NORTHWESTERN UNIVERSITY

Performance and Power Optimization Techniques for High-Performance Processors

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Amirali Baniasadi

Evanston, Illinois
December 2002
To:

Whoever strives to hold back darkness—of whatever kind.
ABSTRACT

Performance and Power Optimization Techniques for High-Performance Processors

Amirali Baniasadi

Rapid performance improvements in the past decade have relied on higher complexity. Modern processors exploit not only more complex structures and units but also faster and larger number of resources per cycle. This trend has created two design consideration. First, exploiting complex structures has resulted in slower clocks. Second, increasing the amount and speed of resources has resulted in increased power dissipation. Future designs will require techniques to solve both problems. This work provides a set of such techniques. We offer performance and power optimization techniques for high-performance processors.

One previously proposed solution to reducing complexity is clustering. An important aspect of clustering is instruction distribution among clusters. In chapter 2, we investigate instruction distribution methods for quad-cluster processors. We study methods with different cost, performance and complexity. We investigate non-adaptive and adaptive methods.

We also introduce and investigate different power optimization techniques.

In chapter 3, we focus on the processor’s front-end. Front-End throttling aims at saving power by selectively deactivating the front-end units when this does not hurt performance. Key to the success of a front-end throttling mechanism are techniques that correctly identify deactivation opportunities without affecting performance. We introduce a new class of front-end throttling methods that exploits information about the number of
instructions passing through the various pipeline stages. These methods save energy by stopping the front-end when we are in the mis-predicted control path and by clustering instruction fetch accesses. We measure performance, energy, power and instruction count reductions.

In chapter 4, we focus on the processor’s back-end. We introduce asymmetric frequency clustering, another technique that reduces the energy dissipated by a processor's back-end while maintaining performance. We introduce an asymmetric dual-cluster, dual-frequency microarchitecture comprising a performance oriented cluster and a power-aware one. We aim at identifying non-critical/critical instructions and executing them in the power-aware/performance-oriented cluster.

In chapter 5, we introduce branch predictor prediction (BPP) as a highly accurate power-aware branch predictor. Our predictor reduces branch prediction power dissipation by selectively and intelligently turning on and off two of the three tables used in the combined branch predictor. We study power and performance trade-offs.
Acknowledgment

“Live as if you were to die tomorrow, learn as if you were to live forever. “- Mahatma Gandhi

There are many people I would like to thank for making my years at Northwestern part of the best and most rewarding years of my life. First and foremost, I would like to thank my advisor Professor Andreas Moshovos. I am indebted to Andreas, for his mentorship, for the great support and attention I have received from him and for being there whenever I needed help. I also would like to thank Dr. Majid Sarrafzadeh, my advisor during my first year at the Ph.D. program. I would like to express my gratitude to Dr. Prith Banerjee, Dr. Alok Choudhary and Dr. Yehea Ismail for serving both on my Orals committee and my thesis review committee. In addition, I would like to express my appreciation to my dear friend Dr. Kiarash Bazargan for sharing his valuable experiences.

My years at Northwestern couldn't have been the same without the support of my beloved wife, Maryam. Her patience and love made the very hard years pass by easier. I would like to acknowledge my beloved parents, and my dear sisters Nazanin, Nargess and Yasamin, for their continuous concern and kindness. I thank my mother, for being a source of love and support during all these years. She always believed in me all along the way. I am indebted to my father for sharing his wisdom, vision and intelligence with me through my life and for never giving up even when everyone else does. I am thankful to my in-laws for their continuos support. I would also like to extend this acknowledgement to my grandparents, my aunts, my uncles and their families. My special gratitude to my late grandfather and former Iran Prime-Minister Dr. Mehdi Bazargan. His sixty years of
struggle for Iran’s freedom, independence and prosperity makes him a role model to generations of Iranians living around the world.

I would like to thank my cousin Dr. Mehran Farid whose knowledge, experience and medical expertise has always been helpful.

