is a five-
stage, in-order, scalar pipeline. Although, OpenSPARC and OR1200 are freely available Verilog models of processors, they do not represent the complexity of an out-of-order and superscalar microarchitecture.

2.4.2 Analytical Approaches to Model Timing, Area, and Power

Analytical approaches to model timing, area, and power of superscalar processors and caches has been an active area of research for many years. Analytical models enable computer architects to quickly evaluate the impact of different design choices on propagation delay, area overhead and power consumption. Wada et al. [94] developed an analytical access time model for on-chip cache memories. The model includes general cache parameters such as cache size, block size, and associativity. The model also provides knobs for subarray configurations within the physical RAM array organization. Wilton and Jouppi [98] refined the model developed by Wada to more closely represent real on-chip caches. Wilton’s model (referred to as CACTI) includes the tag array missing in Wada’s model and provides better transistor models for estimating delay. Reinman and Jouppi [72] augmented the original CACTI model to include cache power modeling, fully-associative caches, and technology scaling. The most recent release of the CACTI model supports SRAM and DRAM based caches as well as plain memory arrays [73]. It uses device models based on the industry-standard ITRS roadmap [8], using MASTAR [8] to calculate device parameters at different technology nodes. FabScalar uses CACTI 5.0, modified to FreePDK 45nm technology, for estimating timing, area, and power of level-1 and level-2 caches.

Palacharla, Jouppi, and Smith [69] developed models for estimating propagation delays of key superscalar pipeline stages: rename, issue, and bypasses. Palacharla’s models use issue width and scheduling window as primary drivers for modeling delays. Brooks, Tiwari and Martonosi [14] developed Wattch, an analytical approach to estimate power consumption at the microarchitecture level. Wattch uses capacitance models of memory arrays and microarchitecture components from CACTI and Palacharla’s models, respectively. Further, it obtains switching events from a microarchitectural simulator to calculate dynamic power consumption. Recently, Li et al. describe a comprehensive power, area, and timing modeling framework for multi-core systems, McPAT [58]. The processor timing models extend Palacharla’s approach to multiple microarchitectural styles, and
memory arrays are modeled based on CACTI models.

FabScalar extends delay and power modeling to other critical pipeline stages such as instruction fetch, arbitrary core logic, and the whole core; it considers sub-pipelining and its imbalances; and it produces RTL implementations of cores. FabScalar’s RTL output underscores a crucial distinction with computer architecture tools: the goal of FabScalar is to streamline the design, verification, and fabrication of chips, i.e., it is meant to serve as a development tool for designing heterogeneous multi-core chips, not just an estimation tool for research.

2.4.3 Automating Processor Design

Strozek and Brooks developed a framework for high level synthesis of very simple cores for embedded microcontrollers [88]. The framework is used to generate Verilog models of custom architectures for sensor and multimedia applications. The architecture is based on simple instruction sets, e.g., MIPS R2000 or ARM7, implemented on unpipelined or minimally pipelined microarchitectures. The framework uses seven microarchitecture parameters, for example number of ISA registers, use of complex unit, and datapath width, to customize and generate the Verilog.

The Program-In-Chip-Out (PICO) framework out of HP labs [45] is closely related in that it customizes VLIW cores and non-programmable accelerators for embedded applications. The PICO framework consists of four key elements: 1) the pre-determined template that defines architectural parameters and set of rules to connect various modules, 2) the spacewalker that traverses the design space, created by different combinations of architecture parameters, to find the Pareto-optimal designs, 3) the constructor that builds a detailed design described in Verilog/VHDL using the architecture parameters provided by the spacewalker, and 4) the evaluator that measures the cost of a design in silicon area and performance in cycles to execute the application.

Tensilica’s Xtensa configurable processors, Xtensa 9 and Xtensa LX4, automate the designer’s task of customizing instructions, functional units, and even VLIW datapaths [7]. The primary target for Tensilica’s processors is digital signal processing (DSP) based applications, e.g., voice and video processing.
A large body of work exists that attempted to synthesize hardware directly from high-level programming language, such as C/C++ [87] [78] [50] [84]. The automatic mapping of C/C++ models to hardware can simplify the design and verification process significantly. Most works both extended and restricted programming language constructs to enable automation. The extensions are required to add concurrency (hardware modules explicitly execute in parallel) and structural information (data formats, input-output). The restrictions are required to avoid constructs that are extremely hard or impossible to translate into hardware, for example, pointers.

FabScalar is unique in that it generates complex superscalar processors and this is evident in the novel composable CPSL.

2.4.4 Asymmetric Multi-core Architecture

Kumar et al. [51] proposed single-ISA heterogeneous multi-core architecture as an approach to reduce power dissipation. The proposed architecture includes different generations of Alpha cores: EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) and EV8 (Alpha 21464). These cores represent different points in the power/performance design space, and an application, based on its performance and power objectives, is dynamically scheduled on the most appropriate core to execute. This work primarily targeted a single-threaded execution environment. Follow-on work by Kumar et al. [53] demonstrate that, for a fixed area budget, a multi-core consisting of complex cores and simpler cores yields better throughput compared to a homogeneous multi-core, in a multiprogramming environment.

The seminal work on core selection by Kumar et al. [52] proposed that for a fixed power and/or area budget, a chip multiprocessor with each core customized to a different set of application characteristic, yields the highest performance. However, the work tacitly assumed all core types have the same cycle time and pipeline depth regardless of their complexity. In this case, the most aggressive core (widest pipeline, largest ILP-extracting units, largest predictors and caches, etc.) will be perceived to be the highest-performing core for all applications. This was borne out in one of their results. The throughput-per-area advantage of heterogeneity diminished for multi-core designs with sufficiently
large power and area budgets: simply replicating the most aggressive core maximized performance. This would not necessarily be the case if customization also tailored the trade-off between pipeline parallelism and propagation delays of pipeline loops, i.e., the IPC/frequency trade-off.

Suleman et al. [90] demonstrated that a chip multiprocessor consisting of a few complex cores (4-wide out-of-order) and many simple cores (2-wide in-order) provides significant performance benefits to multi-threaded applications. The serial parts of the multi-threaded application, such as critical sections, execute on a complex core to reduce the performance impact of serialization, and the parallelized parts execute on many simple cores to obtain high throughput. Najaf-abadi, Choudhary, and Rotenberg [65] proposed core-selectability in chip multiprocessors for accelerating multi-threaded applications. Each processing node in a chip multiprocessor consists of a cluster of multiple differently-designed cores, with the option to dynamically select which core to actively employ. Differently-designed cores target different ILP behavior in a multi-threaded application.

A closely-related body of research is design space exploration for designing heterogeneous multi-core processors. The design space of differently-designed cores can grow significantly (in the millions of design points) with only a modest number of microarchitectural parameters, such as cache size, superscalar width, ROB size, issue queue size, etc. Exhaustively searching the design space for the optimal design for an application is impractical. Several proposals have been made recently to accelerate this search. Most of these proposals belong to two categories: regression analysis techniques and classical search/optimization techniques. Regression analysis involves formulating regression models for microarchitectural performance and power prediction, and applying these models to quickly evaluate the quality of a design point for an application [44] [57] [42]. Classical search/optimization involves applying machine learning techniques, such as genetic algorithms, simulated annealing, ant colony optimization, etc., to the design space search problem [43] [67].
2.4.5 Dynamic Multi-core Architecture

Albonesi’s seminal work on *Complexity-Adaptive Processors* (CAPs) recognized that dynamically trading off IPC and clock rate achieves significant single-thread performance [9]. The work dynamically reconfigures on-chip caches and the issue queue to match the application needs at run-time. Moreover, the work argues that the traditional wire buffering mechanism (repeaters) can be exploited to convert critical hardware structures into configurable ones with little or no cycle time impact. Folegnani and Gonzalez proposed a reconfigurable issue stage design for energy savings [34]. Lee and Brooks [56] evaluated the potential of comprehensive microarchitectural adaptivity, varying pipeline complexity (superscalar width, sizes of ILP-extracting units, sizes of caches, etc.) and pipeline depth within and across applications. Depth was varied in terms of the number of FO4 inverter delays between pipeline latches: presumably the total FO4 delay of the pipeline is known and this delay is sliced finer for deeper pipelines.

*Core Fusion* and *TFlex* are reconfigurable multi-core architectures proposed by Ipek et al. [41] and Kim et al. [46], respectively. Both proposals envision a chip multiprocessor consisting of many simple, low-power dual-issue cores, and the cores can be dynamically composed into a larger core. The hardware resources of independent cores, such as caches, instruction queues, etc., are aggregated to form a single contiguous instruction window. A composed, large core exploits ILP for single-thread performance and many simple cores exploit thread-level parallelism. Core Fusion is based on conventional RISC/CISC ISAs and Tflex is based on the EDGE ISA [17].

A dynamic multi-core architecture potentially exploits a finer granularity of application diversity than asymmetric multi-core. However, the overhead of adding reconfigurability in timing/power critical hardware resources may substantially diminish the benefits of a dynamic multi-core architecture.

2.4.6 Low-Effort Processor Design Methodology

Chinnery and Keutzer [21] study and address the power and performance gap between ASIC and full-custom design methodologies. They recommend techniques such as logic pipelining, alternate algorithmic implementations of logic cells, using dual supply vol-
Bazeghi et al. [11] propose a methodology using regression models to quantify the design effort in the RTL implementation and verification phase in the processor development process. Using their methodology, the design effort of different components of a processor can be estimated, allowing design teams to allocate commensurate resources to the more effort-intensive components. However, their study does not account for physical design effort.

Kim et al. [47] propose designing processor systems out of pre-designed bricks. Bricks are small logic blocks that are independently designed and verified, and they can be assembled to implement a functioning system. Their approach reduces design and verification effort since the functionality of each brick type needs to be verified (and optimized) only once.

The POWER7 is the latest processor of the IBM’s POWER family, and it contains eight simultaneous multithreaded (4-way SMT) cores. The designers of POWER7 make a strong emphasis on enabling higher design efficiency by automating the layout of regular datapaths and memories through the use of Cadence SKILL scripts, replicating memories as a way to extend the number of read and write ports with low custom-design effort, and relying heavily on a novel next-generation synthesis methodology that utilized an extensive multidimensional cell library [35].

The Bobcat is a simple, low-power x86 core designed by AMD for mobile and lower-end desktop markets. The designers of Bobcat rely heavily on automatic synthesis, place and route for the physical design [18]. All memory structures, excluding caches, TLBs, branch predictors, and microcode ROMs, were implemented using flip-flops.
Chapter 3

FabScalar

This chapter presents FabScalar. Section 3.1 describes the three key components of the FabScalar toolset. Section 3.2 details the canonical superscalar processor and existing CPSL implementation. Section 3.3 presents the evaluation methodology. Section 3.4 presents the validation results and discusses the quality of RTL generated by FabScalar. Section 3.5 presents the extensibility aspect of the CPSL.

3.1 FabScalar

This thesis proposes an approach to automating the generation of RTL designs of arbitrary superscalar processors within a canonical template, to address the design-effort problem. Our approach has been implemented in a novel toolset called FabScalar. As shown in Figure 3.1, FabScalar consists of:

- a *Canonical Superscalar Template*, which defines canonical pipeline stages,
- a *Canonical Pipeline Stage Library (CPSL)*, which contains many different synthesizable RTL designs of each canonical pipeline stage, and
- a *Core Generator* for automatically composing the RTL designs of arbitrary superscalar cores by referencing the CPSL.

The canonical superscalar template defined by FabScalar consists of nine canonical pipeline stages: Fetch, Decode, Rename, Dispatch, Issue, Register Read, Execute, Writeback, and Retire. The CPSL provides many different designs for each canonical pipeline
Figure 3.1: FabScalar toolset. All processors generated by FabScalar have the same canonical superscalar template. The CPSL contains different flavors of each canonical pipeline stage.

stage that differ in major superscalar dimensions. Finally, automation is enabled because of invariant interfaces among canonical pipeline stages and confinement of microarchitectural diversity within the canonical pipeline stages.

3.2 Canonical Superscalar Processor and the CPSL

The canonical superscalar processor can represent either the MIPS R10K style of microarchitecture (issue queue instead of reservation stations, unified physical register file that is read after issuing, active list form of reorder buffer, etc.) [99] or the Tomasulo-style of microarchitecture [93] (reservation stations with data, reorder buffer with data, separate architectural register file, etc.). These two styles can be supported in the Canonical Pipeline Stage Library (CPSL) via different versions of the dispatch, issue, and register read stages. For the former style, register values exist and are obtained only in the Register Read stage and there is a single physical register file for both retired and speculative
values. For the latter style, available register values are obtained from a separate architectural register file and reorder buffer in the Dispatch stage, available values are captured by reservation stations in the Issue stage, and the Register Read stage is for reading values from the reservation stations upon issuing the instruction. Moreover, hybrids of these two styles are not precluded by the canonical superscalar processor. Currently, the CPSL is only populated with versions for the MIPS R10K style of microarchitecture.

The CPSL instruction-set architecture (ISA) is PISA [16], a close derivative of the MIPS ISA (minus load and branch delay slots). The rationale for using a simple RISC ISA is three-fold: (1) As a practical matter, our primary experience is with MIPS. (2) Contemporary processors often dynamically transform the binary level ISA into an implementation ISA sometimes referred to as micro-operations. Micro-operations resemble RISC primitives. One can view the chosen CPSL ISA as a canonical implementation ISA. Supporting different register state between a binary level ISA and an implementation ISA is a more difficult issue that may require software binary translation technology [27]. (3) A simpler implementation ISA results in a smaller core.

Since highly-ported RAMs and CAMs are prevalent in superscalar processors and significantly impact area, power, and cycle time, the thesis uses FabMem [80], a tool for automatically generating the physical designs (layouts) of multiported RAMs and CAMs.

The CPSL is populated with many synthesizable RTL designs for each canonical pipeline stage, that differ in their superscalar width, depth of sub-pipelining, and sizes of structures. Figure 3.2 summarizes the microarchitectural diversity available in the current CPSL. The first column identifies the canonical pipeline stage. The second column shows ranges of width and depth. All front-end stages (Fetch through Dispatch) and the Retire stage vary from 1-way to 8-way superscalar. The minimum width of all back-end stages (Issue through Writeback) is currently 4 because at least four different function units (FUs) are required: one each of simple ALU, complex ALU, load/store port, and branch unit. Narrower issue widths can be accommodated by aggregating multiple FU types into one pipeline way. The maximum width of all back-end stages is 8-way superscalar.
The second column in Figure 3.2 also shows ranges of depth of sub-pipelining. Sub-pipelining was guided by natural logic boundaries within each canonical pipeline stage design and timing results from synthesis (including RAM and CAM macros generated by FabMem). Dispatch has only 1-deep implementations in the CPSL and Retire has only 2-deep implementations. All other stages have a range of depths. Fetch goes the deepest, ranging from 2-deep to 5-deep. This is a result of the fetch unit having substantial logic in two phases, Fetch-1 and Fetch-2. Fetch-1 corresponds to accessing all the structures (2-way interleaved instruction cache, branch target buffer, branch predictor) and it also implements the complex next-PC logic. Fetch-2 corresponds to all logic after the instruction cache for extracting the fetch block from two cache blocks and aligning it for the decode stage, including branch predecode logic. Thus, Fetch is at least 2-deep: 1-deep for each of Fetch-1 and Fetch-2. A 2-deep version of Fetch-1 was designed and features block-ahead prediction [79], a somewhat elaborate approach for effectively pipelining the branch prediction logic. To our knowledge, this may be the first RTL implementation of it. The Decode and Rename stages are separated by an instruction buffer (the Fetch Queue), which facilitates assembling the full superscalar width of instructions for the Rename and Dispatch stages. The Decode stage can be as deep as 3-deep due to the logic complexity for steering and writing instructions into the buffer, including cracking doubleword instructions into two micro-ops. The Rename stage varies from 1-deep to 3-deep, with deeper versions increasing the amount of hazard logic for dependencies among consecutive rename bundles in-flight in the Rename stage. The CPSL provides four sub-pipelined implementations of the Issue stage:

- **1-cycle issue / 1-cycle loop (1/1):** 1 cycle for the Issue stage as a whole (i.e., no sub-pipelining), including wakeup, select, and reading the payload RAM. 1 cycle for the critical wakeup+select loop.

- **2-cycle issue / 2-cycle loop (2/2):** 2 cycles for the Issue stage as a whole and 2 cycles for the critical wakeup+select loop. Thus, the logic partitions are: (a) wakeup and (b) select+payloadRAM.

- **2-cycle issue / 1-cycle loop (2/1):** 2 cycles for the Issue stage as a whole but only 1 cycle for the critical wakeup+select loop. Thus, the logic partitions are: (a) wakeup+select and (b) payloadRAM. (An optimization makes the selected instruction’s tag available prior to reading the payloadRAM, decoupling it from the
wakeup-select loop.)

- 3-cycle issue / 2-cycle loop (3/2): 3 cycles for the Issue stage as a whole but 2 cycles for the critical wakeup+select loop. Thus, the logic partitions are: (a) wakeup, (b) select, and (c) payloadRAM.

The Register Read stage ranges from 1-deep to 4-deep. Based on experience with FabMem, we pipeline the wordlines of the Physical Register File, i.e., a sub-word of the register is returned each cycle. Due to pipelining the wordlines, Writeback takes the same number of cycles (write one sub-word per cycle). A nice feature is that the canonical interfaces for bypass buses are invariant with the degree of sub-pipelining: a value is bypassed to canonical pipeline stages Execute and Register Read irrespective of their underlying implementations, and a bypassed value is steered appropriately to one or multiple sub-stages within Register Read.

The third column in Figure 3.2 shows stage-specific structures that are implemented in the RTL. For simulation of the RTL, RAM and CAM structures are implemented with behavioral modules but all of the associated control and datapath logic surrounding them are implemented with synthesizable RTL. For synthesis and place-and-route, the RAM and CAM behavioral modules are replaced with macros generated by FabMem. The L1 instruction and data caches are omitted from Figure 3.2 because the L1 cache controllers are not yet implemented in RTL and are left for future work. Sizes of all stage-specific structures are parameterized in the RTL: since sizes can take on arbitrary values, no ranges are specified for structures in the third column in Figure 3.2.

The final column in Figure 3.2 considers another dimension for microarchitectural diversity, which we refer to broadly as microarchitectural approaches. This dimension is a potpourri of design choices specific to each canonical pipeline stage. It is outside the scope of this thesis to cover all of these techniques in the CPSL, at the level of synthesizable RTL. Nonetheless, we felt it would be of interest to enumerate notable examples in Figure 3.2 to emphasize the potential for growing the CPSL in the future, and to underscore the specificity with which microarchitectural diversity can be targeted to specific instruction-level behavior. For example, certain program phases will favor one branch misprediction recovery strategy over another depending on the frequency of mispredicted branches, their distribution, and the criticality of their backward slices. As
Table 3.1: EDA tools used for ASIC design flow.

<table>
<thead>
<tr>
<th>Phase</th>
<th>EDA tool(s) used</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional verification</td>
<td>Cadence NC-Verilog, vers. 06.20-s006</td>
</tr>
<tr>
<td>logic synthesis</td>
<td>Synopsys Design Compiler, vers. E-2010.12-SP2</td>
</tr>
<tr>
<td>place &amp; route</td>
<td>Cadence SoC Encounter, vers. 7.1</td>
</tr>
<tr>
<td>spice simulation</td>
<td>HSPICE, vers. C-2009.03-SP1</td>
</tr>
</tbody>
</table>

another example, techniques that are of no use for subsets of the workload space can be excluded from a core to streamline its frequency and static and dynamic power. Specific design choices that are represented in the current CPSL are highlighted in boldface in the last column in Figure 3.2.

### 3.3 Validation Methodology

Table 3.1 shows the EDA tools used for functional verification, synthesis, and place-and-route. For synthesis, we used the Nangate 45nm open cell library [49]. The Nangate library is based on the FreePDK BSIM4 predictive technology model [85].

Since specialized, highly-ported RAMs and CAMs are so pervasive and essential to a superscalar processor, we use FabMem for generating their physical layouts and extracting timing, power, and area. It is similar in concept to a memory compiler. For synthesis, RAMs and CAMs are encapsulated in the synthesizable verilog as macros. For simulation, they are represented with behavioral modules.

Custom RAM and CAM macros are used for the rename map table, architectural map table, active list, free list, fetch queue (separates the decode and rename stages), issue queue wakeup CAM and payload RAM, physical register file, load queue CAM and RAM, and store queue CAM and RAM.