Finally, I am grateful to God, who blessed me with a great family, priceless friendships and solid health. He never left me alone, and hopefully, never will.
Table of Contents

1. Introduction .......................................................................................................................... 1

1.1. Contributions ....................................................................................................................... 5
  1.1.1. Instruction Distribution Heuristics for Quad-Cluster Processors................................. 6
  1.1.2. Instruction Front-End Throttling Techniques ................................................................. 7
  1.1.3. Asymmetric Frequency Clustering .............................................................................. 8
  1.1.4. Branch Predictor Prediction ...................................................................................... 9

1.2. Thesis Organization .............................................................................................................. 9

1.3. Experimental Framework ................................................................................................ 10

2. Instruction Distribution Heuristics for Quad-Cluster Processors .................................. 13

2.1. Introduction ......................................................................................................................... 13

2.2. Distribution Trade-Offs ...................................................................................................... 15

2.3. Non-Adaptive Methods ..................................................................................................... 17
  2.3.1. Performance .............................................................................................................. 22

2.4. Adaptive Methods ............................................................................................................ 27
  2.4.1. Performance .............................................................................................................. 29

2.5. Sensitivity Analysis .......................................................................................................... 32
  2.5.1. Inter-Cluster Communication Latency ....................................................................... 33
  2.5.2. Front-End Latency .................................................................................................. 33
References.....................................................................................................................92
List of Tables

Table 1.1.Benchmark Execution Characteristics. Instruction counts (“IC” columns) are in millions.......................................................................................................................11

Table 1.2.Base configuration details. We model an 8-way aggressive, dynamically scheduled superscalar processor having a 256-entry scheduler and an 128-entry load/store queue. Also shown is the default symmetric quad-cluster and asymmetric dual-cluster configuration.............................................................................................................12
List of Figures

Figure 2.1. Example illustrating some of the performance trade-offs in instruction distribution. We assume a dual-cluster configuration and unit latencies for all operations. Furthermore, we assume single-issue clusters. (a) Code fragment with arrows representing data dependences and boxes instructions. (b) A cluster assignment that maximizes performance. Notice that while it takes an additional cycle to propagate C’s result, this does not negatively impact performance. (c) A non-optimal cluster assignment. Execution is delayed while A’s result is propagated between the two clusters....................................................................................................................... 17

Figure 2.2. (a) Performance slowdown of non-adaptive heuristics over a non-clustered architecture. Lower is better. (b) Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reportedly, DDB, MOD1, FF, DEP, LC, BC, SLC and MOD3 from left to right per benchmark (same order as in part (a) with the addition of FF)....................................................................................................................... 22
Figure 2.3. How well some of the non-adaptive methods attack issue-bandwidth and communication restrictions. Four models are simulated per method. The four models are derived by selectively modeling issue-bandwidth and communication restrictions. The models are labeled with an X-Y notation, where X is either I or NI and Y is either C or NC. I indicates that issue bandwidth restrictions are imposed, while NI that they are not. Similarly, C and NC indicate that inter-cluster communication delays are modeled or that they are not respectively. NI-NC corresponds to a non-clustered architecture, while I-C corresponds to a realistic clustered one. Relative performance is reported over the base clustered configuration that utilizes the FF (first-fit) method. Higher is better.

Figure 2.4. (a) Adaptive Method Performance. Performance improvements are over the non-clustered machine. For ease of comparison we also include the corresponding non-adaptive methods. Lower is better. (b) Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reported: DEP, CNT-DEP, MOD3 and MODa from left to right per benchmark.

Figure 2.5. How well some of the adaptive methods attack issue-bandwidth and communication restrictions. Four models are simulated per method. See Figure 2.3 for an explanation of the four models. Higher is better.
Figure 2.6. Performance of some methods when inter-cluster communication latency is increased to 2 cycles. The default centralized configuration without (no communication delays) is the base. Lower is better. ................................................................. 35

Figure 2.7. Performance of some methods with shorter front-end pipelines. Slowdowns are reported over the default non-clustered architecture without any additional front-end stages. Lower is better................................................................. 35

Figure 2.8. Performance when the number of cluster input ports is limited to cluster issue width. All clustered configurations incur 1 cycles for inter-cluster communication. The base, centralized configuration does not incur any communication delays. Lower is better......................................................................................... 38

Figure 3.1. Decoding as fast as possible may not improve performance. (a) Four instructions in program order. (b) Fetch, decode and execute in a conventional 4-way processor. Fetching and decode proceeds as quickly as possible. (c) Decode is gated for one cycle. It still takes 5 cycles to process all four instructions even though some of them are delayed........................................................................................................ 42

Figure 3.2. Performance with various front-end throttling methods. We also include the ideal confidence-based method (ideal). Higher is better.................................................... 48

Figure 3.3.(a) Traffic (instructions) per stage. (b) Energy due to control miss-speculation measured as a fraction of the energy consumed by all instructions................................................. 49