The level-1 (L1) instruction and data caches, branch target buffer (BTB), and conditional branch predictor are abstracted as macros, with timing information obtained from CACTI 5.1 [73]. CACTI uses device and wire parameters derived from ITRS (refer to Tables 4 and 6, respectively, in the CACTI 5.1 report [73]). The BSIM4 Predictive
<table>
<thead>
<tr>
<th>Canonical pipeline stage</th>
<th>Dimensions (W=width, D=depth)</th>
<th>Stage-specific structures (sizes parameterized in RTL)</th>
<th>Microarchitectural approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>W = 1 to 8, D = 2 to 5</td>
<td>Branch prediction history table (BPH) or BHT</td>
<td>Branch prediction algorithm</td>
</tr>
<tr>
<td></td>
<td>Fetch.1: 1 or 2 sub-stages</td>
<td>Branch target buffer (BTB)</td>
<td>No interwoven vs. 2-way interleaving</td>
</tr>
<tr>
<td></td>
<td>Fetch.2: 1 to 3 sub-stages</td>
<td>Return address stack (RAS)</td>
<td>Block-based BTB vs. interleaved BTB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1 instruction Cache</td>
<td>Multi-cycle fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unpipelined</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pipeline using branch-ahead prediction</td>
</tr>
<tr>
<td>Decode</td>
<td>W = 1 to 8, D = 1 to 3</td>
<td>Fetch queue</td>
<td>Micro-operation encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-speculative vs. speculative decode (if variable length ISA)</td>
</tr>
<tr>
<td>Rename &amp; Resolve</td>
<td>W = 1 to 8, D = 1 to 3</td>
<td>Rename map table (RMT)</td>
<td>Architectural map table (AMT)</td>
</tr>
<tr>
<td></td>
<td>W = 1 to 8, D = 2</td>
<td>Architectural map table (AMT)</td>
<td>ARM vs no AMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Shadow map tables: 0 or 4</td>
<td>Branch miss detection recovery</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Free list</td>
<td>checkpoint (shadow map)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active list</td>
<td>handle like exceptions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phys. regs ready bit table</td>
<td>walk active list forward from head</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>walk active list backward from tail</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Exception recovery</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>restore RMT using AMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read new RMT by walking active list backward</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>freeing registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read prev mapping from RMT, active list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>push to RMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read prev mapping from AMT, AMT pushes free list</td>
</tr>
<tr>
<td>Dispatch</td>
<td>W = 1 to 8, D = 1</td>
<td>Issue queue (IQ) free list</td>
<td>Collapsing IQ vs. freelist-based IQ</td>
</tr>
<tr>
<td>Issue</td>
<td>W = 4 to 8, D = 1 to 3</td>
<td>Issue queue (IQ)</td>
<td>Collapsing IQ vs. freelist-based IQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Multiple schedulers vs. single scheduler</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Pipelined wakeup test</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-cycle produces non speculatively wakeup dependents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-cycle produces speculatively wakeup dependents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Load instructions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>predict hit always</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>predict miss always</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>hit predictor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Load SQ conflict (with unknown store address):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>predict no SQ conflict always</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>predict SQ conflict always</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>memory dependence predictor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recovery for miss: wake up &amp; load conflict spec.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>replay from IQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>replay from replay buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>handle like exception (squash)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Split stores</td>
</tr>
<tr>
<td>RegisterRead</td>
<td>W = 4 to 8, D = 1 to 4</td>
<td>Physical register file</td>
<td>n/a</td>
</tr>
<tr>
<td>Execute</td>
<td>W = 4 to 8, D = FU specific</td>
<td>Load queue (IQ)</td>
<td>Store-load forwarding vs no forwarding</td>
</tr>
<tr>
<td></td>
<td># simple ALU: 1 to 3, D = 1</td>
<td>Stone queue (SQ)</td>
<td>Many IQ/DSQ designs possible for reducing associative searches (SLQ, SDW, SQF)</td>
</tr>
<tr>
<td></td>
<td># complex ALU: 1, D = 3</td>
<td>L1 Data Cache</td>
<td></td>
</tr>
<tr>
<td></td>
<td># load/store ports: 1, D = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td># branches until: 1, D = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watermark/Bypass</td>
<td>W = 2 to 8, D = matches RegisterRead</td>
<td>n/a</td>
<td>Full bypasses vs. hierarchical or partial bypasses</td>
</tr>
</tbody>
</table>

Figure 3.2: Overview of canonical pipeline stage designs available in the CPSL.
Table 3.2: Original and adjusted CACTI device parameters (45nm).

<table>
<thead>
<tr>
<th>Device Parameter</th>
<th>HP/LSTP/LOP ITRS Model</th>
<th>BSIM4 Predictive Technology Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lgate (nm)</td>
<td>18/28/22</td>
<td>22.6</td>
</tr>
<tr>
<td>EOT (equiv. oxide thickness) (nm)</td>
<td>0.65/1.4/0.9</td>
<td>1.2</td>
</tr>
<tr>
<td>VDD (V)</td>
<td>1/1.1/0.7</td>
<td>1.1</td>
</tr>
<tr>
<td>Vth (mV)</td>
<td>181/532/256</td>
<td>400</td>
</tr>
<tr>
<td>I\text{on} (A/)</td>
<td>2047/666/749</td>
<td>1110</td>
</tr>
<tr>
<td>C_{ox-elec} (fF/\text{f}^2)</td>
<td>37.7/20.1/28.2</td>
<td>31.3</td>
</tr>
<tr>
<td>FO4 delay (ps)</td>
<td>8.21/31.2/17.86</td>
<td>24.22</td>
</tr>
</tbody>
</table>

Technology Model used by FreePDK and used for our custom circuit design, is different and more conservative. To bring the cache delays in line with our synthesized and custom logic, we recalculated CACTI’s device and wire parameters using the BSIM4 model. Table 3.2 shows the original and adjusted CACTI device parameters for 45nm. Similarly, we adjusted the wire parameters of CACTI for FreePDK.

Each canonical pipeline stage has many underlying implementations in the CPSL that differ in their complexity and sub-pipelining. For each design in the CPSL, we performed multiple synthesis runs with successively tighter timing constraints until the constraint could not be met. In this way, we converged upon the minimum propagation delay.

Because of the sheer number of designs, delay numbers are from synthesis, not place-and-route. Nonetheless, Synopsys Design Compiler does estimate wire delays, using estimates of wire lengths based on a cursory place-and-route. Bypass wires are long enough that they require special attention. We estimated bypass wire delays using SPICE simulation. Applying the method proposed by Palacharla et al. [69], bypass wire lengths are based on the sizes of function units and the physical register file.
3.4 Validation Results

In this section, the quality of the RTL designs produced by FabScalar is evaluated. Prior to the validation experiments presented in this section, extensive unit-level testing of the CPSL and testing of a baseline 4-way superscalar core was performed. The latter enabled perfecting the interfaces and interactions among canonical pipeline stages. Subsequently, assembling and debugging all of the cores in this section proceeded efficiently within the span of a month, moreover, most of the bugs encountered in this period were in the course of implementing the composition tool itself (scripted verilog instantiation and stitching of stages).

Validation is performed along three fronts:

1. **Functional and IPC validation**: A dozen different cores are generated, covering a range of widths, sizes, and depths. 100 million instruction SimPoints [81] of six SPEC2000 integer benchmarks are executed on the cores and the instructions-per-cycle (IPC) results are within expected ranges and follow expected trends. The other six integer benchmarks were not tested because their SimPoints have occasional floating-point instructions and will be tested in future work when the floating-point registers and issue queue are added to the CPSL stage designs.

2. **Timing validation**: We also evaluate the quality of the RTL in terms of cycle time. To validate cycle time, we compare several commercial general-purpose and embedded cores with similarly configured FabScalar generated cores. Validating cycle time is challenging and imperfect for a number of reasons:

   - Different technology nodes, technology libraries, and foundry processes. We deal with this issue by converting cycle time into the number of FO4 inverter delays of the technology, yielding a technology-independent comparison (although the subtle influence that the underlying technology had on design choices and circuit optimization choices cannot be undone).

   - Different degrees of custom design, including the extent of circuit optimization, dynamic logic, and latch based design for accommodating logic partition imbalances. We deal with this issue only partially by employing multiported RAMs and CAMs generated by FabMem. We also compare to a commercial
fully-synthesized embedded core at one end of the spectrum. Regarding latch based design, in addition to comparing cycle time, we also examine raw total logic delay through the pipeline from Fetch to Execute.

- Different ISAs and unique microarchitecture features. For example, the current CPSL does not have Issue stage designs with multiple schedulers (see Figure 3.2, last column) or replicated register files. Multiple smaller schedulers reduce the select logic delay by reducing the number of instructions contending for a given execution pipeline way, at the cost of some load imbalance among the multiple issue queues. More importantly, when there are multiple FUs of the same type, providing each FU with a dedicated issue queue avoids cascading select trees, a big delay savings. Replicated register files reduce the number of read ports in each register file copy, improving their access times. While these techniques are not yet represented in the CPSL, their effect can be modeled for timing validation purposes by applying a smaller/simpler issue queue and a register file with fewer read ports.

3. *Suitability for physical design*: We demonstrate the suitability of the generated RTL for full synthesis and place-and-route by a standard ASIC flow.

### 3.4.1 Functional and IPC Validation

The FabScalar tool was used to generate the RTL designs for the twelve cores described in Table 3.3. Before discussing the cores, several points about the table need clarification. First, some stage depths are omitted from the table: stages that have only a single depth implemented in the CPSL (Dispatch and Retire are always 1-deep and 2-deep, respectively) or stages that are not varied among the twelve cores (Decode is fixed at 1-deep). Second, the Branch Order Buffer (BOB) is a FIFO buffer in the Fetch stage holding control information for all predicted but not yet retired branches. It facilitates updating the Branch History Table (BHT) non-speculatively as branches retire as well as checkpointing certain predictor state (such as the global branch history register). Third, the quoted fetch-to-execute pipeline depth reflects the minimum branch misprediction penalty, and includes 1 cycle of execution in the branch unit (Writeback and Retire depths are excluded from this number). Of the two branch recovery implementations represented in the CPSL – shadow maps vs. handle like exception (Figure 3.2) – all twelve cores employ
the latter lower complexity and lower performance approach.

Cores 1 through 6 were selected primarily to explore stage widths and structure sizes. Except for Core-6, depths are the same across these cores. Core-6 is a particularly narrow core: 2-way superscalar in the front-end. Core-2 has different widths in the front-end (4) and back-end (6). Core-5 is a particularly wide core: 8-way superscalar fetch and execute with large resources.

Cores 6 through 10 aim to explore depths of stages and the fetch-to-execute pipeline depth. Cores 7 and 8 resemble Core-1 except that Core-7 is shallower (fetch-to-execute = 9) and Core-8 is deeper (fetch-to-execute = 14). They differ in their Issue and Register Read depths. Cores 9 and 10 are unique in that their Fetch-1 sub-stage of Fetch is pipelined into two cycles, using block-ahead branch prediction. This yields a total Fetch depth of three cycles. Cores 9 and 10 differ in their Issue and Register Read depths. Core-10 is the deepest of the twelve cores (fetch-to-execute = 15), although not the deepest possible with the CPSL since Rename and Fetch (the Fetch-2 logic) can be deepened further.

Cores 11 and 12 are the same as Cores 1 and 2, respectively, except they use the gshare [61] instead of the bimodal branch predictor. Since the gshare predictor can only conveniently supply one branch prediction per cycle, Fetch stage designs in the CPSL employing gshare present a tradeoff between slightly reducing fetch bandwidth and increasing fetch accuracy.

Results of executing the 100 million instruction SimPoints of six benchmarks are shown in Figure 3.3. Results are shown for both RTL (“Verilog”) and the cycle-accurate C++ simulator (“C++”). Block-ahead prediction is not yet implemented in the C++ so its datapoints are missing for Cores 9/10. The first thing to note is that the cores execute the benchmarks successfully. Second, IPCs are within the norm for SPEC integer benchmarks, especially considering the conservative method for recovering from load violations and branch mispredictions employed by these cores. Third, the RTL and C++ follow each other closely. The latter result increases confidence in the RTL modeling of the design: if performance anomalies are observed, they are more likely inherent in the
design rather than in the RTL modeling of the design.

Mcf is notorious for having extremely low IPC due to cache thrashing. Its high IPC in this case is due to the fact that (as mentioned previously) these RTL simulations do not include L1 cache controllers, hence, loads always take two cycles: 1 cycle for agen and 1 cycle for accessing either the behavioral memory interface or the RTL-implemented store queue (store-to-load forwarding). Also note, loads never stall for prior unknown store addresses and any resulting misspeculated loads cause a squash-based recovery (one of the design choices in Figure 3.2).

Differences in IPCs among cores tend to correspond with their microarchitectural differences. For example, among Cores 1 through 5, we expect Core 5 to have the highest IPC since it is the most aggressive core, the depths are the same, and no negative cycle time consequences are applied in an IPC-only comparison. Cores 8 and 10 are the deepest pipelines and they have lower IPCs than other configurations as a result. Some pairwise comparisons of cores could go either way due to increasing some parameters and decreasing others, leading to potentially non-monotonic cores. For example, Core-6 has the same or higher IPC than Core-2 in all benchmarks except bzip. Core-6 is narrow (2-way fetch) but its advantage over other configurations is its 1-cycle wakeup-select loop. In the case of bzip, however, there is apparently sufficient ILP to outweigh the longer wakeup-select loop.

If there are anomalies, for example, a more aggressive core having lower IPC than a simpler core of the same pipeline depth, they are sometimes caused by more frequent load violations or branch mispredictions that stem from larger window sizes. Extra recoveries are performance-debilitating when recovery is a full squash from the head of the active list.
Table 3.3: Cores used for functional and IPC validation experiments.

<table>
<thead>
<tr>
<th>Core-1</th>
<th>Core-2</th>
<th>Core-3</th>
<th>Core-4</th>
<th>Core-5</th>
<th>Core-6</th>
<th>Core-7</th>
<th>Core-8</th>
<th>Core-9</th>
<th>Core-10</th>
<th>Core-11</th>
<th>Core-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch / Decode / Rename / Dispatch width</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Issue / RR / Execute / WB / Retire width</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>function unit mix (simple, complex, branch, load/store)</td>
<td>1,1,1,1</td>
<td>3,1,1,1</td>
<td>2,1,1,1</td>
<td>3,1,1,1</td>
<td>5,1,1,1</td>
<td>1,1,1,1</td>
<td>1,1,1,1</td>
<td>1,1,1,1</td>
<td>3,1,1,1</td>
<td>3,1,1,1</td>
<td>1,1,1,1</td>
</tr>
<tr>
<td>fetch queue</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>active list (ROB)</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>physical register file (PRF)</td>
<td>96</td>
<td>128</td>
<td>128</td>
<td>192</td>
<td>512</td>
<td>64</td>
<td>96</td>
<td>96</td>
<td>192</td>
<td>192</td>
<td>96</td>
</tr>
<tr>
<td>issue queue (IQ)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>branch predictor</td>
<td>bimodal</td>
<td>Block-ahead</td>
<td>Gshare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch history table (BHT) (# entries)</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
</tr>
<tr>
<td>branch target buffer (BTB) (# entries)</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
<td>4K</td>
</tr>
<tr>
<td>return address stack (RAS)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>branch order buffer (BOB)</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Fetch depth</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Rename depth</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Issue depth: total / wakeup-select loop</td>
<td>2 / 2</td>
<td>2 / 2</td>
<td>2 / 2</td>
<td>2 / 2</td>
<td>2 / 2</td>
<td>1 / 1</td>
<td>1 / 1</td>
<td>3 / 2</td>
<td>2 / 2</td>
<td>3 / 2</td>
<td>2 / 2</td>
</tr>
<tr>
<td>Register Read (and Writeback) depth</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>fetch-to-execute pipeline depth</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>

40
### 3.4.2 Timing Validation

For timing validation, we compare cycle times and fetch-to-execute delays of FabScalar generated cores with three different commercial processors: 90nm POWER5 [82], 180nm Alpha 21364 [62], and 65nm MIPS32 74K [48]. All three implement RISC ISAs and they represent extremes from highly custom designed to fully synthesized (MIPS32 74K). Table 3.4 shows major microarchitecture parameters of the three processors that could be gleaned from the literature.

All delays are converted into the number of FO4 inverter delays for the underlying technology. We obtained the number of FO4 delays in a pipeline stage for each commercial processor, from published data [48] [62] [82] [63].

The last five rows in Table 3.4 shows delay comparisons between the commercial cores and similarly configured FabScalar generated cores. Five numbers are shown:

1. Cycle time of the commercial core.
2. Cycle time of the similarly-configured FabScalar core of the same pipeline depth.
3. Cycle time of a deeper version of the FabScalar core, with its fetch-to-execute pipeline depth shown in parentheses. This shows how much additional sub-pipelining is needed to compensate for the lesser degree of custom design of the FabScalar core.
4. Raw fetch-to-execute delay of the FabScalar core. This is the sum of propagation delays of all the stages between Fetch and Execute.
5. The final number is #4 above divided by the fetch-to-execute pipeline depth of the commercial core. This corresponds to the FabScalar core’s hypothetical cycle time if pipeline registers evenly divided up the raw fetch-to-execute delay (no imbalance among pipeline stages). This cycle time is the best that could be achieved with careful latch based design, for the same pipeline depth.

The cycle time of the FabScalar-Power5 is relatively close to that of the Power5: 29 FO4 compared to 23 FO4, respectively. Slightly deeper pipelining (15 deep instead of 12 deep) yields an even closer 25 FO4 cycle time. The same cycle time of 24 FO4 can also be gotten with ideal latch-based design. All of these comparisons, and especially
Figure 3.3: Results of executing 100 million instruction SimPoints of six benchmarks on the twelve cores.

the latter (raw fetch-to-execute delay), confirm that the FabScalar generated RTL and the FabMem generated RAMs/CAMs are of reasonable quality from the standpoint of propagation delay.

A larger difference is observed for the FabScalar-21364 and 21364: 37 FO4 compared to 25 FO4. What is interesting is that the 21364 has a cycle time close to the Power5 despite the 21364 being half as deep. This is partly due to lower superscalar complexity of the older 21364 but it also suggests a significant degree of total delay optimization (Alpha processors gained a reputation as speed demons). Indeed, the deeper FabScalar-21364 needs nearly twice the pipeline depth to reach the 21364 cycle time, despite being similarly configured.

The MIPS 74K is a fully-synthesized design. This means that structures normally implemented with custom RAMs and CAMs are synthesized to flip-flops (except for caches). Accordingly, for a fair comparison, the delays for FabScalar-74K are also based on syn-
Table 3.4: Delay comparisons of commercial processors with similarly configured Fab-Scalar generated cores.

<table>
<thead>
<tr>
<th></th>
<th>Power5</th>
<th>Alpha-21364</th>
<th>MIPS 74K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Width</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>24</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>Issue Queue(s)</td>
<td>Int+Ld/St: 36, FP: 24, Br.: 12, CR: 10</td>
<td>Int:20, FP:15</td>
<td>Int:8, Agen:8</td>
</tr>
<tr>
<td>L1 I-cache / L1 D-cache (KB)</td>
<td>64 / 32</td>
<td>64 / 64</td>
<td>32 / 32</td>
</tr>
<tr>
<td>fetch-to-execute pipeline depth</td>
<td>12</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Cycle Time of commercial core</td>
<td>23 FO4</td>
<td>25 FO4</td>
<td>33 FO4</td>
</tr>
<tr>
<td>Cycle Time of FabScalar core</td>
<td>29 FO4</td>
<td>37 FO4</td>
<td>32 FO4</td>
</tr>
<tr>
<td>Cycle Time of deeper FabScalar core</td>
<td>25 FO4 (depth=15)</td>
<td>26 FO4 (depth=11)</td>
<td>N/A</td>
</tr>
<tr>
<td>raw fetch-to-execute delay of FabScalar core</td>
<td>291 FO4</td>
<td>188 FO4</td>
<td>384 FO4</td>
</tr>
<tr>
<td>Cycle Time of FabScalar core with ideal latch-based design</td>
<td>24 FO4</td>
<td>32 FO4</td>
<td>N/A</td>
</tr>
</tbody>
</table>

thesis alone: FabMem is not used. The cycle times of these two fully-synthesized cores are nearly identical: 32 FO4 for the FabScalar-74K versus 33 FO4 for the 74K. That both cores are fully-synthesized, use virtually the same ISA, and have the same cycle time, further supports the assertion that the RTL is of reasonable quality from the standpoint of propagation delay.

Power consumption is more challenging to validate because there is no power equivalent of the FO4 concept, i.e., the technology dependence is difficult to abstract away. We present peak power and area, nonetheless, using synthesis of the FabScalar versions of the three processors above. Peak power and areas of RAMs and CAMs are obtained using FabMem. The results are shown in Table 3.5.
Table 3.5: Power and area of FabScalar versions of the three commercial processors (not including floating point pipeline and level-1 caches).

<table>
<thead>
<tr>
<th></th>
<th>FabScalar-Power5</th>
<th>FabScalar-21364</th>
<th>FabScalar-74K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area (mm²)</td>
<td>2.55</td>
<td>1.66</td>
<td>1.17</td>
</tr>
<tr>
<td>Peak Power (W)</td>
<td>4.64</td>
<td>1.43</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Figure 3.4: Physical design of a 4-way superscalar processor.