Figure 3.4. Traffic reduction for the bos, dep, dcr, dad, bdad methods. Higher is better................................................................................................................................. 50
Figure 3.5. (a) Total Energy reduction (b) Front end Energy Reduction (c) Issue Energy Reduction for the BOS, DEP, DCR, DAD, BDAD methods. Higher is better...........52

Figure 3.6. Total Power reduction for BDAD.................................................................53

Figure 3.7. Power dissipation and savings distribution among different processor sections:
(a) Power distribution in a non-power-aware machine. (b) Power savings distribution in a power-aware machine (c) Power dissipation in a non-power-aware machine (left bar) and a power-aware machine exploiting the BDAD method (right bar)..........................................................................................................................54

Figure 4.1. An asymmetric-frequency, dual-cluster processor............................................60

Figure 4.2. Relative Performance v.s. (a) DUAL-UF, and (b) CNTR processors. (c) & (d) Variations of the fast cluster only: (c) FAST-ONLY-WIDE, and (d) FAST-ONLY.....67

Figure 4.3. Understanding where performance is lost in the AFC processor. (a) The fraction of instructions assigned to the fast cluster. (b) Relative performance for a non-realistic machine which loses performance only due to instructions running slower (no cluster-induced stalls).........................................................................................68

Figure 4.4. Back-end energy reduction (when the slow cluster uses 0.7 x Vdd) compared to: (a) DUAL-UF processor, (b) CNTR processor (c) FAST-ONLY-WIDE processor (d) FAST-ONLY processor.................................................................................................71

Figure 5.1. (a) Temporal locality among branch instructions. (b) How often do branches use the same sub-predictor to predict the branch outcome.................................................76
Figure 5.2. How different predictor sizes impact (a) performance, and (b) predictor energy consumption. Results are shown relative to a processor that uses a 64K-entry predictor.

Figure 5.3. (a) Performance (higher is better). Bars from left to right compare BPP to the CMB, BMD and GSH. (b) Percentage of cycles that two of the three tables are gated.

Figure 5.4. Relative branch prediction energy consumption for (a) non-banked BPP and (b) banked BPP. Bars from left to right compare to CMB, BMD and GSH.

Figure 5.5. Relative total energy consumption for a processor using a (a) non-banked BPP and (b) banked BPP. Bars from left to right compare to CMB, BMD and GSH machine.

Figure 5.6. Relative energy consumption compared to non-banked CMB and for different predictor sizes.

Figure 5.7. Relative energy consumption for different buffer sizes. Bars from left to right report for buffers sizes 64, 32, 16 and 8.
1. Introduction

Over the past decades we have witnessed an exponential growth in performance. Newer processor designs typically rely on higher frequencies, more transistors and more complex structures to implement advanced micro-architectural techniques yielding significant performance improvements over previous generations. As a result, many modern high-performance processors rely on instruction-level parallelism and are equipped with wide and large instruction windows. Unfortunately, larger and wider instruction windows come with two serious challenges: First, it is now widely believed that simply scaling existing centralized window designs may not be possible without adversely affecting clock cycle and consequently performance. This suggests that alternative methods may be necessary to maintain the current performance growth rate. Second, the higher operating frequencies and the larger number of transistors also result in ever increasing power dissipation. Today some high-performance designs are already power limited. That is, power dissipation imposed constraints has limited the number of on chip resources that can be used per cycle and their frequency of operation. *This thesis investigates the following potential solutions to both performance and power dissipation for future high-performance processors.*

**Performance:** On the performance front we focus on instruction distribution heuristics and organizations for quad-cluster architectures. Clustering has been proposed as an alternative to wide and deep instruction windows. In clustering a collection of smaller windows with associated functional units is used to approximate a much wider and deeper window. Compared to a centralized organization, clustered designs trade-off scheduling
flexibility for higher clock rates. Since each cluster implements a smaller and narrower window, it can be made faster. However, since multiple smaller windows replace a single, larger one, there are less number of resources available to every instruction. In addition, cluster communication may cause additional delays. Consequently, key to achieving high-performance is an effective instruction distribution method, capable of distributing instructions among the clusters so that clustering-induced stalls are minimized. Such stalls are the result of restricted intra-cluster issue bandwidth and of increased inter-cluster communication latency.

We study a variety of instruction distribution methods for quad-cluster processors. We study techniques with varying complexity and cost requirements. These methods utilize various types of information, including the instruction type, dependences, the data flow graph and history to better distribute instructions across clusters. We introduce adaptive methods that improve performance over previously proposed non-adaptive methods. We also introduce models that identify how effectively each method reduces each type of clustering-induced stalls.