3.4.3 Suitability for Standard ASIC Flows

To demonstrate that FabScalar-generated RTL can be taken through standard ASIC flows, we synthesized and place-and-routed a 4-way superscalar processor. The physical design is shown in Figure 3.4. This particular core uses shadow maps for quickly recovering from mispredictions. Since FabMem cannot generate the highly specialized shadow map to rename map connectivity, the rename map table and shadow map table are synthesized to flip-flops in this physical design: this is evident in the large rename block in the chip diagram.
3.5 Extensibility of CPSL

While it is difficult to make generalizations about extensibility, we present two specific examples. The examples relate to two major IPC bottlenecks in the FabScalar-generated cores presented earlier: load misspeculations and branch mispredictions.

In the cores presented earlier, completion of a load is not stalled by prior unissued stores. If the speculatively-completed load is later discovered to depend on one of the stores, the recovery penalty is severe: the processor waits until the load reaches the head of the active list, squashes the load and all instructions after it, and restarts from the load. It took the authors two days to enhance the CPSL with a simple dependence predictor (predict a load will conflict if it has in the past) and logic to stall completion of suspect loads in the load queue until all prior stores have issued. There is already a mechanism to complete loads that were once stalled (for cache-missed loads): the load is reinjected into the load-store pipe (in the next free cycle) at which time it is safely completed. The changes are localized to the Dispatch stage (dependence predictor) and load queue (bit vectors for tracking store queue entries of unissued stores). The global impact is small: there is an additional signal from the Retire stage to Dispatch stage (PC of a misspeculated load trains the dependence predictor) and loads carry an extra bit with them (whether or not to speculatively complete). Two of the benchmarks, bzip and vortex, have quite low branch misprediction rates, hence, their performance is sensitive to load misspeculations. Consequently, adding the dependence predictor increased the IPC of bzip from 0.89 to 1.39 and the IPC of vortex from 0.76 to 1.14 (for 10M SimPoints).

In the second example, Gandhi et al. [36] changed the Fetch-1 stage to totally decouple the conditional branch predictor (taken/not-taken prediction) from the next-PC logic (BTB, RAS, and next-PC mux), enabling pipelining a large/complex branch predictor arbitrarily deep (high accuracy with fast cycle-time). This change took longer to implement than the previous example because it is more complex and required inventive design. Nevertheless, changes were confined to the Fetch-1 stage.
Chapter 4

Studying Performance and Efficiency Advantages of Employing Microarchitecturally Diverse Cores

FabScalar provides us a unique opportunity to understand the performance and computation-efficiency (referred to as efficiency for brevity) advantages of heterogeneous multi-core compared to homogeneous multi-core using real core designs. Using FabScalar differs from other methodologies for heterogeneous multi-core studies [57] [52], in that it provides whole-processor synthesizable RTL and hard macros for RAMs/CAMs for each configuration in the large design space. Moreover, FabScalar-generated cores highlight two important aspects of logic pipelining which often have been ignored in previous heterogeneous multi-core studies: 1) pipeline imbalance and 2) limitations of sub-pipelining.

A pipeline has a balanced design if the logic complexity of each microarchitectural pipeline stage is equal. Equivalently, the propagation delays of all pipeline stages are equal. However, real logic cannot be arbitrarily partitioned into segments with identical delays. A pipeline stage consists of logic and hardware structures that are typically divided to perform specialized functions. Thus, certain locations in the stage are more suitable for pipelining due to this inherent separation by function. Also, certain hardware structures, like RAMs, cannot be partitioned to obtain an arbitrary division of delay.

This exposes an important property of pipelining actual hardware designs: pipeline
imbalance. There are two aspects of pipeline imbalance stemming from non-arbitrary division of hardware. First, consider the canonical superscalar pipeline where each stage is not sub-pipelined. The delay of the Fetch stage may be non-trivially different from the delay of the issue stage, despite adjusting structure sizes and complexity to minimize this difference. The imbalance, in this case, is at the canonical stage level. Second, consider the sub-pipelining of one of the canonical stages for a given complexity. The issue stage, for example, can be pipelined into three sub-stages [60] by placing pipeline registers at appropriate boundaries. Doing this, however, divides the total issue logic delay unevenly.

The rest of this chapter explores the benefits of employing microarchitecturally diverse cores. Section 4.1 describes a FabScalar-based performance, power, and area (PPA) estimation framework that enables fast design space studies for performance and efficiency. Section 4.2 summarizes the benchmarks used, across this thesis, to study application diversity and heterogeneous multi-core processors. Section 4.3 describes our design space of diverse superscalar cores. Section 4.4 presents and discusses results. Finally, Section 4.5 compares DVFS with heterogeneous multi-core.

4.1 FabScalar-PPA Framework

FabScalar provides RTL of the whole core which can further be implemented, using standard ASIC flow or semi-custom approach, to get its clock frequency, power consumption and silicon area. However, implementing many diverse cores for frequency, power, and area is very time consuming. To overcome this limitation, we create a framework using CPSL components to measure frequency, power, and area of an arbitrary microarchitecture configuration. Figure 4.1 shows our FabScalar-PPA framework.

Fundamentally, any core generated from FabScalar is composed using CPSL components. We implement each component of CPSL, such as 1-wide 1-deep Rename, 2-wide 1-deep Rename, 2-wide 2-deep Rename, etc., individually using Synopsys Design Compiler on FreePDK 45nm standard cell library. Implementing components using Design Compiler provides their delay, energy, and area numbers. Design Compiler uses the statistical wire-load model, provided with the standard cell library, to account for wire delays. The delay of Bypass wires is very dependent on the physical layout. Hence, canonical stages
related to Bypass wires are further implemented using Cadence SoC Encounter. A canonical stage with a hardware buffer in it is implemented for different sizes of the buffer. For example, Retire stage is implemented for different Active List sizes: 64, 96, 128, 192, 256, 384, and 512. Based on implementations using Design Compiler and SoC Encounter, a database consisting of delay, energy, and area numbers of CPSL components is created. Moreover, delay, energy, and area numbers for large memory structures, such as caches and BTB, are estimated using CACTI. The database and numbers from CACTI are used by the FabScalar-PPA framework to get clock frequency, power, and area numbers for a microarchitecture configuration. The framework is integrated with the FabScalar C++ simulator to get activity counts of canonical stages to calculate average power consumption for an application.

Figure 4.1: FabScalar-PPA framework for measuring performance, power, and area for a microarchitecture configuration.
### Table 4.1: The SPEC CPU2000 SimPoints.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Interval-Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>341, 1873, 3089, 9277</td>
</tr>
<tr>
<td>crafty</td>
<td>234, 5647, 12166, 21513</td>
</tr>
<tr>
<td>gap</td>
<td>3229, 15783, 16913, 21010</td>
</tr>
<tr>
<td>gcc</td>
<td>264, 473, 628, 873</td>
</tr>
<tr>
<td>gzip</td>
<td>779, 1361, 5030</td>
</tr>
<tr>
<td>mcf</td>
<td>2018, 3091</td>
</tr>
<tr>
<td>parser</td>
<td>5201, 10758, 19972, 24856</td>
</tr>
<tr>
<td>perl</td>
<td>18, 415, 781, 1309</td>
</tr>
<tr>
<td>twolf</td>
<td>2482, 8155, 15968, 27755</td>
</tr>
<tr>
<td>vortex</td>
<td>5877, 7432, 9025</td>
</tr>
<tr>
<td>vpr</td>
<td>1350, 3463, 6120</td>
</tr>
<tr>
<td>ammp</td>
<td>2423, 2891, 2945, 3018, 3783</td>
</tr>
<tr>
<td>applu</td>
<td>34, 49</td>
</tr>
<tr>
<td>apsi</td>
<td>2739, 21688, 49702</td>
</tr>
<tr>
<td>art</td>
<td>591, 3441, 5922, 10257</td>
</tr>
<tr>
<td>equake</td>
<td>463, 869, 1046, 1093, 2796</td>
</tr>
<tr>
<td>mesa</td>
<td>227, 2297, 3731, 5172, 27859</td>
</tr>
<tr>
<td>mgrid</td>
<td>2977, 3283, 3490, 3657</td>
</tr>
<tr>
<td>swim</td>
<td>1226, 1312, 1582, 1583</td>
</tr>
<tr>
<td>wupwise</td>
<td>12487, 23352, 25364, 97584, 107351</td>
</tr>
</tbody>
</table>

#### 4.2 Application Space

The integer and floating-point benchmarks from the SPEC CPU2000 [6] suite are used for this study. The SimPoint tool [81] was used to generate up to four or five 10 million instruction intervals from each benchmark. The SimPoint tool analyses the dynamic instruction trace of a benchmark to find the intervals that combined together represent the characteristics of the entire trace. Table 4.1 shows all the benchmarks and their corresponding SimPoint interval-numbers. The interval size multiplied with the interval-number (also referred to as id) gives the total number of instructions to skip before starting the timing simulation. Different intervals of a benchmark also represent different phase behaviors within the benchmark. In all, there are 76 SimPoints in the experiments, which we refer to as benchmarks from now on.
4.3 A Generic Heterogeneous Multi-Core

Designing a heterogeneous multi-core processor requires identifying the microarchitecture characteristics of diverse cores that yield optimal objective function value under a set of constraints. The objective function might be maximizing single-thread performance or minimizing energy-delay product and the constraints might be die area or power budget. Prior work achieved similar objectives by exhaustively simulating applications on all cores [64] [66] or multiprogrammed workloads on all core combinations [52], in a huge design space. The design space consists of cores with different superscalar widths, ILP-extracting hardware structures, and speculative mechanisms. The Cartesian product of independent microarchitecture parameters can easily result in millions of valid design points. Designing a general-purpose heterogeneous multi-core processor using a set of benchmarks and cores has two potential drawbacks: 1) the high computational complexity of design space exploration, and 2) the performance robustness of the design trained to a specific workload set.

In this thesis, we take a very different approach to study the performance and efficiency advantages of heterogeneous multi-core. Cores, in our design space, are selected to provide a broad range of microarchitectural configurations without a priori knowledge of applications. The core types are not trained for any specific application and they represent configurations required to target diverse instruction-level behavior [23] [24]. Our design space consists of cores with superscalar width ranging from 1 through 6. The superscalar width is fundamental to the core complexity and determines maximum achievable instruction throughput for an application. For each width, small, medium and large ILP-extracting structures, e.g., issue-queue and ROB, are considered to target different forms of ILP: near, average, and far. The structure sizes are determined by constraining a given superscalar width for three different clock frequencies, yielding small, medium, and large structures. Core names and their configurations are presented in Table 4.2. There are 18 diverse core types in the design space. The fetch width is equal to issue width and all the caches use a uniform line size of 64B. Table 4.3 presents each cores’ frequency, area, and peak power on the FreePDK 45nm technology. The clock frequencies of cores range from 2.5GHz (for core 1W-S) to 1.11GHz (for core 6W-L). The off-chip memory latency is fixed to 100ns for all designs.
Table 4.2: Microarchitecture design space for studying advantages of heterogeneous multi-core. ILP-Extracting Buffers (Issue Queue, Active List, LQ/SQ), L1-ICache (Size (KB), Associativity), L1-DCache (Size (KB), Associativity), Pipeline Depth (Fetch-to-Execute).

<table>
<thead>
<tr>
<th>Core Name</th>
<th>Superscalar Width</th>
<th>ILP-Extracting Buffers</th>
<th>L1-ICache</th>
<th>L1-DCache</th>
<th>Pipeline Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1W-S</td>
<td>1</td>
<td>8, 64, 16</td>
<td>8, 1</td>
<td>8, 1</td>
<td>15</td>
</tr>
<tr>
<td>1W-M</td>
<td>1</td>
<td>24, 128, 32</td>
<td>16, 2</td>
<td>16, 2</td>
<td>15</td>
</tr>
<tr>
<td>1W-L</td>
<td>1</td>
<td>64, 384, 128</td>
<td>32, 4</td>
<td>32, 4</td>
<td>14</td>
</tr>
<tr>
<td>2W-S</td>
<td>2</td>
<td>32, 96, 64</td>
<td>16, 4</td>
<td>16, 4</td>
<td>16</td>
</tr>
<tr>
<td>2W-M</td>
<td>2</td>
<td>48, 192, 128</td>
<td>32, 4</td>
<td>32, 4</td>
<td>14</td>
</tr>
<tr>
<td>2W-L</td>
<td>2</td>
<td>64, 384, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>13</td>
</tr>
<tr>
<td>3W-S</td>
<td>3</td>
<td>16, 64, 32</td>
<td>16, 4</td>
<td>16, 4</td>
<td>18</td>
</tr>
<tr>
<td>3W-M</td>
<td>3</td>
<td>48, 128, 64</td>
<td>32, 4</td>
<td>32, 4</td>
<td>14</td>
</tr>
<tr>
<td>3W-L</td>
<td>3</td>
<td>64, 384, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>15</td>
</tr>
<tr>
<td>4W-S</td>
<td>4</td>
<td>32, 128, 32</td>
<td>32, 4</td>
<td>32, 4</td>
<td>16</td>
</tr>
<tr>
<td>4W-M</td>
<td>4</td>
<td>48, 256, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>15</td>
</tr>
<tr>
<td>4W-L</td>
<td>4</td>
<td>64, 384, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>15</td>
</tr>
<tr>
<td>5W-S</td>
<td>5</td>
<td>24, 64, 32</td>
<td>32, 4</td>
<td>32, 4</td>
<td>16</td>
</tr>
<tr>
<td>5W-M</td>
<td>5</td>
<td>48, 192, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>16</td>
</tr>
<tr>
<td>5W-L</td>
<td>5</td>
<td>64, 256, 128</td>
<td>64, 4</td>
<td>32, 4</td>
<td>16</td>
</tr>
<tr>
<td>6W-S</td>
<td>6</td>
<td>32, 128, 64</td>
<td>16, 4</td>
<td>32, 4</td>
<td>15</td>
</tr>
<tr>
<td>6W-M</td>
<td>6</td>
<td>48, 256, 128</td>
<td>32, 4</td>
<td>64, 4</td>
<td>16</td>
</tr>
<tr>
<td>6W-L</td>
<td>6</td>
<td>64, 384, 128</td>
<td>64, 4</td>
<td>64, 4</td>
<td>15</td>
</tr>
<tr>
<td>Core Name</td>
<td>Frequency (GHz)</td>
<td>Area (mm$^2$)</td>
<td>Peak Power (W/GHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>--------------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1W-S</td>
<td>2.50</td>
<td>1.582</td>
<td>0.6052</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1W-M</td>
<td>2.00</td>
<td>1.653</td>
<td>0.635</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1W-L</td>
<td>1.67</td>
<td>1.892</td>
<td>0.860</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2W-S</td>
<td>2.00</td>
<td>1.454</td>
<td>0.896</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2W-M</td>
<td>1.67</td>
<td>1.803</td>
<td>1.138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2W-L</td>
<td>1.43</td>
<td>2.32</td>
<td>1.233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3W-S</td>
<td>2.00</td>
<td>1.42</td>
<td>1.011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3W-M</td>
<td>1.67</td>
<td>1.798</td>
<td>1.348</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3W-L</td>
<td>1.43</td>
<td>2.412</td>
<td>1.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4W-S</td>
<td>1.67</td>
<td>1.85</td>
<td>1.696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4W-M</td>
<td>1.43</td>
<td>2.428</td>
<td>2.038</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4W-L</td>
<td>1.25</td>
<td>2.561</td>
<td>2.206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5W-S</td>
<td>1.67</td>
<td>1.833</td>
<td>1.700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5W-M</td>
<td>1.43</td>
<td>2.496</td>
<td>2.484</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5W-L</td>
<td>1.25</td>
<td>2.321</td>
<td>2.766</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6W-S</td>
<td>1.43</td>
<td>2.063</td>
<td>2.744</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6W-M</td>
<td>1.25</td>
<td>2.515</td>
<td>3.386</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6W-L</td>
<td>1.11</td>
<td>2.834</td>
<td>3.763</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.4 Heterogeneous Multi-core Analysis

We study three different metrics for our application and design space to understand the advantage of heterogeneous multi-core compared to homogeneous multi-core: billions of instructions per second (BIPS), $BIPS^2/Watt$, and $BIPS^3/Watt$. BIPS is a function of IPC and clock frequency, and it is a performance-only metric. $BIPS^2/Watt$ is the inverse of energy-delay product (EDP) metric and $BIPS^3/Watt$ is the inverse of energy-delay$^2$ product (ED$^2$P) metric. $BIPS^2/Watt$ and $BIPS^3/Watt$ are both efficiency metrics, however, they are used in different design contexts \cite{38} \cite{15}. $BIPS^2/Watt$ or EDP-based formulation is appropriate where battery life (or energy consumption) is the primary driver, for example application processors in mobile devices. For high-performance server-class processors, the $BIPS^3/Watt$ or ED$^2$P-based formulation is more appropriate. Typically, server-class processors are designed for performance and designers optimize the design for BIPS as long as it is within a maximum power budget constraint.

We intentionally don’t constrain die area in any of the metrics because current and future designs are primarily limited by the power wall (refer to the discussion on TechnologyTrends in Chapter 2).

Our goal is to maximize single-thread performance (BIPS) or efficiency ($BIPS^2/Watt$, $BIPS^3/Watt$) by employing microarchitecturally diverse cores. Each benchmark in Table 4.1 is simulated on our core design space using the FabScalar-PPA framework, and the desired metrics are generated. First, we discuss performance-only metric results followed by efficiency metric results. For each metric, the best core on average is found, i.e., the core that yields the best average metric across all benchmarks. Average BIPS of the application space is calculated by taking the harmonic-mean of all benchmarks’ BIPS. Average power consumption is measured by dividing total energy consumed by all the benchmarks with total time to execute all the benchmarks. The time to execute a benchmark is calculated by dividing number of instructions committed (10 million in our case) with the benchmark’s BIPS. The average core is referred to as Best-1 for rest of this chapter. The Best-1 core serves as the core design for our baseline homogeneous multi-core, and it represents an approximate compromise across all benchmarks to accommodate a range of instruction-level behaviors. Further, the optimal core for each benchmark is found, i.e., the core that yields the best metric for a benchmark. The optimal core for a
benchmark represents the microarchitecture configuration that is the best trade-off between the benchmark’s instruction-level behaviors and core’s hardware implementation cost. The performance (efficiency) metric of the optimal core for a benchmark is compared to its performance (efficiency) metric on the Best-1 core. The comparison provides us the potential gain a benchmark can achieve if the microarchitecture is tuned to its instruction-level behaviors. The gain is measured in percentage improvement. We summarize all benchmarks’ results using average, median, and maximum gain.

Employing 18 diverse cores in a multi-core processor might not be feasible because of die area or design time constraints. We measure the benefit of employing N core types in a multi-core. N is varied from 2 through 5. All possible 18-combination-N possibilities are evaluated to find the best N-core-type heterogeneous multi-core processor. Finally, we present a table for each metric that shows the best core for each benchmark. Recall that our benchmarks are SimPoints of SPEC CPU2000 workload suite. The fact that SimPoints of a SPEC workload prefer different cores shows the potential advantage of dynamic thread migration in a heterogeneous multi-core processor.
Figure 4.3: Performance (BIPS) gain achieved by benchmarks on their optimal cores.

Figure 4.4: Distribution of benchmarks falling in different BIPS-gain ranges. Each sector in the chart represents the number of benchmarks falling in the corresponding gain range.
Table 4.4: Performance (BIPS) gain of various N-core-type heterogeneous multi-core processors.

<table>
<thead>
<tr>
<th>N</th>
<th>Cores</th>
<th>Average Performance (BIPS) Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4W-M, 6W-L</td>
<td>2.9%</td>
</tr>
<tr>
<td>3</td>
<td>4W-S, 4W-M, 6W-L</td>
<td>5.03%</td>
</tr>
<tr>
<td>4</td>
<td>3W-L, 4W-S, 4W-M, 6W-M</td>
<td>6.1%</td>
</tr>
<tr>
<td>5</td>
<td>2W-S, 3W-L, 4W-S, 4W-M, 6W-M</td>
<td>6.67%</td>
</tr>
</tbody>
</table>

4.4.1 Performance and Efficiency Results

The Best-1 core for BIPS is 4W-M. Figure 4.2 shows the summary of BIPS improvement (w.r.t Best-1) due to employing microarchitecturally diverse cores. The average gain achieved is 7.46% and the maximum gain achieved is 33.73%. Figure 4.3 presents BIPS-gain achieved by each benchmark by executing on its optimal core. 4W-M is the optimal core for many benchmarks, hence, their gains are zero. Other benchmarks, for example, bzip.341, gcc.628, swim.1226, swim.1538 and wupwise.25364, achieve significant gain. The pie-chart in Figure 4.4 shows the number of benchmarks falling in different BIPS-gain ranges. 27 out of 76 benchmarks achieve more than 10% gain.

The BIPS-gain achieved by various N-core-type heterogeneous multi-core processors is presented in Table 4.4. With only 4 core types, much of the performance, with respect to 18 core types, can be achieved. Moreover, 4W-M (Best-1 core for BIPS) appears in all N-core-type designs.