**Power Dissipation:** Power dissipation is the second focus area of our research effort. In our work, we are interested in power optimizations at the architectural level. Conventional architectural decisions were concerned with performance, complexity and cost. Power was either an after-thought or not a concern. For this reason existing architectural decisions may be significantly revised taking power dissipation into account. As a result, it is expected that it may be possible to reduce power dissipation while maintaining competitive performance.

We introduce three micro-architectural techniques that achieve power savings
while maintaining performance at a competitive level. We focus on different areas where power improvement may be possible. Specifically, we our optimizations target the processor’s front-end (i.e., fetch and decode), its back-end (i.e., issue, write-back and commit) and the branch predictor. These optimizations are the following:

*Power-Aware Processor Front-End:* We investigate methods for pipeline gating or throttling. Pipeline gating amounts to shutting down part of the pipeline for a while so that it consumes no power (or, to be precise, very little power). Of course, throttling must be used judiciously as it may result in significant performance degradation. Previous work on throttling has focused on reducing the number of mis-speculated instructions that enter the pipeline (due to branch prediction). As we explain in detail in chapter 3, these mechanisms rely on confidence prediction to identify mis-speculated branches. Our work, uses instruction flow information to reduce the power dissipation due to mis-speculated instructions. We rely on information about the instructions that flow through each pipeline stage. We use this information to estimate the amount of instruction level parallelism currently present in the processor. We stall processor front-end when we estimate that sufficient parallelism is already available. Flow-based methods are orthogonal to previous methods. In addition they are independent of the branch organization used by the processor. These methods use power efficient structures and make further power savings possible while maintaining performance in a competitive level.

*Power-Aware Processor Back-End:* Existing modern processors execute all instructions as quickly as possible. However, not all instructions contribute to performance equally. While there are instructions (i.e., critical instructions) that should execute as soon as possible, others could execute at a slower pace without harming performance.
To take advantage of this opportunity, we introduce and evaluate an asymmetric dual-cluster, dual-frequency microarchitecture comprising a performance oriented cluster and a power-aware one. The key idea is to execute critical instructions in the fast cluster and others at the slow cluster. Moreover, as we explain in chapter 4, we offer a practical organization for implementing units operating at different frequencies on the same die.

Our organization aims at: (1) at reducing switching power by executing non-critical instructions slower, and (2) at maintaining performance by executing performance critical instructions as fast as possible. In our organization non-critical instructions are meant to execute in the power-aware cluster that is narrow and uses a lower frequency and power supply. Performance critical instructions are meant to execute in the performance oriented cluster that is wide and uses a higher frequency and voltage supply. By localizing the two frequency/voltage domains, we mitigate many of the complexities associated with maintaining multiple supply voltage and frequency domains on the same chip.

Essential to the success of our technique are methods for distributing instructions across the two clusters intelligently. We present two new heuristics for doing so.

**Power-Aware Branch Predictor:** We propose *branch predictor prediction* or BPP as a power-efficient extension to the commonly used combined predictors. Combined predictors use three underlying *sub-predictors* that are all accessed for every branch. Out of the three prediction hints, one is used to select among the other two. While using three tables makes better prediction possible, it also demands increased power dissipation.

BPP relies on typical program behavior to gate two out of the three underlying sub-predictors for most branches. In particular, BPP exploits the following two phenomena: (1) Often branches tend to use the same sub-predictor. (2) Branch instructions show
strong temporal locality.

We use a small power-efficient structure (BPP buffer) to record the sub-predictor that recent branches have used. Our study shows that due to temporal locality, this small structure captures the vast majority of dynamic branches. Later we refer to this stored information to decide whether two of the three sub-predictors can be gated without harming performance.

1.1. Contributions

The contributions of this thesis are: (1) we investigate instruction distribution heuristics for quad-clustered architectures. Moreover, we introduce adaptive distribution methods. We also study our methods sensitivity to various parameters. Previous work in this area focused on heuristics for dual-cluster architectures. In addition, we propose a modeling methodology that exposes the pros and cons of each heuristic. (2) We introduce and study instruction flow-based front-end throttling methods as a set of techniques that reduce processor power dissipation and stage activity while maintaining performance. These techniques exploit instruction flow information to gate the processor front-end. (3) We present asymmetric frequency clustering (AFC) as a power-aware processor back-end. This technique saves processor power (while maintaining performance) by executing non-critical instructions exploiting power-efficient resources. We also propose two instruction distribution heuristics that effectively exploit the AFC organization. (4) We introduce branch predictor prediction (BPP) as a highly accurate power-aware branch predictor for high performance processors.

In the next few sections we present in detail our contributions in each aforemen-
tioned area. In Section 1.1.1 we present in detail our contributions in the area of instruction distribution techniques for quad cluster architectures. In Section 1.1.2 we discuss front-end throttling. In section 1.1.3 and 1.1.4 we discuss AFC and BPP respectively.

1.1.1. Instruction Distribution Heuristics for Quad-Cluster Processors

- This is the first work that looks at quad cluster designs. Previous work on clustering has focused on dual cluster designs. We investigate the underlying trade-off with the goal of introducing effective instruction distribution methods. For the same reasons that justified dual-cluster designs, building even wider and deeper windows may require additional clusters. For this reason, we chose four clusters as a potential target for future processors. While applying previously proposed dual cluster distribution heuristics to quad cluster architectures is possible, we investigate whether the existence of more clusters (and therefore of higher fraction of resource fragmentation) creates a need for more elaborate heuristics.

- **Besides investigating existing heuristics on quad-cluster architectures, we introduce a set of instruction distribution heuristics.** We investigate a variety of methods with various cost, complexity and performance characteristics, including adaptive and non-adaptive methods. Non-adaptive methods use fixed policies that do not change during runtime. Most previously proposed heuristics for dual-clusters are non-adaptive. We also propose a number of adaptive methods that aim at improving over non-adaptive methods. Adaptive methods may change their decision based on past behavior. Our best non-adaptive method performs within 8% of a non-clustered processor. This is 7% for our best adaptive method.
We propose elaborate heuristics that utilize various types of information, including dependences, data-flow graph depth, instruction types and past behavior. For some, but not all benchmarks, our methods improve over previously proposed methods.

- **We develop a set of models that allow us to identify how well each method attacks different important performance degrading factors (i.e., issue-bandwidth and inter-cluster communication restrictions) in a quad-cluster architecture.**

- **We investigate the sensitivity of our methods to inter-cluster communication latency, front-end pipeline depth and the number of cluster input ports. These are parameters that could change due to practical design restrictions.**

### 1.1.2. Instruction Front-End Throttling Techniques

- **We investigate flow-based front-end throttling heuristics.** While previous work on throttling has focused on identifying branch mis-predictions our methods are oblivious to branch prediction. Instead, they rely on identifying when sufficient parallelism already exists in the pipeline. When this is so, our methods throttle decode/fetch. This delays some instructions from entering the pipeline while having a negligible impact on performance. Our methods utilize simple mechanisms that predict when stalling the front-end will not impact performance significantly. This prediction is done on a cycle-by-cycle basis using information that is available at run-time. In our investigation we have found that a simple heuristic based on information about instruction flow and dependences results in increased power savings and *reduced stage activity* in the decode and fetch unit, while maintaining performance at a competitive level.
We extend previous work in the area of power-aware front-end throttling by evaluating these methods using a power model of a modern high-performance processor. We show that for a subset of SPEC benchmarks, 15.3% of the power dissipated could be saved with a minor 1.9% performance loss. Moreover, we study how front-end throttling techniques affect power consumption in different processor sections and structures.

1.1.3. Asymmetric Frequency Clustering

We introduce and evaluate an asymmetric dual-cluster, dual-frequency microarchitecture comprising a performance oriented cluster and a power-aware one. Our goal is to execute critical/non-critical instruction in the performance oriented/power-aware cluster. Key to success of such a technique are methods that can identify non-critical instructions efficiently. Such methods take clustering induced stalls into consideration efficiently. For example, an instruction that is non-critical in a centralized architecture may become critical in a clustered architecture if it has to communicate across clusters or compete for resources.

We have introduced two new methods that are effective and power efficient. The first, looks at the gap in cycles between the time an instruction writes its result and the time its children, if any, get to issue. This heuristic also uses an approximation of dynamic instruction distance between an instruction and its children. The second new heuristic estimates and uses the completion time of instructions to distribute them between the two clusters.

We show that by exploiting the above heuristics it is possible to save up to 19% of back-end energy with only a 0.8% performance loss compared to a conventional, dual-
clustered processor. When compared to a centralized non-clustered machine we save 41% of the back-end energy with 4.2% performance loss.