The Best-1 core for BIPS²/Watt is 2W-S. Figure 4.5 shows the summary of BIPS²/Watt-gain achieved by employing microarchitecturally diverse cores. The average gain achieved is 16.47% and the maximum gain achieved is 11.24%. Figure 4.6 presents BIPS²/Watt-gain achieved by each benchmark by executing on its optimal core. 2W-S is the optimal core for many benchmarks, hence, their gains are zero. The benchmarks, crafty.234, ammp.2945, ammp.3018, art.5922, and swim.1583, achieve significant gains. The pie-chart in Figure 4.7 shows the number of benchmarks falling in different BIPS²/Watt-gain ranges. 19 out of 76 benchmarks achieve more than 50% gain.
Table 4.5: Best-performing core (BIPS) for each benchmark in the application space.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip.1873</td>
<td>4W-M</td>
<td>perl.18</td>
<td>4W-M</td>
<td>art.591</td>
<td>4W-M</td>
</tr>
<tr>
<td>bzip.3089</td>
<td>6W-L</td>
<td>perl.1309</td>
<td>3W-M</td>
<td>art.3441</td>
<td>4W-M</td>
</tr>
<tr>
<td>bzip.341</td>
<td>6W-S</td>
<td>perl.781</td>
<td>4W-S</td>
<td>art.5922</td>
<td>4W-M</td>
</tr>
<tr>
<td>bzip.9277</td>
<td>6W-S</td>
<td>perl.415</td>
<td>4W-S</td>
<td>art.10257</td>
<td>4W-M</td>
</tr>
<tr>
<td>crafty.21513</td>
<td>4W-M</td>
<td>twolf.2482</td>
<td>4W-S</td>
<td>equake.1046</td>
<td>3W-L</td>
</tr>
<tr>
<td>crafty.12166</td>
<td>4W-M</td>
<td>twolf.27755</td>
<td>4W-S</td>
<td>equake.1093</td>
<td>3W-L</td>
</tr>
<tr>
<td>crafty.5647</td>
<td>4W-M</td>
<td>twolf.15968</td>
<td>4W-S</td>
<td>equake.2796</td>
<td>3W-L</td>
</tr>
<tr>
<td>crafty.234</td>
<td>4W-M</td>
<td>twolf.8155</td>
<td>4W-S</td>
<td>equake.463</td>
<td>6W-S</td>
</tr>
<tr>
<td>gap.3229</td>
<td>4W-M</td>
<td>vortex.7432</td>
<td>3W-M</td>
<td>equake.869</td>
<td>3W-L</td>
</tr>
<tr>
<td>gap.15783</td>
<td>4W-S</td>
<td>vortex.9025</td>
<td>3W-M</td>
<td>mesa.227</td>
<td>4W-S</td>
</tr>
<tr>
<td>gap.16913</td>
<td>4W-S</td>
<td>vortex.5877</td>
<td>3W-M</td>
<td>mesa.2297</td>
<td>4W-M</td>
</tr>
<tr>
<td>gap.21010</td>
<td>4W-S</td>
<td>vpr.3463</td>
<td>6W-S</td>
<td>mesa.3731</td>
<td>4W-S</td>
</tr>
<tr>
<td>gcc.264</td>
<td>5W-L</td>
<td>vpr.1350</td>
<td>6W-S</td>
<td>mesa.5172</td>
<td>4W-M</td>
</tr>
<tr>
<td>gcc.873</td>
<td>2W-S</td>
<td>vpr.6120</td>
<td>2W-M</td>
<td>mesa.27859</td>
<td>4W-S</td>
</tr>
<tr>
<td>gcc.473</td>
<td>6W-S</td>
<td>ammp.2423</td>
<td>6W-M</td>
<td>mgrid.2977</td>
<td>6W-L</td>
</tr>
<tr>
<td>gcc.628</td>
<td>6W-S</td>
<td>ammp.2891</td>
<td>6W-L</td>
<td>mgrid.3283</td>
<td>6W-L</td>
</tr>
<tr>
<td>gzip.1361</td>
<td>4W-S</td>
<td>ammp.2945</td>
<td>3W-L</td>
<td>mgrid.3490</td>
<td>6W-L</td>
</tr>
<tr>
<td>gzip.779</td>
<td>4W-S</td>
<td>ammp.3018</td>
<td>3W-L</td>
<td>mgrid.3657</td>
<td>6W-M</td>
</tr>
<tr>
<td>gzip.5030</td>
<td>4W-S</td>
<td>ammp.3783</td>
<td>3W-L</td>
<td>swim.1226</td>
<td>6W-L</td>
</tr>
<tr>
<td>mcf.2018</td>
<td>2W-S</td>
<td>applu.34</td>
<td>6W-L</td>
<td>swim.1312</td>
<td>6W-L</td>
</tr>
<tr>
<td>mcf.3091</td>
<td>2W-S</td>
<td>applu.49</td>
<td>4W-M</td>
<td>swim.1582</td>
<td>6W-L</td>
</tr>
<tr>
<td>parser.5201</td>
<td>2W-M</td>
<td>apsi.2739</td>
<td>4W-S</td>
<td>swim.1583</td>
<td>6W-L</td>
</tr>
<tr>
<td>parser.19972</td>
<td>4W-S</td>
<td>apsi.21688</td>
<td>4W-M</td>
<td>wupwise.12487</td>
<td>6W-M</td>
</tr>
<tr>
<td>parser.24856</td>
<td>4W-S</td>
<td>apsi.49702</td>
<td>4W-M</td>
<td>wupwise.23352</td>
<td>6W-M</td>
</tr>
<tr>
<td>parser.10758</td>
<td>4W-M</td>
<td></td>
<td></td>
<td>wupwise.25364</td>
<td>6W-M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wupwise.97584</td>
<td>6W-M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wupwise.107351</td>
<td>6W-M</td>
</tr>
</tbody>
</table>
Table 4.6: Efficiency (BIPS²/Watt) gain of N-core-type heterogeneous multi-core processors.

<table>
<thead>
<tr>
<th>N</th>
<th>Cores</th>
<th>Average Efficiency Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><strong>2W-S, 2W-L</strong></td>
<td>10.57%</td>
</tr>
<tr>
<td>3</td>
<td><strong>2W-S, 2W-L, 4W-S</strong></td>
<td>14.42%</td>
</tr>
<tr>
<td>4</td>
<td><strong>2W-S, 2W-L, 3W-M, 4W-S</strong></td>
<td>15.26%</td>
</tr>
<tr>
<td>5</td>
<td><strong>1W-L, 2W-S, 2W-L, 3W-M, 4W-S</strong></td>
<td>15.71%</td>
</tr>
</tbody>
</table>
Figure 4.6: Efficiency (BIPS²/Watt) gain achieved by benchmarks on their optimal core. Note upper limit on the Y-axis scale is fixed to 100. Some benchmarks achieve more than 100% gain.
Figure 4.7: Distribution of benchmarks falling in different BIPS$^2$/Watt-gain ranges. Each sector in the chart represent the number of benchmarks falling within the corresponding gain range.

The BIPS$^2$/Watt-gain achieved by various N-core-type heterogeneous multi-core processors are presented in Table 4.6. With only 3 core types, much of the efficiency, with respect to 18 core types, can be achieved. Moreover, 2W-S (Best-1 core for BIPS$^2$/Watt) appears in all N-core-type designs.

The Best-1 core for BIPS$^3$/Watt is 4W-S. Figure 4.8 shows the summary of BIPS$^3$/Watt-gain achieved by employing microarchitecturally diverse cores. The average gain achieved is 18.41% and the maximum gain achieved is 400.63%. Figure 4.9 presents BIPS$^3$/Watt-gain achieved by each benchmark by executing on its optimal core. 4W-S is the optimal core for many benchmarks, hence, their gains are zero. Some benchmarks, for example, gap.3229, mcf.3091, ammp.3018, art.10257, equake.869, mgrid.3657 and swim.1583, achieve significant gains. The pie-chart in Figure 4.10 shows the number of benchmarks falling in different BIPS$^3$/Watt-gain ranges. 21 out of 76 benchmarks achieve more than 90% gain.

The BIPS$^3$/Watt-gain achieved by various N-core-type heterogeneous multi-core processors are presented in Table 4.8. With only 5 core types, much of the efficiency, with
Table 4.7: Best efficiency-oriented core (BIPS²/Watt) for each benchmark in the application space.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip.1873</td>
<td>1W-M</td>
<td>perl.18</td>
<td>3W-M</td>
<td>art.591</td>
<td>1W-L</td>
</tr>
<tr>
<td>bzip.3089</td>
<td>2W-M</td>
<td>perl.1309</td>
<td>3W-S</td>
<td>art.3441</td>
<td>1W-L</td>
</tr>
<tr>
<td>bzip.9277</td>
<td>2W-S</td>
<td>perl.415</td>
<td>3W-S</td>
<td>art.10257</td>
<td>1W-L</td>
</tr>
<tr>
<td>crafty.21513</td>
<td>2W-L</td>
<td>twolf.2482</td>
<td>1W-M</td>
<td>equake.1046</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.12166</td>
<td>2W-L</td>
<td>twolf.27755</td>
<td>2W-S</td>
<td>equake.1093</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.5647</td>
<td>2W-L</td>
<td>twolf.15968</td>
<td>1W-M</td>
<td>equake.2796</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.234</td>
<td>2W-L</td>
<td>twolf.8155</td>
<td>1W-M</td>
<td>equake.463</td>
<td>4W-S</td>
</tr>
<tr>
<td>gap.3229</td>
<td>1W-L</td>
<td>vortex.7432</td>
<td>2W-S</td>
<td>equake.869</td>
<td>2W-L</td>
</tr>
<tr>
<td>gap.15783</td>
<td>2W-S</td>
<td>vortex.9025</td>
<td>3W-M</td>
<td>mesa.227</td>
<td>2W-M</td>
</tr>
<tr>
<td>gap.16913</td>
<td>4W-S</td>
<td>vortex.5877</td>
<td>3W-M</td>
<td>mesa.2297</td>
<td>1W-L</td>
</tr>
<tr>
<td>gap.21010</td>
<td>2W-S</td>
<td>vpr.3463</td>
<td>2W-S</td>
<td>mesa.3731</td>
<td>3W-M</td>
</tr>
<tr>
<td>gcc.264</td>
<td>2W-S</td>
<td>vpr.1350</td>
<td>2W-S</td>
<td>mesa.5172</td>
<td>3W-M</td>
</tr>
<tr>
<td>gcc.873</td>
<td>1W-M</td>
<td>vpr.6120</td>
<td>2W-S</td>
<td>mesa.27859</td>
<td>2W-M</td>
</tr>
<tr>
<td>gcc.473</td>
<td>5W-S</td>
<td>ammp.2423</td>
<td>2W-M</td>
<td>mgrid.2977</td>
<td>2W-L</td>
</tr>
<tr>
<td>gcc.628</td>
<td>2W-S</td>
<td>ammp.2891</td>
<td>2W-L</td>
<td>mgrid.3283</td>
<td>2W-L</td>
</tr>
<tr>
<td>gzip.1361</td>
<td>4W-S</td>
<td>ammp.2945</td>
<td>2W-L</td>
<td>mgrid.3490</td>
<td>2W-L</td>
</tr>
<tr>
<td>gzip.779</td>
<td>2W-S</td>
<td>ammp.3018</td>
<td>2W-L</td>
<td>mgrid.3657</td>
<td>2W-L</td>
</tr>
<tr>
<td>gzip.5030</td>
<td>3W-S</td>
<td>ammp.3783</td>
<td>2W-L</td>
<td>swim.1226</td>
<td>2W-L</td>
</tr>
<tr>
<td>mcf.2018</td>
<td>3W-S</td>
<td>applu.34</td>
<td>4W-S</td>
<td>swim.1312</td>
<td>2W-L</td>
</tr>
<tr>
<td>mcf.3091</td>
<td>1W-S</td>
<td>applu.49</td>
<td>4W-S</td>
<td>swim.1582</td>
<td>2W-L</td>
</tr>
<tr>
<td>parser.5201</td>
<td>2W-S</td>
<td>apsi.2739</td>
<td>4W-S</td>
<td>swim.1583</td>
<td>2W-L</td>
</tr>
<tr>
<td>parser.19972</td>
<td>1W-M</td>
<td>apsi.21688</td>
<td>4W-S</td>
<td>wupwise.12487</td>
<td>2W-S</td>
</tr>
<tr>
<td>parser.24856</td>
<td>2W-S</td>
<td>apsi.49702</td>
<td>4W-S</td>
<td>wupwise.23352</td>
<td>2W-S</td>
</tr>
<tr>
<td>parser.10758</td>
<td>2W-M</td>
<td></td>
<td></td>
<td>wupwise.25364</td>
<td>3W-M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wupwise.97584</td>
<td>2W-S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wupwise.107351</td>
<td>2W-S</td>
</tr>
</tbody>
</table>
Figure 4.8: Summary of efficiency (BIPS$^3$/Watt) gains.

Table 4.8: Efficiency (BIPS$^3$/Watt) gain of various N-core-type heterogeneous multi-core processors.

<table>
<thead>
<tr>
<th>N</th>
<th>Cores</th>
<th>Average Efficiency Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2W-L, 4W-S</td>
<td>9.95%</td>
</tr>
<tr>
<td>3</td>
<td>2W-L, 3W-M, 4W-S</td>
<td>14.26%</td>
</tr>
<tr>
<td>4</td>
<td>2W-L, 3W-M, 4W-S, 4W-M</td>
<td>15.87%</td>
</tr>
<tr>
<td>5</td>
<td>2W-S, 2W-L, 3W-M, 4W-S, 4W-M</td>
<td>17.35%</td>
</tr>
</tbody>
</table>
Figure 4.9: Efficiency (BIPS$^3$/Watt) gain achieved by benchmarks on their optimal core. Note upper limit on the Y-axis scale is fixed to 100. Some benchmarks achieve more than 100% gain.
Figure 4.10: Distribution of benchmarks falling in different BIPS\(^3\)/Watt-gain ranges. Each sector in the chart represents the number of benchmarks falling in the corresponding gain range.

respect to 18 core types, can be achieved. Moreover, 4W-S (Best-1 core for BIPS\(^3\)/Watt) appears in all N-core-type designs.

### 4.4.2 Result Summary

In summary, employing diverse superscalar designs on a die yields both performance and efficiency gains. The highlights of our results are as follows:

- The efficiency gains are much higher than performance-only gains for our application and design space. Some benchmarks achieve more than 100% efficiency gain.

- Most of the gains can be achieved by only employing several diverse cores, for example, three to five core types.

- The N-core-type heterogeneous multi-core processor always contains the Best-1 core. The other N-1 cores removes key microarchitecture bottlenecks, for example, ROB or Issue Queue size, fetch width, etc., in the Best-1 core to capture diversity in the application space’s instruction-level behaviours.

- Different phases of many applications prefer different cores, hence, making a case for dynamic thread migration. The dynamic thread migration can be handled by
Table 4.9: Best efficiency-oriented core (BIPS^3/Watt) for each benchmark in the application space.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
<th>Benchmark Name</th>
<th>Best Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip.1873</td>
<td>1W-M</td>
<td>perl.18</td>
<td>4W-S</td>
<td>art.591</td>
<td>1W-L</td>
</tr>
<tr>
<td>bzip.3089</td>
<td>4W-L</td>
<td>perl.1309</td>
<td>3W-S</td>
<td>art.3441</td>
<td>2W-L</td>
</tr>
<tr>
<td>bzip.341</td>
<td>6W-S</td>
<td>perl.781</td>
<td>3W-M</td>
<td>art.5922</td>
<td>1W-L</td>
</tr>
<tr>
<td>bzip.9277</td>
<td>4W-S</td>
<td>perl.415</td>
<td>2W-M</td>
<td>art.10257</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.21513</td>
<td>2W-L</td>
<td>twolf.2482</td>
<td>2W-S</td>
<td>equake.1046</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.12166</td>
<td>2W-L</td>
<td>twolf.27755</td>
<td>2W-S</td>
<td>equake.1093</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.5647</td>
<td>4W-M</td>
<td>twolf.15968</td>
<td>2W-S</td>
<td>equake.2796</td>
<td>2W-L</td>
</tr>
<tr>
<td>crafty.234</td>
<td>2W-L</td>
<td>twolf.8155</td>
<td>2W-S</td>
<td>equake.463</td>
<td>4W-S</td>
</tr>
<tr>
<td>gap.3229</td>
<td>1W-L</td>
<td>vortex.7432</td>
<td>3W-M</td>
<td>equake.869</td>
<td>2W-L</td>
</tr>
<tr>
<td>gap.15783</td>
<td>2W-M</td>
<td>vortex.9025</td>
<td>3W-M</td>
<td>mesa.227</td>
<td>4W-S</td>
</tr>
<tr>
<td>gap.16913</td>
<td>4W-S</td>
<td>vortex.5877</td>
<td>3W-M</td>
<td>mesa.2297</td>
<td>1W-L</td>
</tr>
<tr>
<td>gap.21010</td>
<td>2W-S</td>
<td>vpr.3463</td>
<td>2W-S</td>
<td>mesa.3731</td>
<td>4W-S</td>
</tr>
<tr>
<td>gcc.264</td>
<td>2W-S</td>
<td>vpr.1350</td>
<td>2W-S</td>
<td>mesa.5172</td>
<td>4W-S</td>
</tr>
<tr>
<td>gcc.873</td>
<td>2W-S</td>
<td>vpr.6120</td>
<td>2W-M</td>
<td>mesa.27859</td>
<td>4W-S</td>
</tr>
<tr>
<td>gcc.473</td>
<td>5W-S</td>
<td>ammp.2423</td>
<td>4W-M</td>
<td>mgrid.2977</td>
<td>4W-M</td>
</tr>
<tr>
<td>gcc.628</td>
<td>4W-S</td>
<td>ammp.2891</td>
<td>2W-L</td>
<td>mgrid.3283</td>
<td>3W-L</td>
</tr>
<tr>
<td>gzip.1361</td>
<td>4W-S</td>
<td>ammp.2945</td>
<td>2W-L</td>
<td>mgrid.3490</td>
<td>2W-L</td>
</tr>
<tr>
<td>gzip.779</td>
<td>2W-S</td>
<td>ammp.3018</td>
<td>2W-L</td>
<td>mgrid.3657</td>
<td>5W-L</td>
</tr>
<tr>
<td>gzip.5030</td>
<td>2W-S</td>
<td>ammp.3783</td>
<td>2W-L</td>
<td>swim.1226</td>
<td>2W-L</td>
</tr>
<tr>
<td>mcf.2018</td>
<td>3W-S</td>
<td>applu.34</td>
<td>4W-M</td>
<td>swim.1312</td>
<td>2W-L</td>
</tr>
<tr>
<td>mcf.3091</td>
<td>1W-S</td>
<td>applu.49</td>
<td>4W-M</td>
<td>swim.1582</td>
<td>2W-L</td>
</tr>
<tr>
<td>parser.5201</td>
<td>2W-S</td>
<td>apsi.2739</td>
<td>4W-S</td>
<td>swim.1583</td>
<td>2W-L</td>
</tr>
<tr>
<td>parser.19972</td>
<td>2W-S</td>
<td>apsi.21688</td>
<td>4W-M</td>
<td>wupwise.12487</td>
<td>3W-M</td>
</tr>
<tr>
<td>parser.24856</td>
<td>2W-S</td>
<td>apsi.49702</td>
<td>4W-S</td>
<td>wupwise.23352</td>
<td>3W-M</td>
</tr>
<tr>
<td>parser.10758</td>
<td>2W-M</td>
<td>wupwise.25364</td>
<td>5W-L</td>
<td>wupwise.97584</td>
<td>3W-M</td>
</tr>
</tbody>
</table>

65
software (operation system or compiler/programmer) or hardware (power management unit). However, thread migration has both performance and energy penalties incurred due to moving architectural register state and additional cold misses suffered in caches and predictors. Engineering solutions are required to mitigate this penalty to reap the most benefit of employing diverse cores.

- Practically all current systems, from PCs to mobile devices and workstations to the Exascale supercomputers, are now power limited. Our results demonstrate that employing microarchitecturally diverse cores is certainly an approach to overcome the power limitation.

## 4.5 Comparing Heterogeneous Multi-core with DVFS

Dynamic Voltage and Frequency Scaling (DVFS) is a technique to dynamically adjust the voltage-frequency pair for a core. DVFS exploits the execution characteristics of an application to trade-off performance and energy by adjusting the supply voltage and the operating frequency. Energy is quadratically proportional to voltage, and frequency is directly proportional to voltage. There is a large body of research that has, in the past, explored the applicability of DVFS in different design contexts [97] [71] [77].

Recently, high-performance processors, constrained by limited power budget, are using DVFS in a multi-core to provide the best of throughput and latency. Cores operate at the nominal or sub-nominal voltage when multiple threads are available to execute. This allows multiple cores to remain powered-on simultaneously to deliver high program throughput. When single-thread performance is required, only a few cores remain active and the cores operate at a voltage that is higher than the nominal voltage. The higher voltage leads to higher clock frequency, hence, reduced single-thread execution latency. A hardware DVFS mechanism, decides the number of active cores and their operating voltage(s) at run-time. Intel’s Turbo Boost technology present in their recent commercial products, is one such example [20]. With Turbo Boost, a core’s clock frequency is increased based on the estimated power consumption, the number of active cores, and the core temperature.