1.1.4. Branch Predictor Prediction

- We introduce *branch predictor prediction* as a highly accurate and power-aware branch predictor. Our predictor relies on typical program behavior, to gate two out of the three underlying sub-predictors for most branches. We show that our predictor can reduce branch predictor power considerably over a conventional, banked combined branch predictor.

- We show that a processor with BPP is *always* more power efficient than one that uses any of the sub-predictors alone. On average, BPP reduces energy consumption 22% compared to a non-banked and 13% compared to a banked conventional combined predictor. This comes with a negligible performance degradation.

- We show that while BPP is more effective for larger predictors, it still reduces energy even for smaller branch predictors. More importantly, by comparing energy consumption and performance to alternative designs we show that when one considers the overall processor energy consumption, BPP-enhanced processors always dissipate less energy compared to those using either just the conventional combined predictor, or just one of its underlying sub-predictors (Bimodal or Gshare).

1.2. Thesis Organization

The remainder of this thesis is organized as follows: in Section 1.3 we detail the experimental framework used to evaluate the ideas and techniques we propose. In chapter 2, we discuss instruction distribution heuristics for quad-cluster high-performance proces-
sors. First, we discuss a number of trade-offs relevant to the design of instruction distribution methods. Later, we present a number of non-adaptive and adaptive heuristics and evaluate their performance. We also investigate the sensitivity of the better performing heuristics to increased inter-cluster communication latency, front-end pipeline stages and the number of per cluster input ports. In chapter 3 we discuss front-end throttling methods. First we review the methods we considered in our study and their rationale. Later we report performance, instruction count reduction, and overall processor energy and power savings. Furthermore, we present a breakdown of the relative power dissipation of various key processor structures. In chapter 4 we study asymmetric frequency clustering. We present the rationale of our approach and discuss various heuristics for determining non-critical instructions. Later, we report performance and energy results compared to various alternatives. In chapter 5, we introduce BPP. First we explain our power-aware branch predictor. Later we report performance, and power savings. We report relative power reduction for both the predictor and the entire processor. We compare our model to different alternatives. Moreover, we study how BPP reacts to changes in predictor size and BPP configuration. Finally, we summarize our findings in chapter 6 and offer suggestions on how this work can be extended.

1.3.Experimental Framework

We have used the SPEC’2k programs which we compiled for the MIPS-4-like instruction set architecture used by the Simplescalar simulation toolset [3]. We used GNU’s gcc compiler (flags: -O2 -funroll-loops -finline-functions). To attain reasonable simulation times we simulated 200M of the instructions (committed) after skipping the
initialization. Table 1.1 reports the dynamic instruction count and branch prediction rate. In the interest of space, we use the abbreviations shown under the “Ab.” column.

We have modified Simplescalar’s out-of-order simulator. Unless stated otherwise, throughout this thesis the main architectural parameters of our processor model are as shown in table 1.2. To estimate power we used and modified Wattch [40]. Wattch is a commonly used framework for analyzing microprocessor power dissipation at the architectural level. The power simulation is based on a suite of power models for different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation. We modeled an aggressive 2GHz superscalar microarchitecture manufactured under a 0.1micron technology. To estimate the relevant process parameters, we used the process scaling methodology developed for CACTI [12] and that is incorporated in WATTCH.

<table>
<thead>
<tr>
<th>Program</th>
<th>Ab.</th>
<th>Branch Prediction Rate</th>
<th>Program</th>
<th>Ab.</th>
<th>Branch Prediction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>amm</td>
<td>99%</td>
<td>vortex</td>
<td>vor</td>
<td>98%</td>
</tr>
<tr>
<td>bzip</td>
<td>bzp</td>
<td>97%</td>
<td>vpr</td>
<td>vpr</td>
<td>92%</td>
</tr>
<tr>
<td>equake</td>
<td>equ</td>
<td>98%</td>
<td>wolf</td>
<td>wlf</td>
<td>92%</td>
</tr>
<tr>
<td>mcf</td>
<td>mcf</td>
<td>91%</td>
<td>gcc</td>
<td>gcc</td>
<td>90%</td>
</tr>
<tr>
<td>mesa</td>
<td>mes</td>
<td>99%</td>
<td>compress</td>
<td>com</td>
<td>90%</td>
</tr>
<tr>
<td>parser</td>
<td>prs</td>
<td>93%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1. Benchmark Execution Characteristics. Instruction counts (“IC” columns) are in millions.
<table>
<thead>
<tr>
<th></th>
<th>Default Non-Clustered Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch Predictor</strong></td>
<td>64K GShare+64K bimodal w/ 64K selector</td>
</tr>
<tr>
<td><strong>Fetch Unit</strong></td>
<td>Up to 16 instr. per cycle.</td>
</tr>
<tr>
<td><strong>Instruction Window Size</strong></td>
<td>256 entries</td>
</tr>
<tr>
<td><strong>Load/Store Queue</strong></td>
<td>128 entries, 4 loads or stores per cycle</td>
</tr>
<tr>
<td><strong>Issue/Decode/Commit Bandwidth</strong></td>
<td>8 instructions / cycle</td>
</tr>
<tr>
<td><strong>Functional Unit Latencies</strong></td>
<td>same as MIPS R10000</td>
</tr>
<tr>
<td><strong>L1 - Instruction cache</strong></td>
<td>64K, 2-way SA, 32-byte blocks, 2 cycles</td>
</tr>
<tr>
<td><strong>L1 - Data cache</strong></td>
<td>64K, 4-way SA, 32-byte blocks, 2 cycles</td>
</tr>
<tr>
<td><strong>Unified L2</strong></td>
<td>256K, 4-way SA, 64-byte blocks, 12 cycles</td>
</tr>
<tr>
<td><strong>Main Memory</strong></td>
<td>Infinite, 100 cycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Default Quad-Clustered Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clusters</strong></td>
<td>4, each 2-way issue w/ uniform distribution of functional units</td>
</tr>
<tr>
<td></td>
<td>64-entry windows and 32-entry load/store queues per cluster</td>
</tr>
<tr>
<td><strong>Inter-cluster delay</strong></td>
<td>1 cycle both for registers and store-load forwarding</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Default Asymmetric Clustered Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Schedulers</strong></td>
<td>fast cluster: 64 entries slow cluster: 32 entries, RUU-like</td>
</tr>
<tr>
<td><strong>Inter-cluster delay</strong></td>
<td>2 cycle both for registers and store-load forwarding</td>
</tr>
<tr>
<td><strong>Fetch Unit</strong></td>
<td>Up to 6 instr. per cycle. Max 2 branches per cycle</td>
</tr>
<tr>
<td></td>
<td>64-entry Fetch Buffer</td>
</tr>
<tr>
<td><strong>Decode width</strong></td>
<td>any 6 instructions / cycle</td>
</tr>
<tr>
<td><strong>Load/Store Queue</strong></td>
<td>64 entries, 3 loads or stores per cycle</td>
</tr>
<tr>
<td></td>
<td>Perfect disambiguation</td>
</tr>
<tr>
<td><strong>Issue, Commit width</strong></td>
<td>fast cluster: any 4 insts / cycle</td>
</tr>
<tr>
<td><strong>Decode width</strong></td>
<td>any 6 instructions / cycle</td>
</tr>
</tbody>
</table>

|                        | Table 1.2. Base configuration details. We model an 8-way aggressive, dynamically scheduled superscalar processor having a 256-entry scheduler and an 128-entry load/store queue. Also shown is the default symmetric quad-cluster and asymmetric dual-cluster configuration. |


2. Instruction Distribution Heuristics for Quad-Cluster Processors

2.1. Introduction

Exploiting instruction-level parallelism via out-of-order execution facilitated rapid performance improvements during the past decade. An evolutionary path to continuing this performance growth calls for larger and wider instruction windows. The hope is that such instruction windows will expose more parallelism leading to higher concurrency and hence higher performance. Unfortunately, it is now widely believed that simply scaling the existing centralized window designs may not be possible without adversely affecting clock cycle and consequently performance. There are several reasons why that include the fundamental scaling limitations of centralized designs [12] and the changing semiconductor technology trade-offs, e.g., [2,10] (e.g., it may not be possible to route results within a single cycle in a wide superscalar processor).

Accordingly, clustering has been proposed as an alternative to wide and deep organizations. In clustering, a collection of smaller windows with associated functional units is used to approximate a much wider and deeper window. Compared to a centralized organization, clustered designs trade-off scheduling flexibility for higher clock rates. Consequently, to achieve high performance we need to distribute instructions among the clusters so that clustering-induced stalls are minimized. Such stalls are primary the result of restricted intra-cluster issue bandwidth and of increased inter-cluster communication latency.