Heterogeneity in core microarchitecture and DVFS are orthogonal techniques, and
they can be applied simultaneously [51]. However, we want to understand if heterogeneous multi-core can provide single-thread performance gains above and beyond what DVFS-enabled homogeneous multi-core can provide. DVFS is a well-understood technique by designers and doesn’t need to employ diverse cores. The fundamental question is whether or not designing multiple cores is worth the engineering effort. Specifically, we are interested in the context of high-performance processors designed within a power envelope. Moreover, we would like to highlight that recent work has shown diminishing returns of DVFS for future designs. As transistor dimensions have shrunk to a few nanometers (nm) and the difference between transistor’s threshold and operating voltage has reduced to a few millivolts (mV), the usefulness of DVFS has reduced significantly [55]. Increasing operating voltage has two major concerns: 1) it increases the energy consumption quadratically and 2) it might impact the reliability of nano-scale transistors. These concerns make the comparison between heterogeneous multi-core and DVFS even more compelling.

We consider two multi-core designs for our study: a homogeneous multi-core with DVFS enabled on each core (referred to as Design-A) and a heterogeneous multi-core (referred to as Design-B). BIPS is used for measuring performance and BIPS³/Watt is used for measuring efficiency. 4W-S is the core used for Design-A (following from the results discussed in the previous section). The 4W-S core in Design-A can operate at three voltage-frequency points as shown in Table 4.10. The first row in the table refers to our baseline operating voltage. The next two rows reflect the two voltage-boosted DVFS points used in the experiment. The Issue and Register Read pipeline stages in 4W-S are its timing-critical stages. We performed detailed SPICE simulation on the timing-critical paths of these stages to find the new clock frequency for an increased operating voltage. FabMem [80] was used to generate the SPICE netlists for RAMs and CAMs in the timing-critical stages of 4W-S.

Design-B refers to the 5-core-type heterogeneous multi-core in Table 4.8. All the cores in Design-B operate at the nominal voltage of 1.1V.

Figure 4.11 presents single-thread performance gains achieved by Design-A and Design-B across all benchmarks. The gain is measured with respect to the 4W-S core operating
Table 4.10: Voltage-frequency pairs used for the core in Design-A. The nominal operating voltage for our 45nm technology library is 1.1V.

<table>
<thead>
<tr>
<th>Design Point Label</th>
<th>Voltage (V)</th>
<th>Frequency (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design-A_1.1V</td>
<td>1.1</td>
<td>1.67</td>
</tr>
<tr>
<td>Design-A_1.2V</td>
<td>1.2</td>
<td>1.78</td>
</tr>
<tr>
<td>Design-A_1.3V</td>
<td>1.3</td>
<td>1.93</td>
</tr>
</tbody>
</table>

at the nominal voltage, i.e., Design-A_1.1V is the baseline. Two DVFS points are used for Design-A and separate results for each point are presented in the figure. Increasing operating voltage from 1.1V to 1.3V leads to a 15.6% increase in clock frequency. In an ideal world, with perfect branch prediction, ideal caches, etc., an application should achieve linear performance gain with increasing frequency. The performance gain achieved by DVFS, across all benchmarks, is below 15.6%. The benchmark gcc.473 achieves 14.2% gain and the benchmark equake.463 achieves 14.1% gain. While increasing frequency accelerates the execution of serial dependence chains in the benchmark, it doesn’t remove any microarchitecture bottleneck. For example, a benchmark suffering from high branch misprediction rate derives lesser benefit from the increased frequency. Moreover, benchmarks with a high percentage of long-latency instructions hardly get any benefit, e.g., art.591, equake.2796, and mcf.3091. Diverse cores in Design-B eliminate microarchitectural bottlenecks of the 4W-S core for certain ILP characteristics. This leads to significant performance gains achieved by many benchmarks, e.g., art.591, ammp.3018, crafty.234, and bzip.3089. Moreover, the gain in performance is efficient as well (see Figure 4.9 for efficiency gains achieved by these benchmarks compared to executing on the 4W-S core).
Figure 4.11: Performance gain comparison between heterogeneous multi-core and DVFS-enabled homogeneous multi-core. The gains are presented with respect to Design-A operating at the nominal voltage.
Chapter 5

Time-to-Market Sensitive
Superscalar Processor Design Approach

FabScalar addresses the design-effort problem by automating the generation of register-transfer-level (RTL) designs of entire superscalar cores. A good quality RTL design is an essential starting point of the chip design cycle, including design tuning for performance and power, verification, and physical design. By generating the RTL design of an entire core, FabScalar partially addresses the design-effort problem but it does not remove the burden of implementing a superscalar core completely. This thesis systematically investigates the additional RTL tuning and physical design effort required to meet performance and power requirements for future time-to-market sensitive application processors (APs).

The context of the design-effort/quality trade-off of physical design of superscalar processors has changed for APs (refer to Chapter 2 for detailed background): (1) Frequencies of APs are not as high as desktop/laptop/server processors, due to form factor and power. (2) The intrinsic speed afforded by advanced technology can be leveraged to meet slightly less ambitious frequency targets. (3) Decades of investment in CAD has reduced the gap between automated and manual design. The chapter explores the viability of heavily relying on automated synthesis and place-and-route (SPR), using RTL of several processors configured similarly to commercial APs. First, SPR is used to gauge the design quality (frequency, power, and area) achievable with minimum design
effort. Then, different techniques to increase frequency are implemented. The techniques fall into three classes: 1) memory structure optimization (since highly-ported memories are pervasive in superscalar processors), 2) physical-design motivated microarchitecture adjustments, and 3) circuit-level optimization. We measure each technique’s individual impact: its frequency contribution, its impact on instructions-per-cycle (IPC), power, and area, and its required design effort (designers time). This data enables constructing a frontier of combinations that minimize design effort for a given design quality.

Section 5.1 outlines the evaluation methodology. Section 5.2 describes and evaluates all the optimizations that we performed. Finally, Section 5.3 presents a frontier of combinations that minimize design effort for a given design quality.

5.1 Evaluation Methodology

5.1.1 Synthesizable Cores

The objective of this chapter is to understand and characterize a low-effort physical design methodology for superscalar processors. This study requires RTL designs of whole superscalar processors. Using the FabScalar toolset, we generated RTL designs of two reference cores, a 2-way and a 4-way superscalar processor with respect to fetch width. These are referred to as Core-2W and Core-4W, respectively. Table 5.1 shows their microarchitectural configurations. We determined the microarchitectural configurations of Core-2W and Core-4W based on two guiding principles. Firstly, the two cores should represent the spectrum of commercial application processors in terms of superscalar width and key structures for exposing and exploiting instruction-level parallelism (ILP). The fetch width, issue width, and instruction window size of Core-2W and Core-4W are based closely on the AMD Bobcat and ARM Cortex-A15, respectively. Secondly, the microarchitectural resources should be balanced in terms of no one structure being the sole limiter of instructions-per-cycle (IPC). We performed extensive IPC simulations using SPEC2000 integer benchmarks to balance the design. The processor widths and instruction window size (i.e., reorder buffer and physical register file) are set as per above, and then all other structures were downsized until the point at which they noticeably reduced IPC. The pipeline depth of both cores is the shallowest that can be generated by the FabScalar toolset, yielding the highest IPC for a given superscalar complexity (stage widths and
Table 5.1: Microarchitecture configurations of two reference cores used for this work. The configuration only reflects the integer pipeline. BTB: Branch Target Buffer, BPB: Branch Prediction Buffer, RAS: Return Address Stack. FU mix: S=simple ALU, C=complex ALU, B=branch, Ld/St=load/store pipeline.

<table>
<thead>
<tr>
<th></th>
<th>Core-2W</th>
<th>Core-4W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Width</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Issue Width</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Functional Unit Mix</td>
<td>1S/C, 1B, 1Ld/St</td>
<td>1S, 1S/C, 1B, 1Ld/St</td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>Load / Store Queues</td>
<td>8 / 8</td>
<td>12 / 12</td>
</tr>
<tr>
<td>Phys. Register File Size</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>BTB Size (# entries)</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>BPB Size (# entries)</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>RAS Size</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>L1 I-cache / L1 D-cache (KB)</td>
<td>16 / 16</td>
<td>32 / 32</td>
</tr>
<tr>
<td>fetch-to-execute pipeline depth (simple / load-store)</td>
<td>8 / 9</td>
<td>8 / 9</td>
</tr>
</tbody>
</table>

structure sizes).

Figure 5.1 shows the pipelines of (a) Core-2W and (b) Core-4W. All instructions flow through 10 canonical pipeline stages. All stages take 1 cycle, except load/store and complex ALU instructions take multiple cycles in the Execute Stage: 2 and 3 cycles, respectively. Core-2W is 2 instructions wide in the Fetch-1 through Dispatch Stages. It has three execution lanes, hence, Issue through Writeback are 3 instructions wide. Core-4W is 4 instructions wide in the Fetch-1 through Dispatch Stages. It has four execution lanes (a fourth lane providing an extra simple ALU), hence, Issue through Writeback are 4 instructions wide as well. The Retire Stage in both Core-2W and Core-4W examines a retire bundle of up to 4 instructions.

Table 5.2 shows the major memory structures and the number of read and write ports for each structure, for each of Core-2W and Core-4W. For CAMs, read port means match port. The characters in parentheses generalize the number of ports as a function of fetch
width (F), dispatch width (D), issue width (X, the number of execution lanes), number of load/store execution lanes (M), and retire width (R).

5.1.2 ASIC Design Flow

The commercial CAD tools used for all experiments are listed in Table 3.1 in Chapter 3. All designs are implemented using the Nangate 45nm open cell library. The Nangate library provides liberty-formatted timing and power libraries, geometric libraries in Library Exchange Format (LEF), and simulation libraries in Verilog and Spice. We additionally use the proprietary ARM 130nm cell library for studying the advantage technology scaling provides to achieve fast time-to-market. For experiments involving microarchitecture-level optimization for improving physical design results, we also made modifications to the baseline RTL. To test the RTL, we followed a similar methodology as described in Chapter 3, simulating 100 million instruction SimPoints of SPEC integer benchmarks. All EDA tools have been run on an Intel Xeon X5560, a relevant piece of information given that time to run EDA tools is included in design-effort quantification.

The RTL design is iterated through synthesis and placement & routing until the de-
Table 5.2: Major memory structures in reference cores.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Read Ports</th>
<th>Write Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Core-2W/Core-4W</td>
<td>Core-2W/Core-4W</td>
</tr>
<tr>
<td>L1 Instruction Cache (2-way interleaved)</td>
<td>1 shared read/write port per bank</td>
<td></td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td>1 per bank</td>
<td>1 per bank</td>
</tr>
<tr>
<td>Branch Prediction Buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>2 / 4 (D)</td>
<td>2 / 4 (F)</td>
</tr>
<tr>
<td>Rename Map Table</td>
<td>4 / 8 (2D)</td>
<td>2 / 4 (D)</td>
</tr>
<tr>
<td>Free List</td>
<td>2 / 4 (D)</td>
<td>4 (R)</td>
</tr>
<tr>
<td>Architectural Map Table</td>
<td>4 (R)</td>
<td>4 (R)</td>
</tr>
<tr>
<td>Active List (ROB)</td>
<td>4 (R)</td>
<td>2 / 4 (D)</td>
</tr>
<tr>
<td>Memory Dependence Predictor</td>
<td>2 / 4 (D)</td>
<td>1</td>
</tr>
<tr>
<td>Issue Queue Wakeup CAM</td>
<td>3 / 4 (X)</td>
<td>2 / 4 (D)</td>
</tr>
<tr>
<td>Issue Queue Payload RAM</td>
<td>3 / 4 (X)</td>
<td>2 / 4 (D)</td>
</tr>
<tr>
<td>Physical Register File</td>
<td>6 / 8 (2X)</td>
<td>3 / 4 (X)</td>
</tr>
<tr>
<td>Load Queue CAM</td>
<td>1 (M)</td>
<td>1 (M)</td>
</tr>
<tr>
<td>Store Queue CAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Queue RAM</td>
<td>1 (M)</td>
<td>different fields: 1 (M) or 2 / 4 (D)</td>
</tr>
<tr>
<td>Store Queue RAM</td>
<td>1 (M)</td>
<td></td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>1 (M)</td>
<td>1</td>
</tr>
</tbody>
</table>
design meets its timing, power, and area objectives. Clock frequency, power, and area are reported after placement & routing, i.e., all results reflect the full layout using Cadence SoC Encounter. The power results assume 20% switching activity for all of the logic, and they include both dynamic and leakage power.

The starting point for each core is a fully-synthesized design, including all memory structures (except the L1 instruction and data caches). This means memory structures are implemented with flip-flops in the baseline physical design. We subsequently consider synthesis to latches, followed by various strategies that leverage SRAMs produced by memory compilers. We use the FabMem memory compiler because Nangate does not provide a memory compiler nor do we have access to a memory compiler for a 45nm foundry process.

For the L1 instruction and data caches, timing is obtained from CACTI 5.1 adjusted to the FreePDK BSIM4 predictive technology model and the FabMem memory compiler is used to estimate the LEF geometry for layout.

5.1.3 Measuring IPC

For measuring IPC, we use the SPEC2000 integer benchmark suite and the cycle-accurate C++ simulator from the FabScalar toolset.

5.2 Results

At the outset, we synthesize and place-and-route Core-2W and Core-4W. All memory structures, except for the L1 caches, are synthesized to flip-flops. This initial exercise (Section 5.2.1) determines the frequency, power, and area that is possible with minimum design effort. Next, we explore different techniques to improve frequency in a cost-effective way, i.e., with low design effort. We explore three classes of techniques: 1) optimizing memory structures (Section 5.2.2), 2) adjusting the microarchitecture (Section 5.2.3), and 3) optimizing circuits (Section 5.2.4). All the while, we leverage SPR as much as possible. We measure each technique’s individual impact: its frequency contribution, its impact on IPC (for microarchitecture adjustments), power, and area, and its required design effort. To quantify design effort, we documented the hours of effort spent by the
Table 5.3: Baseline results for Core-2W and Core-4W designs using 45nm cell library.

<table>
<thead>
<tr>
<th></th>
<th>Frequency (MHz)</th>
<th>Power (mW/MHz)</th>
<th>Area (mm²)</th>
<th>Design Effort (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core-2W</td>
<td>834</td>
<td>0.433</td>
<td>1.048</td>
<td>60</td>
</tr>
<tr>
<td>Core-4W</td>
<td>667</td>
<td>0.635</td>
<td>2.052</td>
<td>270</td>
</tr>
</tbody>
</table>

author and collaborators, including implementation time and time to set up and run the CAD tools (including iteration of SPR). For microarchitecture adjustments, we also report the number of lines of verilog code that were modified and added to the baseline RTL [11]. All of the results reported are after full layout of the design using Cadence SoC Encounter.

5.2.1 Baseline: minimum design effort

To establish the baseline frequency, power, and area, we implemented Core-2W and Core-4W using SPR with no modifications to the FabScalar-generated RTL. Table 5.3 shows baseline results using the 45nm cell library. The design effort column quantifies the total hours taken by the SPR tools to achieve the reported frequency. Multiple iterations of SPR were needed to achieve the frequency.

For Core-2W, the baseline frequency is 834 MHz and the baseline effort is 60 hours. Increasing superscalar complexity from Core-2W to Core-4W decreases frequency by 20% (from 834 MHz to 667 MHz), increases area by 96%, and increases design effort by a factor of $4.5\times$, from 60 hours to 270 hours.

Figure 5.2a shows the IPCs of various SPEC SimPoints on Core-2W and Core-4W. The number associated with each benchmark is its SimPoint id. On average, the IPC of Core-4W is 37% better than the IPC of Core-2W.

In both pipelines, Fetch-1 is the most timing-critical stage. Its longest path is reading the banked BTB for all instructions in the fetch bundle, identifying the first predicted-taken branch instruction in the fetch bundle, and, if it is a call instruction, then updating the RAS with the call’s return address. Investigating further, Figure 5.2b shows the slack...
in each pipeline stage. Slack reflects the logic imbalance that exists in different stages. The next most critical stages in both cores are Issue, Register Read, and the LSU (the second stage of load/store execution which involves searching the LQ/SQ). Compared to Core-2W, the Register Read and LSU slacks are lower in Core-4W, whereas slacks of other stages increased. This reflects a significant increase in complexity of the Register Read and LSU stages. In Register Read, the culprits are a larger physical register file and longer and more complex bypasses (spanning four execution lanes). The larger SQ, for store-to-load forwarding, impacts the LSU. In general, SPR tools are very advanced in optimizing ALUs: it was very easy to achieve approximately 2.5GHz for the ALUs. CAD vendors provide pre-designed libraries for fast carry-save adders, carry-lookahead adders, and other common elements. However, SPR tools perform poorly for wire-dominated stages. For instance, the author spent a considerable amount of time to make the Register Read stage routable. The place-and-route algorithm is divided into phases, most notably, placement and detailed routing [92]. Since both of these phases involve problems that are either NP-Hard or NP-Complete [75] [76] [92], heuristics are used to obtain an approximate solution. In the placement phase, logic elements are placed based on their connectivity but routing resources may not be considered in detail. This may lead to underestimation of the routing resources and the detailed routing phase may not yield a solution, requiring a repetition of the placement phase.

We also implemented Core-2W using the 130nm cell library and the results are presented in Table 5.4. The large difference in Core-2W frequencies between 45nm and 130nm clearly support our motivation. The intrinsic speed afforded by advanced technology can certainly be leveraged to meet AP-class frequency targets in time-to-market sensitive designs.
Figure 5.2: (a) IPCs of Core-2W and Core-4W for different SPEC SimPoints. (b) Slack in each pipeline stage as a percentage of cycle time.

5.2.2 Optimizing memory structures

As shown in Table 5.2, multi-ported memories are pervasive in superscalar processors. They are often the dominant cycle time, power, and area contributors within their respective pipeline stages. Therefore, in high-end superscalar processors, each memory is typically a custom-designed SRAM or CAM. Designers of the IBM POWER7 processor reported that custom-designing an SRAM takes one to two person-years [35]. This approach is not viable for time-to-market sensitive application processors. It may be viable by staffing more engineers but doing so impacts cost.

In the baseline implementation, we synthesized memories to flip-flops. This is certainly a low design-effort approach but memories synthesized to flip-flops suffer from multiple inefficiencies. A flip-flop in a typical standard cell library has 25 to 30 transistors. An SRAM cell, on the other hand, uses 6 to 8 transistors per bit, yielding lower area and lower power. Moreover, large memories implemented with flip-flops suffer long access times.
Figure 5.3: Frequency, power, and area of different depth memories synthesized to flip-flops vs. implemented in SRAM (from FabMem). Depth is varied from 32 to 256 words and the word size is 4 bytes. Each memory has 2-read and 2-write ports.

Figure 5.3 compares frequency, power, and area of memories synthesized to flip-flops and the same memories implemented in SRAM using the FabMem tool. All the memories are multi-ported (2 read, 2 write) and 4 bytes wide. Interestingly, delays of flip-flop-based memories are better or comparable for smaller sizes (less overhead at small sizes, e.g., no sense amps) but for larger sizes delays are much worse. For all configurations, flip-flop based memories are less efficient in terms of power and area. The inefficiency is low for small memories and grows significantly for large memories (the trend is similar for power and area). Moreover, the automated place & route tool suffers in handling many wires localized in a small space. Multi-ported memories implemented with flip-flops have many local wires, attributed to large fan-in and fan-out of individual flip-flops because of multiple decoders (for writes) and multiplexors (for reads). Wire routing in custom-designed SRAM is optimized manually.
Implementing memories using level-sensitive latches

As a first approach to optimize memories, we implemented memories using level-sensitive latches. Latches are comprised of 12 to 14 transistors, about half the size of flip-flops. The drawback of using latches is that write-after-read hazards must be explicitly handled in certain pipeline stages. For example, the Rename Stage leverages the fact that, in a flip-flop implementation, RMT writes are synchronous with the clock edge and happen at the end of the clock cycle. Therefore, writes by younger instructions in the rename bundle do not interfere with reads by older instructions in the rename bundle, despite accessing the same logical register. With latches, however, the writes happen during the second half of the cycle, potentially interfering with concurrent reads. The solution is to defer the rename bundle’s RMT updates to the next cycle, i.e., pipeline the RMT reads and writes from the same rename bundle. In turn, deferring the writes requires a second level of RMT bypasses to pass tags from the current rename bundle to the rename bundle that follows it; such bypasses already exist for intra-bundle dependences.

Thus, using latches required RTL modifications, incurring additional design and verification effort. RTL modifications were purely local to the affected modules and had no global impact.

Implementing memories using foundry memory compilers

In the second memory optimization, we explore using foundry memory compilers (hypothetically, since we use FabMem in place of a commercial memory compiler). The advantage of using a memory compiler is that it requires less effort than custom-designing an SRAM from scratch. Unfortunately, memory compilers are typically limited to one or two ports. (FabMem is not limited, but we are using it as if it were limited, for demonstration.)

To work around port limitations, the effect of more ports can be achieved by replicating SRAMs. In fact, this work-around is often employed in FPGAs with dual-ported block RAMs [32] [33] [74]. Figure 5.4 shows how it works. Figure 5.4a shows a 2R1W SRAM implemented with two 1R1W SRAMs. A write happens to both SRAMs so that two reads can access the same data in parallel. Figure 5.4b shows a 1R2W SRAM, also implemented with two 1R1W SRAMs. Each SRAM reflects writes from only one write port. A read consults the ram_select_vector (implemented with flip-flops) to know which
SRAM was most recently written at the selected row. Figure 5.4c combines the two cases to construct a 2R2W SRAM. More generally, the number of 1R1W SRAMs needed is the number of logical read ports times the number of logical write ports. If SRAM building blocks with more than two ports are available, then the degree of replication required is less.

**Memory optimization results**

Figure 5.5 shows the frequency, power, and area of three pipeline stages individually: Rename, Register Read, and Issue. The only variation is in the implementation of their memories, specifically, the RMT in the case of Rename, Physical Register File in the case of Register Read, and payload RAM in the case of Issue.
Using latches instead of flip-flops substantially reduces delay in Rename, moderately in Register Read, and not much in Issue. Power is significantly reduced for Register Read and Issue, but increases a bit for Rename. Unexpectedly, area for flip-flops and latches are about the same. In hindsight, the memories are probably dominated by wires, decoders, and muxes to access the flip-flop and latch arrays with many ports.

The next three points implement the highly-ported memories by replicating 1R1W, 2R2W, or 3R3W building blocks generated by FabMem. The final point, Custom, is meant to represent a custom SRAM with the exact number of ports required by the core, also generated by FabMem. 3R3W cannot be used for Core-2W’s Rename and Issue Stages, hence, the corresponding points are intentionally missing. For Core-2W, flip-flops and latches have similar or better access times than SRAM blocks. The SRAM blocks yield much lower power for two of the stages, however. The 1R1W and 2R2W SRAM blocks yield worse area than latches and flip-flops. This is likely due to the fact that Core-2W’s memories are sufficiently small with moderate number of ports (compared to Core-4W), such that the overhead of SRAM replication is too high, yielding worse access times and areas.

It seems the one case where SRAM blocks are favored for frequency, is in Core-4W’s Physical Register File. It is the largest memory considered in this section and has 12 ports. Consequently, for Core-4W Register Read, 2R2W yields the highest frequency, low power, and area similar to latches and flip-flops.

Table 5.5 shows the design effort (lines of RTL and hours) for the different memory implementations of the Rename, Register Read, and Issue Stages.

5.2.3 Adjusting the microarchitecture

Pipelining timing critical stages

The frequency achieved by SPR can be boosted by pipelining timing-critical stages, at the cost of additional power, area, and design effort. Increasing pipeline depth may also negatively impact IPC. In this section, we evaluate this cost/benefit tradeoff of pipelining.
Figure 5.5: Frequency, power, and area comparisons of different memory implementations, for the RMT in Rename Stage, Phys. Register File in Register Read Stage, and payload RAM in Issue Stage.
Table 5.5: Design effort for the different memory implementations of Rename, Register Read, and Issue.

<table>
<thead>
<tr>
<th></th>
<th>Core-2W: Rename Stage</th>
<th>Core-4W: Rename Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTL Lines Added</td>
<td>Design Effort (Hours)</td>
</tr>
<tr>
<td>Latches</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>SRAM-1R1W</td>
<td>281</td>
<td>15</td>
</tr>
<tr>
<td>SRAM-2R2W</td>
<td>113</td>
<td>15</td>
</tr>
<tr>
<td>SRAM-3R3W</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Core-2W: Register Read Stage</td>
<td>Core-4W: Register Read Stage</td>
</tr>
<tr>
<td></td>
<td>RTL Lines Added</td>
<td>Design Effort (Hours)</td>
</tr>
<tr>
<td>Latches</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>SRAM-1R1W</td>
<td>522</td>
<td>16</td>
</tr>
<tr>
<td>SRAM-2R2W</td>
<td>355</td>
<td>16</td>
</tr>
<tr>
<td>SRAM-3R3W</td>
<td>124</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Core-2W: Issue Stage</td>
<td>Core-4W: Issue Stage</td>
</tr>
<tr>
<td></td>
<td>RTL Lines Added</td>
<td>Design Effort (Hours)</td>
</tr>
<tr>
<td>Latches</td>
<td>28</td>
<td>10</td>
</tr>
<tr>
<td>SRAM-1R1W</td>
<td>232</td>
<td>15</td>
</tr>
<tr>
<td>SRAM-2R2W</td>
<td>168</td>
<td>15</td>
</tr>
<tr>
<td>SRAM-3R3W</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 5.6 shows the pipelining experiments that we performed for both cores. Fetch-1-Pipe1 delays pushing a call’s return target onto the RAS by one cycle. If the return instruction is in the following fetch bundle, it obtains its target from a newly-introduced RAS bypass. Rename-Pipe1 and Rename-Pipe2 pipelines the Rename Stage into two or three cycles, and adds more levels of bypassing to handle cross-rename-bundle dependences. Issue-Pipe1 involves splitting wakeup-select-payloadRead logic into wakeup-select and payloadRead. This maintains a single-cycle wakeup-select loop, ensuring single-cycle producers and their consumers still execute in consecutive cycles. The cost, however, is that the select logic datapath is widened to include not only instructions’ request/grant signals but also their tags. The select logic must simultaneously generate grants and steer the granted instruction’s tag to the wakeup port, whereas previously the tag was obtained from the payload RAM after the select logic. RegRead-Pipe1 and RegRead-Pipe2 pipeline the Physical Register File into 2 and 3 stages, respectively. This further complicates the bypass network. Pipelining the LSU adds no additional bypass logic but increases the load-to-use latency. The Decode and Dispatch Stages are straightforward to pipeline, they only require additional pipeline registers. We pipelined them as well and account for their overheads in the results.

The graphs on the left-hand side of Figure 5.6 show the per-stage frequency improve-
Figure 5.6: (left) Frequency increase and (right) costs, for individual stages.

ments of pipelining for the two cores. In both cores, Rename, Issue, Register Read, and LSU are initially close in frequency. The adjustment made to Issue is less effective than the adjustments made to the other three stages. Fetch-1 was already the most critical in both cores, and its adjustment did not substantially help. Consequently, these experiments show that Fetch-1 and Issue remain frequency bottlenecks. Therefore, we apply further adjustments to these in the sub-sections that follow.

The graphs on the right-hand side of Figure 5.6 show the power and area increase on a per-stage basis, and relative design effort contribution for each stage. Rename and Register Read suffer the largest power and area increases, which is not unexpected due to the increase in bypass complexity in both stages.
Figure 5.7: Distribution of RTL lines added and design effort for pipelining the various stages of Core-2W.

Figure 5.7 shows the distribution of design effort and RTL lines added for pipelining the various stages of Core-2W. Core-4W’s distribution (not shown) follows a similar trend as Core-2W. It took one person 316 hours to pipeline all the stages in Core-2W. Pipelining Register Read took most of the design effort, followed by Dispatch, Issue and LSU. Interestingly, design effort contribution of a stage correlates with the number of RTL lines added to pipeline that stage. Bazeghi et al. made a similar observation [11]. Although Dispatch is an exception, Dispatch involves plenty of control logic to write into all the back-end stages, i.e., IQ in Issue, LQ/SQ in LSU, and Active List (ROB), and generating appropriate stall (or ready) signals for the front-end.
Table 5.7: IQ partitioning schemes in Core-2W and Core-4W. All the schemes employ Issue-Pipe1 implementation.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core-2W-IQP</td>
<td>Partitioned the IQ into INT:8, AGEN:8</td>
</tr>
<tr>
<td>Core-4W-IQP1</td>
<td>Partitioned the IQ into INT:16, AGEN:8</td>
</tr>
<tr>
<td>Core-4W-IQP2</td>
<td>Partitioned the IQ into INT0:8, INT1:8, AGEN:8</td>
</tr>
</tbody>
</table>

Distributing the Issue Queue

The monolithic IQ is best for IPC but it is timing critical \cite{69}. We explore different IQ partitioning schemes for Core-2W and Core-4W (shown in Table 5.7): Core-2W-IQP, Core-4W-IQP1 and Core-4W-IQP2. In Core-2W-IQP, the AGEN IQ holds load and store instructions and the INT IQ holds non-memory instructions. The same applies for Core-4W-IQP1. In Core-4W-IQP2, non-memory instructions are additionally split across INT0 and INT1 partitions based on a simple round-robin based policy for load balancing. In addition, for all schemes, cross-partition tag-broadcast (wakeup) requires an additional cycle, preventing a single-cycle producer and its consumer in a different partition from executing in consecutive cycles.

Figure 5.8 and Figure 5.9 show the frequency gain and IPC loss, respectively, of IQ partitioning. (The monolithic IQ designs for Core-2W and Core-4W are referred to as Core-2W-IQ and Core-4W-IQ, respectively). Note that all designs implement Issue-Pipe1. Core-2W-IQP achieves 17\% higher frequency but the IPC degrades by 5\%, on average, compared to Core-2W-IQ. Core-4W-IQP1 achieves 28.5\% higher frequency but the IPC degrades by 6\%, on average, compared to Core-4W-IQ. Core-4W-IQP2 achieves 50\% higher frequency but the IPC degrades by 8\%, on average, compared to Core-4W-IQ. Although average IPC degradation of Core-4W-IQP2 is only 8\%, there are a few benchmarks that suffer significantly, e.g., bzip.3089, bzip.9227 and parser.5201.

The partitioned IQ required additional design effort of implementing steering logic in the Dispatch Stage. It took one person 40 hours to implement the steering logic and the required changes in the Issue Stage to support the partitioning scheme.
Figure 5.8: Frequency achieved due to IQ partitioning schemes in Core-2W and Core-4W.

Figure 5.9: %IPC reduction of each partitioned IQ design with respect to (wrt) monolithic IQ.
Fetch-1

Fetch-1 remains a stubborn bottleneck because of the branch prediction logic. The other pipeline stages are able to achieve more than 1500MHz using various adjustments described thus far. The complexity of branch prediction logic in time-to-market sensitive processors will grow further to support deeper pipelines. Unfortunately, pipelining the branch prediction logic requires sophisticated algorithms [79] that balloon design effort as documented by others that implemented them in RTL [36].

We propose using multiple frequency domains (MFD) [31] for time-to-market sensitive designs to remove the Fetch-1 bottleneck. The Fetch-1 stage can operate at a slower frequency than the rest of the pipeline. State-of-art CAD tools are very advanced in handling MFD and cross frequency domain communication [19]. To compensate for a slower frequency, the fetch width can be increased. Thus, the Fetch-1 stage will deliver fetch bundles at a lower frequency but more instructions per fetch bundle. We explored two MFD-based design choices for Core-2W using Cadence SoC Encounter: Core-2W-MFD-Fetch-2W and Core-2W-MFD-Fetch-4W. In the former, Fetch-1 does not increase fetch bundle size (2) with respect to Core-2W and is clocked at half the frequency. In the latter, Fetch-1 doubles its fetch bundle size (to 4) and is clocked at half the frequency. We account for additional latency introduced by cross-domain synchronization buffers in IPC simulation.

Referring to Figure 5.10, Core-2W-MFD-Fetch-2W degrades IPC by 30%, on average, with respect to Core-2W. The average IPC gradation for Core-2W-MFD-Fetch-4W is only 3.5%. Increasing the Fetch-1 complexity from 2-wide to 4-wide increases the power of Core-2W by 6%.

5.2.4 Optimizing the circuit

The design space for circuit-level optimization techniques is very large [10] [28] [35]. The techniques range from using dynamic logic for reducing delay, specializing transistors for specific datapaths and arrays, layout compaction for reducing area, etc. Unfortunately, most of the circuit-level optimizations have large design effort associated with them. In this section, we present a low design effort circuit optimization that we performed to
improve clock frequency.

**Optimizing frequently-used cells in library**

Analyzing the gate-level netlists from 45nm and 130nm implementations, we observe that approximately 10% of the standard cell library elements constitutes more than 80% of the netlist (placed & routed design). We redesigned a set of frequently used cells, 8 cells in the 45nm cell library, for higher performance. Primarily, the sizes of driver transistors in frequently-used cells were increased such that area of the cells does not increase by more than 20%. Furthermore, these cells were characterized for different loads and input slews, and their layout was updated. The optimization took one person 32 hours of design effort. Note that this optimization is different from the traditional approach of specializing transistors for specific datapaths. The optimized cells are still generic and their use in the automated flow is left to CAD tools.

We implemented Core-2W and Core-4W using the optimized 45nm cell library, with none of the previous memory optimizations or microarchitecture adjustments. Table 5.8 reports the results. Core-2W achieves 20% higher frequency and Core-4W achieves 7% higher frequency compared to the baseline implementation. The performance gain for
Table 5.8: Results for Core-2W and Core-4W designs using optimized 45nm cell library.

<table>
<thead>
<tr>
<th></th>
<th>Frequency (MHz)</th>
<th>Power (mW/MHz)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core-2W</td>
<td>1000</td>
<td>0.474</td>
<td>1.18</td>
</tr>
<tr>
<td>Core-4W</td>
<td>714</td>
<td>0.774</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Core-4W is moderate because of the increased area of the design resulting in longer wires to be routed among cells. The increased cell area, because of circuit optimization, offsets the benefit of increased performance of individual cells. With additional design effort, the 8 cells can be further optimized for achieving better performance in more complex designs as well.

5.3 Putting it all together

The previous section applied the memory structure optimizations and microarchitecture adjustments to individual pipeline stages and quantified design quality and design effort at that granularity. In this section, we set out to achieve a certain frequency (or other design quality metric) with minimum design effort, for the core as a whole. For example, suppose we want to target 1 GHz. For each pipeline stage, we search for a minimum-effort combination of optimizations/adjustments that meet 1 GHz. By varying the target, we can obtain an optimal frontier of core designs that maximize return on investment.

The solid-line with square-labels in Figure 5.11 shows the minimum-effort frequency frontier. Each square corresponds to a core design that achieves the frequency on the x-axis for minimum design effort on the y-axis. Design effort is normalized to the baseline design effort (just SPR). The power consumption for each core design is shown with the line connected by triangle-labels and relates to the alternate y-axis on the right. The labels are decoded in Table 5.9.

Because Fetch-1 is so critical and because the MFD optimization performs so well, it is no surprise that applying it alone achieves a significant frequency boost with little design effort. It also must remain in all subsequent core designs.
Figure 5.11: Optimal frontier of Core-2W designs that minimize design effort for a given frequency.
Table 5.9: Key for decoding the labels in Figure 5.11.

<table>
<thead>
<tr>
<th>Label</th>
<th>Memory Optimizations and Microarchitecture Adjustments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Baseline</td>
</tr>
<tr>
<td>B</td>
<td>STD Cell</td>
</tr>
<tr>
<td>C</td>
<td>Fetch-1-4W-MFD</td>
</tr>
<tr>
<td>D</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1</td>
</tr>
<tr>
<td>E</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF</td>
</tr>
<tr>
<td>F</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-FF</td>
</tr>
<tr>
<td>G</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-IQP-SRAM1R1W</td>
</tr>
<tr>
<td>H</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1</td>
</tr>
<tr>
<td>I</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1, Rename-Latch, RegRead-Latch</td>
</tr>
<tr>
<td>J</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1, Rename-FF-Pipe1, RegRead-Latch</td>
</tr>
<tr>
<td>K</td>
<td>Fetch-1-4W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1, Rename-Latch, RegRead-Pipe1-FF</td>
</tr>
</tbody>
</table>
Figure 5.12: Optimal frontier of Core-4W designs that minimize design effort for a given frequency.

Figure 5.12 and Table 5.10 show the results for Core-4W. After point C, the Register Read stage is critical and requires the 2R2W SRAM optimization as was prominently discussed in Section 5.2.2. The LSU is also critical and requires pipelining.
<table>
<thead>
<tr>
<th>Label</th>
<th>Memory Optimizations and Microarchitecture Adjustments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Baseline</td>
</tr>
<tr>
<td>B</td>
<td>STD Cell</td>
</tr>
<tr>
<td>C</td>
<td>Fetch-1-4W-MFD</td>
</tr>
<tr>
<td>D</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1</td>
</tr>
<tr>
<td>E</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1, IssuePipe1-FF</td>
</tr>
<tr>
<td>F</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1, IssuePipe1-IQP1-FF</td>
</tr>
<tr>
<td>G</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1, IssuePipe1-IQP1-FF, Dispatch-Pipe1, Decode-Pipe1</td>
</tr>
<tr>
<td>H</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1, IssuePipe1-IQP1-FF, Dispatch-Pipe1, Decode-Pipe1, Rename-Latch</td>
</tr>
<tr>
<td>I</td>
<td>Fetch-1-4W-MFD, RegRead-SRAM2R2W, LSU-Pipe1, IssuePipe1-IQP1-FF, Dispatch-Pipe1, Decode-Pipe1, Rename-FF-Pipe2</td>
</tr>
<tr>
<td>J</td>
<td>Fetch-1-4W-MFD-SRAM1R1W, RegRead-SRAM2R2W, LSU-Pipe1, Issue-Pipe1-IQP1-FF, Dispatch-Pipe1, Decode-Pipe1, Rename-Latch</td>
</tr>
<tr>
<td>K</td>
<td>Fetch-1-4W-MFD-Pipe1, RegRead-SRAM2R2W, LSU-Pipe1, Issue-Pipe1-IQP1-FF, Dispatch-Pipe1, Decode-Pipe1, Rename-FF-Pipe2</td>
</tr>
<tr>
<td>L</td>
<td>Fetch-1-4W-MFD-SRAM1R1W, RegRead-Pipe2-FF, LSU-Pipe2, Issue-Pipe1-IQP2-FF, Dispatch-Pipe1, Decode-Pipe1, Rename-Latch</td>
</tr>
</tbody>
</table>
Chapter 6

Design-Effort Alloy: An Approach for Optimizing Multi-core Efficiency

We presented a detailed evaluation of the performance and efficiency advantages of heterogeneous multi-core in Chapter 4. The results in Chapter 4 are based on 18 diverse superscalar cores implemented using a similar physical design flow. In particular, we used an automated SPR design flow, using the Nangate 45nm standard cell library, and the FabMem tool, to measure cores’ frequency and power. The FabMem tool generates SPICE netlists of RAMs/CAMs, however, the circuit elements used to create the full SPICE netlist are generic. Few of the circuit elements, for example, bitcell, row decoder, sense amplifier, etc., are optimized for a specific datapath or a specific memory array size. The SPR flow generates a good-quality physical design with very low effort, however, it fails to meet the performance and efficiency achieved by a high-performance processor. A commercially available server-class processor can operate at 2.5GHz to 3.5GHz [20]. Designers of such processors spend significant effort to tune the microarchitecture, RTL model, circuits, and physical layout to optimize performance and energy usage [86] [35]. Moreover, the effort is justified because the non-recurring engineering (NRE) cost of the processor development is amortized over a large market volume. Figure 6.1 compares the average BIPS$^3$/Watt of the 4W-S core implemented using low-effort and high-effort flows, evaluated across benchmarks in Table 4.1. Modeling of the high-effort flow will be explained in Section 6.2.

Despite the overall increase in efficiency by investing high-effort, the one-size-fits-
Figure 6.1: Efficiency comparison of a core implemented using low-effort and high-effort design flows. Efficiency (BIPS$^3$/Watt) is normalized to the efficiency of the low-effort implementation. 4W-S is the best core on average for BIPS$^3$/Watt in our design space. Frequency and energy modelling of the high-effort core is explained in Section 6.2.

all microarchitecture still suffers from suboptimal efficiency on individual applications. Alternatively, multiple diverse cores can be implemented using the high-effort flow to provide even better efficiency. However, the NRE cost of such a design would be insurmountable. 

This thesis proposes a new class of heterogeneous multi-core processor that is composed of high-effort and low-effort cores. A core designed using the high-effort implementation flow is referred to as high-effort core and a core designed using the low-effort implementation flow is referred to as low-effort core. The insight behind our proposal follows from experiments performed in Chapter 4 and Chapter 5:

- The N-core-type heterogeneous multi-core processor always consists of the best “average” core, i.e., the best single core type for a homogeneous multi-core. Many benchmarks in the application space prefer executing on the average core. The other N-1 core types remove key microarchitecture bottlenecks, for example, ROB or Issue Queue size, fetch width, etc., in the average core to capture diversity in the application space’s instruction-level behaviours. The average core can be the high-effort core and other core types can be low-effort cores. Low-effort cores primarily trade-off frequency for IPC.
Decades of investment in CAD and the intrinsic speed afforded by advanced technology are reducing the gap between automated and manual design. The SPR flow can be used to achieve a clock frequency range of 1.0GHz to 2.0GHz depending on the microarchitecture complexity. With further advancement in technology nodes and CAD algorithms, the quality of layouts generated by the SPR flow will improve further.

We call our solution *Design-Effort Alloy* (DEA), in reference to using two distinct implementation flows to provide a lower NRE cost heterogeneous multi-core. Moreover, we demonstrate that the design-effort alloy based heterogeneous multi-core achieves higher efficiency than the high-effort based homogeneous multi-core. Finally, our proposal is different from combining one complex superscalar core with replicated simple superscalar cores. The simple superscalar core doesn’t fundamentally remove any performance or efficiency bottleneck in the complex core. It simply provides a lower-energy operating point.

The rest of the chapter is organized as follows. Section 6.1 describes the design techniques for the low-effort core. The SPR flow is one way to generate a low-effort core. Section 6.2 describes our approach to model the clock frequency and energy consumption of a high-effort core. Finally, Section 6.3 presents our result.

### 6.1 Designing a Low-Effort Core

The design techniques for creating a low-effort core can potentially be many. Figure 6.2 shows a few example techniques, including using SPR flow (“Automated Synthesis and Physical Design” box). Other techniques that could further simplify designing low-effort cores are implementing a subset of the ISA and not fully validating the core.

For complex ISAs, like x86, the low-effort core might implement only a subset of the full ISA. The subset should reflect commonly executed instructions in the ISA. Such a design will create a form of functional asymmetry, where only the high-effort core provides the full functionality of the ISA and low-effort cores are optimized for a subset of the ISA. Li et al. [59] studied the x86-based heterogeneous multi-core with functional asymmetry. The authors extended the existing operating system with a fault-and-migrate mechanism to ensure correct execution of applications. In our context, if the low-effort
core encounters an unsupported instruction while executing an application, the hardware generates a fault-type exception. The exception handler migrates the thread to the high-effort core which can correctly execute the faulting instruction. Implementing only a subset of the ISA in the low-effort core reduces both physical design complexity and functional verification complexity.

Sudhakrishnan et al. [89] proposed the Beta Core Solution (BCS), which includes a complex core that is not fully verified functionally, a checker core, and logic to detect functional bugs. The checker core can simply be an in-order core that is easy to verify. In the Retire stage of the complex core, the commit information is verified against the output generated by the checker core. In case of a functional bug detected in the complex core, the correct architectural state from the checker core is copied to the complex core for further execution. Our low-effort superscalar cores can employ BCS to further reduce the NRE cost of employing diverse core types.

In this chapter, our primary focus is using an automated SPR flow for generating the low-effort core types. However, additionally implementing an ISA subset and BCS has the potential to reduce the effort further.
6.2 Designing a High-Effort Core

Designing a high-performance general-purpose processor design is a quantitative art. Engineers spend a significant amount of time to balance the microarchitectural resources, in terms of no one structure being the sole limiter of instructions-per-cycle (IPC), balance delay across pipeline stages, fine-tune the circuits, and compact the layout to get the best of both performance and efficiency. The engineering effort to optimize a core is very design-team specific. Nonetheless, we classify three majors areas, where engineers spend significant effort, based on a literature review of the IBM POWER7 [35], IBM POWER6 [86], and Intel Pentium 4 [28] design methodologies:

1. Logic partitioning: The canonical pipeline stage boundaries are blurred to balance the delay of pipeline stages. Logic is partitioned across the pipeline stages to minimize the timing slack in each stage, hence, maximize the instruction throughput. The partitioning incurs additional design and verification complexity.

2. Custom design for IPC-critical stages: The IPC-critical stages, for example, Issue and Fetch, are carefully implemented using quality circuit and layout design. Moreover, the IPC-critical stages are more likely energy-critical stages as well. Hence, the pressure is even more on engineers to balance both performance and energy.

3. Clock distribution: An efficient clock tree distribution is extremely crucial for high-performance processors. An unbalanced clock tree can lead to higher clock-skew which negatively impacts the clock frequency. Moreover, the clock tree typically accounts for 20-30% of dynamic power consumption. Engineers spend considerable effort to size clock-wires and clock-buffers to minimize skew and power consumption.

6.2.1 Modeling Frequency and Energy of High-Effort Core

To demonstrate the advantage of Design-Effort Alloy based heterogeneous multi-core, we need to define a high-effort core. Since we are interested in the efficiency metric (BIPS$^3$/Watt metric), the microarchitecture configuration of 4W-S is chosen as the configuration for the high-effort core. 4W-S is the best single core, on average, for BIPS$^3$/Watt, and it appears in all the N-core-type heterogeneous multi-core configurations optimized
for BIPS³/Watt.

Unfortunately, we cannot simply use the clock frequency and energy consumption generated by the FabScalar-PPA framework for the 4W-S core. The FabScalar-PPA framework is primarily based on automated logic synthesis and the FabMem memory compiler, and it does not reflect the frequency and energy of cores designed using a high-effort implementation flow. For example, 4W-S can only operate at a 1.67GHz frequency in our design space. However, a commercially available server- and desktop-class core can operate at 2.5GHz to 3.5GHz. It is not very difficult to configure our cycle-accurate C++ simulator to operate at a higher frequency. The challenging task is to model the energy consumption of the 4W-S core operating at this higher frequency. In other words, what is the increase in energy when circuits are optimized for achieving higher frequency? We take two parallel approaches to understand the relationship between energy and frequency. Firstly, we hand-optimize the timing-critical paths of the 4W-S core so that the core can operate at a higher frequency. The SPICE netlists of the timing-critical paths are generated, and transistor sizes are optimized to speed-up the critical path. In particular, timing-critical paths in the Issue and Register Read stages are optimized (as these are most timing-critical in 4W-S). With increasing transistor width, the energy consumption increased as well. Figure 6.3 (“SPICE Opt.” line) presents the factor by which energy increases with increasing target frequency. A baseline frequency of 1.5GHz is used to calculate the factors. However, the naive hand-optimization approach saturated around 2.3GHz and hardly any increase in frequency was achieved thereafter. Certainly, more sophisticated circuit optimizations can be applied to speed-up the design but it will require significant person-hours. In the second approach, we use the McPAT tool for modelling frequency and energy of the high-effort core [58]. As a first step, we adjusted McPAT technology parameters for FreePDK. This adjustment is similar to modifications we made to CACTI device and wire parameters (refer to Section 3.3 for more details). Using McPAT, the increase in core energy is measured for different target frequencies. Figure 6.3 shows the trend of energy increase with increasing frequency for the 4W-S and 2W-S cores. McPAT, adjusted to FreePDK technology parameters, fails to achieve a 3.0GHz frequency for 4W-S. This reflects the pessimism in the FreePDK technology that Nangate libraries are based on. However, we extrapolate the trend for the 4W-S core for simulation purposes. The linear extrapolation is based on the observation of the
Figure 6.3: Energy-increase trend of the high-effort core with increasing clock frequency. The increase in energy is normalized to 1.5GHz core.

The linear trend shown by the 2W-S core. Interestingly, the energy increase trend shown by hand-optimization is very close to the trend shown by McPAT for the 4W-S core. Note that frequency and energy are linearly related to transistor width.

For all our experiments in the following section, we use the trend extracted from McPAT to scale the energy of the 4W-S core operating at higher frequency. Moreover, we refer to the high-effort 4W-S core as HE-4W-S (HE emphasizes that it is a high-effort core).

6.3 Results

Figure 6.4 shows two multi-core designs used for this study. Figure 6.4 (a) is the conventional homogeneous multi-core composed of only high-effort (HE) cores. This is referred to as Design-A. Figure 6.4 (b) is the proposed DEA based heterogeneous multi-core composed of HE and low-effort (LE) cores. This is referred to as Design-B. The microarchitecture configuration of the HE core is the same as the 4W-S core, however, it differs in the operating frequency. Two frequency points are used for the HE core: 2.5GHz and 3.0GHz. The HE core operating at 2.5GHz is referred as HE-4W-S_2500 and it is referred to as HE-4W-S_3000 when operating at 3.0GHz. The higher frequency of the 4W-S core
Table 6.1: Low-effort core types used for evaluating Design-Effort Alloy based heterogeneous multi-core.

<table>
<thead>
<tr>
<th>Core Name</th>
<th>Memory Optimizations and Microarchitecture Adjustments</th>
<th>Frequency (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE-1W-S</td>
<td>Fetch-1-2W-MFD, LSU-Pipe1, Decode-Pipe1, Issue-Pipe1-FF, Dispatch-Pipe1, Rename-Latch, RegRead-Pipe1-FF</td>
<td>1.8</td>
</tr>
<tr>
<td>LE-2W-L</td>
<td>Fetch-1-4W-MFD, LSU-Pipe2, RegRead-Pipe4-FF, Issue-Pipe1-FF, Decode-Pipe1, Dispatch-Pipe1, Rename-Latch</td>
<td>1.25</td>
</tr>
</tbody>
</table>

reflects the huge effort that goes into microarchitecture tuning, circuit tuning, and crafting a high quality physical design. The baseline energy of the 4W-S core, when operating at 1.5GHz, comes from the FabScalar-PPA. Further, the energy of the 4W-S core, when operating at 2.5GHz and 3.0GHz, is scaled based on the graph in Figure 6.3. Note that the HE cores in both Design-A and Design-B are the same. Moreover, when we compare Design-A and Design-B, their HE cores’ clock frequencies are the same.

Two LE core types, LE-1W-S and LE-2W-L, are used for Design-B. LE is prefixed to the core names to highlight that they are implemented using automated SPR flow. Table 6.1 shows the sequence of memory optimizations and microarchitecture adjustments performed to the low-effort cores and their final operating frequencies. Refer to Chapter 5 for details on memory optimizations and microarchitecture adjustments. Although we are using only two LE core types for Design-B, more cores can be added based on the target application space.

Figure 6.5 shows overall efficiency gain achieved by Design-B over Design-A. Design-B is 6% and 5.2% better than Design-A when the HE core is operating at 2.5GHz and 3.0GHz, respectively. The low overall gain achieved by Design-B is not surprising because the majority of the benchmarks prefer executing on the best average core. Moreover, higher frequency leads to more performance for many benchmarks, for less than linear increase in energy consumption. Note that, doubling the clock frequency of 4W-S increases energy by a factor of less than $1.6 \times$. However, there are benchmarks that gain significant benefits from executing on LE cores. Figure 6.6 and Figure 6.7 show ef-
Figure 6.4: Two multi-core designs used for studying the advantage of design-effort alloy. 
(a) The conventional homogeneous multi-core composed of all HE cores. (b) The proposed DEA based heterogeneous multi-core composed of HE and LE cores.
Figure 6.5: Overall efficiency (BIPS$^3$/Watt) gain achieved by Design-B over Design-A when the HE core is operating at 2.5GHz and 3.0GHz.

Efficiency gains achieved by 25 benchmarks (out of 76 benchmarks) from executing on LE cores. Some benchmarks, e.g., swim.1226, ammp.3783, and art.591, achieve more than $3 \times$ efficiency advantage. Although, the HE core is a highly-tuned machine, its generic microarchitecture becomes a bottleneck for these benchmarks. The LE core-types target the HE core-type’s inherent compromises on outlier ILP characteristics, accelerating these benchmarks to further boost single-thread performance and efficiency where possible. This is evident from the result that LE cores actually provide value above and beyond the HE core, despite sacrificing as much as half of the frequency possible with microarchitecture, circuits, and physical design tuning. Finally, Table 6.2 shows the LE core type preferred by each of the 25 benchmarks.
Figure 6.6: Efficiency (BIPS$^3$/Watt) gains achieved by 25 benchmarks from executing on LE cores. The HE core is operating at 2.5GHz.

Figure 6.7: Efficiency (BIPS$^3$/Watt) gains achieved by 25 benchmarks from executing on LE cores. The HE core is operating at 3.0GHz.
Table 6.2: The preferred LE core type in Design-B yielding efficiency gains compared to executing on the HE core type.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>LE Core</th>
<th>Benchmark Name</th>
<th>LE Core</th>
<th>Benchmark Name</th>
<th>LE Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim.1226</td>
<td>2W-L</td>
<td>art.10257</td>
<td>2W-L</td>
<td>art.3441</td>
<td>2W-L</td>
</tr>
<tr>
<td>swim.1583</td>
<td>2W-L</td>
<td>art.5922</td>
<td>2W-L</td>
<td>mcf.3091</td>
<td>1W-S</td>
</tr>
<tr>
<td>swim.1582</td>
<td>2W-L</td>
<td>equake.869</td>
<td>2W-L</td>
<td>mesa.2297</td>
<td>2W-L</td>
</tr>
<tr>
<td>swim.1312</td>
<td>2W-L</td>
<td>equake.2796</td>
<td>2W-L</td>
<td>mgrid.3283</td>
<td>2W-L</td>
</tr>
<tr>
<td>ammp.3783</td>
<td>2W-L</td>
<td>equake.1046</td>
<td>2W-L</td>
<td>mgrid.2977</td>
<td>2W-L</td>
</tr>
<tr>
<td>ammp.3018</td>
<td>2W-L</td>
<td>gap.3229</td>
<td>2W-L</td>
<td>mgrid.3657</td>
<td>2W-L</td>
</tr>
<tr>
<td>ammp.2945</td>
<td>2W-L</td>
<td>equake.1093</td>
<td>2W-L</td>
<td>ammp.2891</td>
<td>2W-L</td>
</tr>
<tr>
<td>art.591</td>
<td>2W-L</td>
<td>mgrid.3490</td>
<td>2W-L</td>
<td>mcf.2018</td>
<td>1W-S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>parser.10758</td>
<td>2W-L</td>
</tr>
</tbody>
</table>
Chapter 7

Summary

This thesis proposed FabScalar, a novel approach for automatically composing the synthesizable register-transfer-level (RTL) designs of arbitrary cores within a canonical superscalar template. Each canonical pipeline stage has many variants that differ in their complexity (superscalar width and stage-specific structure sizes) and depth of sub-pipelining, and canonical pipeline stages are composable into an overall core. The thesis provided detailed validation experiments along three fronts to evaluate the quality of RTL designs generated by FabScalar: functional and performance (IPC) validation, timing validation (cycle time), and confirmation of suitability for standard ASIC flows. These experiments confirmed that FabScalar-generated RTL designs are of good quality.

FabScalar addresses the design-effort problem posed by single-ISA heterogeneous multi-core processors and time-to-market sensitive APs. Moreover, it helps mitigate practical issues that currently impede proliferating microarchitecturally diverse cores.

Further, the thesis explored automated synthesis and place-and-route (SPR) flow to achieve acceptable-quality physical designs for time-to-market sensitive superscalar cores. A range of memory structure optimizations, microarchitecture adjustments, and circuit-level optimizations, in conjunction with SPR were explored. Results show that, with different combinations of optimizations, acceptable clock frequencies can be achieved while relying primarily on automated SPR.

Finally, this thesis proposed a new class of heterogeneous multi-core processor that
is composed of high-effort and low-effort cores. The approach is referred to as *Design-Effort Alloy* (DEA). The high-effort core is implemented using a semi-custom flow (like a server-class core design) and multiple, diverse low-effort cores are implemented using a fully-automated SPR flow (as explored in Chapter 5). DEA leads to a lower NRE cost heterogeneous multi-core. Results show a DEA based heterogeneous multi-core provides significant performance and efficiency gains compared to a homogeneous multi-core.
REFERENCES


[54] David Lammer. Intel cancels Tejas, moves to dual-core designs. EETimes article.


Appendix A

Detail Design of CPSL

CPSL provides different implementations of each canonical pipeline stage defined in Fab-Scalar. Different implementations differ in superscalar width, ILP-extracting structure sizes, pipeline depth etc. It is beyond the scope of this thesis to provide detailed design of entire CPSL, however, this chapter provides detailed design of a superscalar configuration generated from FabScalar toolset. A 4-wide superscalar core, with the shallowest pipeline depth, is used to discuss logic designs and interfaces of canonical pipeline stages.

Rest of this chapter provides detailed logic design and input/output signals for each canonical pipeline stage for a 4-wide superscalar core. Our approach to describe logic design is as follows. First, we briefly describe the functionality of the canonical stage. Further, we describe input/outputs signals using flags and packets. Flags are single-bit signals, and primarily represent control signals. Packets, on the other hand, are multi-bit signals. Finally, we provide a detailed logic diagram of the canonical stage’s RTL module including all the major sub-modules and the logic connecting them.

Note: In the logic diagram, a grey coloured box represents a sub-module and a green coloured box represents edge-triggered flip-flops (registers).

A.1 Design of Fetch

The Fetch stage is responsible for providing continuous instruction stream to rest of the pipeline. In FabScalar, the Fetch stage is always sub-pipelined in two stages: 1) Fetch-1
(described in Section A.1.1) and 2) Fetch-2 (described in Section A.1.2).

A.1.1 Fetch-1

The Fetch-1 stage is responsible for accessing instruction cache and in parallel predicting the program counter (PC) for the next cycle. Table A.1 describes the input/output signals to the Fetch-1 stage. Figure A.1 provides detailed logic diagram of the Fetch-1 stage. The stage contains four major sub-modules and the next-PC logic. The four sub-modules are Level-1 Instruction Cache (L1 ICache), Branch Target Buffer (BTB), Branch Predictor, and Return Address Stack (RAS).

L1 ICache contains cache array, and it provides instruction bundle, pertaining to the current PC, to the Fetch-2 stage. The cache array is 2-way interleaved to obviate the need for a dual-port SRAM, guaranteeing two consecutive instruction cache blocks in a cycle, from which four sequential instructions can be extracted from any unaligned starting PC. BTB, Branch Predictor and RAS provide control instruction specific information, e.g., control type, target address, and predicted directions of the conditional branches, to the next-PC logic. BTB module, shown in Figure A.2, contains 4-way interleaved SRAM and an entry in SRAM contains PC of the control instruction, its type, and the associated target address. Branch Predictor module, shown in Figure A.4, contains Branch Prediction Buffer (BPB) and predicts the direction of the conditional branch (taken or not-taken). RAS module, shown in Figure A.3, provides return address if the current fetch bundle contains a return instruction. In case the current fetch bundle contains a Call instruction then the return address is pushed on the top of the stack in RAS module. BTB and Branch Predictor modules receive updates from control-transfer instruction queue (CTIQ) in the Fetch-2 stage.

A.1.2 Fetch-2

The Fetch-2 stage contains the instruction alignment logic and extracts up to four consecutive instructions (from among the two consecutive blocks coming from the Fetch-1 stage) based on the starting PC, or until the first predicted-taken branch, whichever comes first. Fetch-2 pre-decodes the four instructions to explicitly identify control instructions within the fetch bundle and calculate their target addresses. If the BTB misses for a control in-
Figure A.1: Fetch-1 module design.
Figure A.2: Branch Target Buffer (BTB) module design.

Figure A.3: Return Address Stack (RAS) module design.
Table A.1: Fetch-1 input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recover flag</td>
<td>Input</td>
<td>Flag is high in case of control misprediction or memory exception (e.g. load violation).</td>
</tr>
<tr>
<td>Recover PC</td>
<td>Input</td>
<td>PC used for fetching next instruction bundle after recover flag is high.</td>
</tr>
<tr>
<td>Exception flag</td>
<td>Input</td>
<td>Flag is high in case of SYSCALL instruction.</td>
</tr>
<tr>
<td>Exception PC</td>
<td>Input</td>
<td>PC used for fetching next instruction bundle after SYSCALL is serviced.</td>
</tr>
<tr>
<td>Recover from Fetch-2</td>
<td>Input</td>
<td>Fetch-2 stage pre-decodes instruction bundle for control instructions to detect BTB miss. In case of a BTB miss, Fetch-2 sends recover signals and PC used for fetching next instruction bundle.</td>
</tr>
<tr>
<td>Update from CTIQ</td>
<td>Input</td>
<td>Updates to BTB and BPB when a control instruction retires from CTIQ.</td>
</tr>
<tr>
<td>Cache-block from lower-level memory</td>
<td>Input</td>
<td>Cache-block received from lower-level memory in case of L1 ICache miss.</td>
</tr>
<tr>
<td>Instruction bundle</td>
<td>Output</td>
<td>Instruction bundled fetched from ICache.</td>
</tr>
<tr>
<td>BTB/BPB outcome</td>
<td>Output</td>
<td>BTB-hit, predicted target address, and predicted branch direction sent to Fetch-2 for detecting and fixing a BTB miss.</td>
</tr>
<tr>
<td>Request to lower-level memory</td>
<td>Output</td>
<td>Cache-block request sent to lower-level memory in case of L1 ICache miss.</td>
</tr>
</tbody>
</table>
to in-order updates of the branch prediction structures.

### A.2 Design of Decode

The Decode stage contains two modules, Decode and Instruction Buffer. Decode decodes up to four instructions in a cycle to generate opcodes, operands and other relevant control information. A complex instruction, e.g., multiply and divide, are decomposed into two simpler micro-operations. Table A.3 describes the input/output signals to the Decode module. Figure A.6 provides detailed logic diagram of the Decode module. The module contains one sub-module, Decode.PISA. In Instruction Buffer, the decoded instructions are written into a circular buffer at the tail pointer. Instruction Buffer decouples instruction fetching with the rest of the front-end stages. Four instructions are read from the head pointer for renaming and dispatching. Table A.4 describes the input/output signals to Instruction Buffer module. Figure A.7 provides detailed logic diagram of Instruction Buffer module.
### Table A.2: Fetch-2 input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recover flag</td>
<td>Input</td>
<td>Flag is high in case of control misprediction or memory exception (e.g. load violation).</td>
</tr>
<tr>
<td>Instruction bundle</td>
<td>Input</td>
<td>Instruction bundled provided by Fetch-1.</td>
</tr>
<tr>
<td>Control inst. execution info</td>
<td>Input</td>
<td>Computed information of a control instruction, e.g., branch-direction, target address, sent from Writeback.</td>
</tr>
<tr>
<td>Control commit</td>
<td>Input</td>
<td>CTIQ receives commit signal from Active list on the retirement of a control instruction.</td>
</tr>
<tr>
<td>BTB/Branch Predictor outcome</td>
<td>Input</td>
<td>BTB-hit, predicted target address, and predicted branch direction received from Fetch-1 for detecting and fixing a BTB miss.</td>
</tr>
<tr>
<td>Update to BTB/BPB</td>
<td>Output</td>
<td>Updates sent to BTB and BPB when a control instruction retires from CTIQ.</td>
</tr>
<tr>
<td>Instruction packet</td>
<td>Output</td>
<td>Individual instructions are extracted from the bundle and sent to decode. Each packet carries a valid signal.</td>
</tr>
</tbody>
</table>

### Table A.3: Decode input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction packet</td>
<td>Input</td>
<td>Un-decoded instructions received from Fetch-2.</td>
</tr>
<tr>
<td>Decoded instruction packet</td>
<td>Output</td>
<td>The packet contains all the decoded information, e.g., opcode, operand, immediate value, pertaining to an instruction.</td>
</tr>
</tbody>
</table>
Figure A.5: Fetch-2 module design.

Table A.4: Instruction-buffer input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoded instruction packet</td>
<td>Input</td>
<td>The packet sent from Decode and written to Instruction buffer.</td>
</tr>
<tr>
<td>Stall fetch</td>
<td>Output</td>
<td>The flag is high when Instruction buffer is full. Instruction fetching should stall when the flag is high.</td>
</tr>
<tr>
<td>Instruction buffer ready</td>
<td>Output</td>
<td>The flag is high when instruction buffer has equal to or more than dispatch width equivalent instructions.</td>
</tr>
<tr>
<td>Decoded instruction packet</td>
<td>Output</td>
<td>The packet read from Instruction buffer for renaming.</td>
</tr>
</tbody>
</table>
Figure A.6: Decode module design.
Figure A.7: Instruction Buffer module design.
A.3 Design of Rename

The Rename stage remaps the architectural source and destination registers to physical source and destination registers. Register renaming removes the false dependencies among instructions which are artefacts of limited architectural registers. Table A.5 describes the input/output signals to the Rename stage. Figure A.8 provides detailed logic diagram of the Rename stage. The stage contains two sub-modules, SpecFreeList and Rename Map Table.

SpecFreeList implements a circular buffer that contains the unused physical registers, and a physical destination register is obtained for an instruction with an architectural destination register by popping a free physical register from the circular buffer. Rename Map Table module, shown in Figure A.9, maintains the physical registers to which architectural registers are currently mapped. Accordingly, each architectural source register of the instruction is renamed to a physical source register by looking up its mapping in the RMT array. After renaming an instruction’s source registers, its new architectural-to-physical destination register mapping is updated in the RMT array for future instructions to observe. Simultaneously, true dependencies between source registers and preceding destination registers are checked for the group of instructions being renamed concurrently.

A.4 Design of Dispatch

The Dispatch stage is responsible for checking the available space in the back-end pipeline stages, in particular, the Retire, Issue, and Load-Store Unit, for newly renamed instructions. If the space is available, the Dispatch stage writes the new instructions in the respective back-end resources. In case of the unavailability of enough space in these resources, the dispatch stage generates a stall signal for the rename stage. Table A.6 describes the input/output signals to the Dispatch stage. Figure A.10 provides detailed logic diagram of the Dispatch stage. The stage contains two sub-modules, ExePipeSchedule and LoadViolationPred.

ExePipeSchedule module assigns an instruction to a particular execution lane based on its type and a round-robin scheduling policy. Pre-assigning execution lanes simplifies the selection process in the Issue stage. LoadViolationPred module predicts if an incoming
<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode ready</td>
<td>Input</td>
<td>The flag is high when instruction buffer has equal to or more than dispatch width instructions.</td>
</tr>
<tr>
<td>Decoded instruction packet</td>
<td>Input</td>
<td>The packet read from Instruction buffer for renaming.</td>
</tr>
<tr>
<td>Commit packet</td>
<td>Input</td>
<td>The packet contains freed physical register that gets added to Free List. The packet is sent from AMT.</td>
</tr>
<tr>
<td>Recover packet</td>
<td>Input</td>
<td>Architectural logical-to-physical register mapping sent from AMT in case of recovery from mis-speculation or exception.</td>
</tr>
<tr>
<td>Free list empty</td>
<td>Output</td>
<td>The flag is high when sufficient physical registers are not available for logical-to-physical register mapping. Reading from instruction buffer should stall.</td>
</tr>
<tr>
<td>Rename ready</td>
<td>Output</td>
<td>The flag is high when Rename is ready with dispatch-width equivalent renamed instructions.</td>
</tr>
<tr>
<td>Renamed packet</td>
<td>Output</td>
<td>The instruction packet after logical-to-physical register mapping.</td>
</tr>
</tbody>
</table>
Figure A.8: Rename module design.
load instruction would potentially violate the memory-order dependency. The module receives update from the Retire stage.

### A.5 Design of Issue

The Issue stage buffers the renamed instructions and selects instructions for execution based on the availability of their source operands. Table A.7 describes the input/output signals to the Issue stage. Figure A.11 and A.12 provide detailed logic diagram of the Issue stage. The stage contains multiple sub-modules. Major sub-modules are Issue Queue Free List, Payload RAM, Result Shift Register (RSR), Src-0 CAM, Src-1 CAM, and Select.

Issue Queue Free List module implements a circular buffer that contains the unused entries in Issue stage. A dispatched instruction is assigned an entry in Payload RAM and Src-0/1 CAM by popping a free entry from the circular buffer. Payload RAM and Src-0/1 CAM modules hold instruction payload and source register specifiers, respectively. Each
<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renamed packet</td>
<td>Input</td>
<td>The packet sent from Rename.</td>
</tr>
<tr>
<td>Load violation flag</td>
<td>Input</td>
<td>The flag is high in case of a load violation detected in Retire.</td>
</tr>
<tr>
<td>Load violation PC</td>
<td>Input</td>
<td>The PC of violating load sent from Retire. The PC is used for updating load-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>violation predictor table.</td>
</tr>
<tr>
<td>Load queue count</td>
<td>Input</td>
<td>Count of occupied Load queue entries.</td>
</tr>
<tr>
<td>Store queue count</td>
<td>Input</td>
<td>Count of occupied Store queue entries.</td>
</tr>
<tr>
<td>Issue queue count</td>
<td>Input</td>
<td>Count of occupied Issue queue entries.</td>
</tr>
<tr>
<td>Active list count</td>
<td>Input</td>
<td>Count of occupied Active list entries.</td>
</tr>
<tr>
<td>Stall front-end</td>
<td>Output</td>
<td>The flag is high in case one or more of back-end’s instruction buffering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>structures have insufficient space. Reading from instruction buffer should</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stall.</td>
</tr>
<tr>
<td>Back-end ready</td>
<td>Output</td>
<td>The flag is high when back-end’s buffering structures have sufficient space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for newly dispatched instructions.</td>
</tr>
<tr>
<td>Issue queue packet</td>
<td>Output</td>
<td>The Packet is sent to Issue. It contains opcode, physical register names</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(src. and dest.), immediate values etc.</td>
</tr>
<tr>
<td>Active list packet</td>
<td>Output</td>
<td>The Packet is sent to Retire. It contains PC and logical and destination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>physical mapping.</td>
</tr>
<tr>
<td>LSQ packet</td>
<td>Output</td>
<td>The Packet is sent to LSU. It only contains load/store control flags.</td>
</tr>
</tbody>
</table>
Extract Memory Instructions
Rename packet-0
Rename packet-1
Rename packet-2
Rename packet-3

Figure A.10: Dispatch module design.
Table A.7: Issue input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-end ready</td>
<td>Input</td>
<td>The flag is high when back-end’s buffering structures have sufficient space for newly dispatched instructions.</td>
</tr>
<tr>
<td>Dispatched packet</td>
<td>Input</td>
<td>The packet is sent from Dispatch. It contains opcode, physical register names (src. and dest.), immediate values etc.</td>
</tr>
<tr>
<td>Active list ID</td>
<td>Input</td>
<td>Active list entry number for newly dispatched instruction.</td>
</tr>
<tr>
<td>LSQ ID</td>
<td>Input</td>
<td>Load/Store queue entry number for newly dispatched instruction.</td>
</tr>
<tr>
<td>Load tag</td>
<td>Input</td>
<td>Physical destination tag of the load instruction sent from Writeback. It is used for waking-up load’s dependent instructions.</td>
</tr>
<tr>
<td>Physical register valid vector</td>
<td>Input</td>
<td>Physical register file valid bit-vector. It is used for checking source operands’ readiness of dispatched instructions.</td>
</tr>
<tr>
<td>Issue queue count</td>
<td>Output</td>
<td>Count of occupied Issue queue entries.</td>
</tr>
<tr>
<td>RSR tag</td>
<td>Output</td>
<td>Destination tag of an issued instruction is sent to Register Read for updating physical register file valid vector.</td>
</tr>
<tr>
<td>Granted packet</td>
<td>Output</td>
<td>The packet contains selected instruction and its payload for execution.</td>
</tr>
</tbody>
</table>

cycle, RSR module broadcasts destination register specifiers of previously issued instructions to wake-up their dependents in timely fashion. Ready instructions and newly woken instructions participate in the selection process. Select module selects an instruction per execution lane.

A.6 Design of Register Read

The Register Read stage contains the physical register file (PRF), which holds all the committed and non-committed instruction results. The source register specifiers of an issued instruction index into the PRF to read the corresponding values. At the same
Figure A.11: Issue module (part-1) design.
Figure A.12: Issue module (part-2) design.
Table A.8: Register Read input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granted packet</td>
<td>Input</td>
<td>The packet contains selected instruction and its payload for execution.</td>
</tr>
<tr>
<td>RSR tag</td>
<td>Input</td>
<td>Destination tag of issued instruction is sent to Register Read for updating physical register file valid vector.</td>
</tr>
<tr>
<td>Bypass packet</td>
<td>Input</td>
<td>The packet contains physical destination tag and data value sent from Writeback.</td>
</tr>
<tr>
<td>Un-map tag</td>
<td>Input</td>
<td>The physical register being freed by ArchMapTable.</td>
</tr>
<tr>
<td>FU packet</td>
<td>Output</td>
<td>The packet is similar to the granted packet. It also contains source operand values.</td>
</tr>
</tbody>
</table>

time, source register specifiers are also compared with the Writeback destination register specifiers to detect the scenario whereby a producer instruction’s result needs to be directly bypassed to a consumer instruction. Table A.8 describes the input/output signals to the Register Read stage. Figure A.13 provides detailed logic diagram of the Register Read stage.

A.7 Design of Execute

The Execute stage performs an arithmetic or logic operation on the source operands of an instruction, and the result of the operation is written into the Writeback latches. At the input of Execute stage, source register specifiers are compared with the Writeback destination register specifiers to detect the scenario whereby a producer instruction’s result needs to be directly bypassed to a consumer instruction. Table A.9 describes the input/output signals to the Execute stage. Figure A.14 provides detailed logic diagram of the Execute stage. The stage contains four sub-modules, Simple ALU, Complex ALU, Control ALU, and AGEN.
Figure A.13: Register Read module design.

Table A.9: Execute input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU packet</td>
<td>Input</td>
<td>The packet received from Register Read.</td>
</tr>
<tr>
<td>Bypass packet</td>
<td>Input</td>
<td>The packet contains physical destination tag and data value sent from Writeback.</td>
</tr>
<tr>
<td>Executed packet</td>
<td>Output</td>
<td>The packet contains information, e.g., destination operand, branch outcome, execution-flags, of an instruction after its execution.</td>
</tr>
</tbody>
</table>
Figure A.14: Execute module design.
A.8 Design of Load-Store Unit

The Load-Store Unit implements a separate load queue (LQ) and store queue (SQ) to maintain the uncommitted memory operations in their program order. It primarily contains store-to-load forwarding logic, load violation detection logic, and level-1 data cache. Table A.10 describes the input/output signals to the Load-Store Unit stage. Figure A.15 and A.16 provide detailed logic diagram of the Load-Store Unit stage. The stage contains three sub-modules, Dispatched Load, Dispatched Store, and L1 Data Cache.

Dispatched Load and Dispatched Store modules find entries in LQ and SQ for incoming load and store instructions, respectively. Moreover, it establishes control information required by store-to-load forwarding logic and load violation detection logic. For example, the oldest store and its SQ entry with respect to a load. L1 Data Cache module implements data cache array. In case the AGEN packet contains a load, store-to-load forwarding logic performs associative searches to resolve address dependencies (also referred to as load disambiguation). A load might find its data from the data cache or the SQ depending on the outcome of the load disambiguation logic. The access to the data cache happens in parallel with the load disambiguation logic. If a load is predicted to violate the memory order dependency then it is stalled in LQ in case there is an unresolved store between the last matching store and the load. The stalled load is replayed when it reaches head of the LQ. In case the AGEN packet contains a store, load violation detection logic performs associative searches to resolve if any load violated memory order dependency.

A.9 Design of Writeback

The Writeback stage contains the latches holding the results from the Execute stage, which serve as the source for feeding the bypass network and sending completion information to Retire stage. The bypass network forwards the result values from the executed instructions to the dependent instructions. Table A.11 describes the input/output signals to the Writeback stage. Figure A.17 provides detailed logic diagram of the Writeback stage.
Figure A.15: Load Store Unit module (part-1) design.
Figure A.16: Load Store Unit module (part-2) design.
Figure A.17: Writeback module design.
Table A.10: Load-Store input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSQ packet</td>
<td>Input</td>
<td>The Packet received from Dispatch. It only contains load/store control flags.</td>
</tr>
<tr>
<td>Commit load</td>
<td>Input</td>
<td>Commit signal sent to Load queue if a load instruction is retiring from Active list.</td>
</tr>
<tr>
<td>Commit store</td>
<td>Input</td>
<td>Commit signal sent to Store queue if a store instruction is retiring from Active list. The store is now non-speculative and write to the data cache.</td>
</tr>
<tr>
<td>AGEN packet</td>
<td>Input</td>
<td>The packet contains memory address.</td>
</tr>
<tr>
<td>Load queue count</td>
<td>Output</td>
<td>Count of occupied Load queue entries.</td>
</tr>
<tr>
<td>Store queue count</td>
<td>Output</td>
<td>Count of occupied Store queue entries.</td>
</tr>
<tr>
<td>LSU packet</td>
<td>Output</td>
<td>The packet contains load/store execution-flags, and the load data received from either store queue or data cache.</td>
</tr>
<tr>
<td>Load violation packet</td>
<td>Output</td>
<td>The packet has Active list ID of the load instruction that executed pre-maturely.</td>
</tr>
</tbody>
</table>

A.10 Design of Retire

The Retire stage updates the architectural processor state in the correct program order to maintain the sequential execution model. The in-order commit of instructions naturally leads to the implementation of precise interrupts. The Retire stage contains two modules, ActiveList and ArchMapTable. Table A.12 describes the input/output signals to ActiveList module. Figure A.18 and A.19 provide detailed logic diagram of ActiveList module. ActiveList module maintains the program order among instructions using a circular FIFO with head and tail pointers. The dispatched instructions are inserted into the FIFO at the tail pointer, giving each instruction a unique entry into the FIFO. Upon execution of an instruction, the Writeback stage updates the completed bit in the Active List entry for this instruction.

ArchMapTable module maintains an Architectural Map Table (AMT), containing mappings between architectural registers and physical registers for committed versions of architectural registers. Table A.13 describes the input/output signals to ArchMapTable
Table A.11: Writeback input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed packet</td>
<td>Input</td>
<td>The packet received from Execute.</td>
</tr>
<tr>
<td>LSU packet</td>
<td>Input</td>
<td>The packet received from LSU.</td>
</tr>
<tr>
<td>Load violation packet</td>
<td>Input</td>
<td>The packet received from LSU.</td>
</tr>
<tr>
<td>Bypass packet</td>
<td>Output</td>
<td>The destination physical register and the value of a completed instruction is broadcasted on the bypass bus. This packet is also used for writing value in physical register file.</td>
</tr>
<tr>
<td>Execution-flags packet</td>
<td>Output</td>
<td>The execution-flags, e.g., mispredict, exception, fission-instruction, associated with a completed instruction is sent to Active list.</td>
</tr>
<tr>
<td>Computed address</td>
<td>Output</td>
<td>The target address of the control instruction is sent to Active list.</td>
</tr>
<tr>
<td>Control inst. execution information</td>
<td>Output</td>
<td>Computed information, e.g., branch-direction, target address, of a control information is sent to CTIQ in Fetch-2.</td>
</tr>
</tbody>
</table>
module. Figure A.20 provides detailed logic diagram of ArchMapTable module. ActiveList keeps probing the completed bits for the entries starting from the head pointer, and any completed instructions at the head are committed and removed from the FIFO. When an instruction commits, ActiveList sends updates to AMT with the instruction’s physical destination register specifier and ArchMapTable releases the previously mapped physical register specifier. The released physical register gets added to the circular buffer in SpecFreeList module. In the case of a store instruction, ActiveList signals the SQ to commit the store data to memory. In case of branch misprediction and load violation, ActiveList module sends recovery signal to all pipeline stages.
Table A.12: Retire-ActiveList input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-end ready</td>
<td>Input</td>
<td>The flag is high when back-end’s buffering structures have sufficient space for newly dispatched instructions.</td>
</tr>
<tr>
<td>Active list packet</td>
<td>Input</td>
<td>The Packet is received from Dispatch. It contains PC and logical and destination physical mapping.</td>
</tr>
<tr>
<td>Execution-flags packet</td>
<td>Input</td>
<td>Execution-flags received from Writeback.</td>
</tr>
<tr>
<td>Computed address</td>
<td>Input</td>
<td>Target address of a control instruction received from Writeback.</td>
</tr>
<tr>
<td>Load violation packet</td>
<td>Input</td>
<td>The packet is received from Writeback.</td>
</tr>
<tr>
<td>Active list ID</td>
<td>Output</td>
<td>Active list entry number for newly dispatched instruction.</td>
</tr>
<tr>
<td>Active list count</td>
<td>Output</td>
<td>Count of occupied Active list entries.</td>
</tr>
<tr>
<td>AMT packet</td>
<td>Output</td>
<td>The packet contains logical and physical register numbers of the retiring instruction.</td>
</tr>
<tr>
<td>Commit load</td>
<td>Output</td>
<td>The flag is high when a load instruction is retiring. This flag is used by load queue in Load-Store unit.</td>
</tr>
<tr>
<td>Commit store</td>
<td>Output</td>
<td>The flag is high when a store instruction is retiring. This flag is used by store queue in Load-Store unit.</td>
</tr>
<tr>
<td>Commit control instruction</td>
<td>Output</td>
<td>The flag is high when a control instruction is retiring. This flag is used by CTIQ in Fetch-2.</td>
</tr>
<tr>
<td>Recover flag</td>
<td>Output</td>
<td>Flag is high in case of control misprediction or memory exception (e.g. load violation).</td>
</tr>
<tr>
<td>Recover PC</td>
<td>Output</td>
<td>PC used for fetching next instruction bundle after recover flag is high.</td>
</tr>
<tr>
<td>Exception flag</td>
<td>Output</td>
<td>Flag is high in case of SYSCALL instruction.</td>
</tr>
<tr>
<td>Exception PC</td>
<td>Output</td>
<td>PC used for fetching next instruction bundle after SYSCALL is serviced.</td>
</tr>
</tbody>
</table>
Figure A.19: ActiveList module (part-2) design.

Table A.13: Retire-ArchMapTable input/output signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMT packet</td>
<td>Input</td>
<td>The retirement packet received from Active list.</td>
</tr>
<tr>
<td>Release physical</td>
<td>Output</td>
<td>The packet contains freed physical register that gets added to Free list.</td>
</tr>
<tr>
<td>recover packet</td>
<td>Output</td>
<td>Architectural logical-to-physical register mapping sent from AMT in case of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recovery from mis-speculation or exception.</td>
</tr>
</tbody>
</table>
Figure A.20: Architecture Map Table (ArchMapTable) module design.