Previous work investigated various instruction distribution (or, cluster assignment)
methods for dual-cluster designs [4,6,12] (see Section 2.6. for additional information). Moreover, the ALPHA 21264 processor already uses a dual-cluster core [9]. Building even wider and deeper windows may require additional clusters. However, whether such designs are appropriate requires close investigation of the underlying trade-offs. Accordingly, in this work we investigate instruction distribution methods for a quad-cluster, dynamically-scheduled superscalar organization. We investigate a variety of methods with various cost, complexity and performance characteristics including adaptive and non-adaptive methods. Non-adaptive methods use fixed policies that do not change during runtime, while adaptive methods may change their decisions based on past behavior. We study methods that utilize various types of information, including dependences, dataflow graph depth, instruction types and past behavior. To gain additional insight we also vary intra-cluster issue, inter-cluster communication restrictions and the number of dispatch ports per cluster. These are all central, practical considerations that concern clustered organizations.

Some (i.e., the non-adaptive), but not all of the methods have been proposed and evaluated before in the context of dual-cluster processors. To the best of our knowledge, no other study of instruction distribution heuristics for quad-clustered, dynamically-scheduled superscalar processors has been published. Of course, there is a multitude of architectural parameters that are relevant for clustered designs and for the methods we studied. Moreover, cluster-aware compiler scheduling techniques warrant further attention. Nevertheless, we study a variety of representative configurations varying a set of key architectural parameters.

The rest of this chapter is organized as follows. In Section 2.2., we briefly discuss
a number of trade-offs relevant to the design of instruction distribution methods. In Section 2.3 we present a number of non-adaptive heuristics and evaluate their performance. In Section 2.4, we discuss a number of adaptive heuristics. In Section 2.5, we investigate the sensitivity of the better performing heuristics to increased inter-cluster communication latency, front-end pipeline stages and the number of per cluster input ports. In Section 2.6 we review related work. Finally, in Section 2.7 we summarize our findings and offer concluding remarks. For clarity we use the term *distribution method* in place of *instruction distribution method*. We also use *communication* instead of *inter-cluster communication*. Finally, we use the terms *centralized* and *non-clustered* architecture interchangeably.

### 2.2. Distribution Trade-Offs

In this Section, we discuss the trade-offs involved in developing instruction distribution methods. Throughout this chapter we assume a uniform, quad-cluster organization. The front-end delivers instructions which are then distributed to the four clusters via a distribution mechanism. Our focus is on this distribution mechanism. As we later show, this assignment process can dramatically impact performance. We assume that each cluster contains each own scheduler and set of functional-units. Furthermore, we assume that once an instruction is assigned to a cluster the decision is final (an alternative would be to decouple execution resources and schedulers). Each cluster has its own set of functional units including data cache ports. Dependent instructions can issue back-to-back provided that they both reside in the same cluster. However, propagating results across clusters requires additional cycles.

Throughout this study, our goal is to maximize performance through appropriate
distribution methods. To achieve maximal performance an ideal schedule is needed. However, this is a hard problem even for a centralized architecture. Accordingly, it is convenient to approach distribution as a problem of minimizing clustering-induced stalls compared to an equivalent (i.e., same overall instruction window and resources) centralized architecture. Clustering-induced stalls are either the result of limited per cluster issue bandwidth (and in general, resource distribution including functional units) or of inter-cluster communication latencies.

As a result of non-optimized uniform resource distribution in clustered architectures, each cluster is limited to only a fraction of the total issue slots per cycle (for example, each of the four clusters can issue only 2 instructions of the total of 8 per cycle). Accordingly, it is possible for an otherwise ready-to-issue instruction to get stalled in one cluster while free issue slots exist in other clusters. Moreover, since we assume that it takes additional cycles to propagate results across clusters, it is possible for an instruction to get stalled waiting for data that is currently available at another cluster. However, it is not strictly true that our mechanism should minimize such stalls. To be precise, it is only those stalls that impact the critical path through the computation that are really important. It may be possible to tolerate some stalls. Accordingly, we can categorize stalls into benign (those that do not affect performance compared to a centralized organization) and harmful. An example illustrating some of the trade-offs is given in Figure 2.1.

While maximum performance is desirable, the potential performance benefits of a distribution method should be weighted against its cost and complexity. Of particular concern is the size of any auxiliary structures used. For example, in Section 2.4, we will study a method that significantly improves performance while utilizing sizeable cache-like
17

Moreover, care must be taken to consider not only IPC improvements but also the potential impact on clock cycle and pipeline depth. Besides the number of steps required by the method, particular attention should also be given to the type of information used. It is desirable to use information that is readily available at the decode stage or earlier and preferably early in the clock cycle. For example, in Section 2.3, we will examine methods that utilize dependence information (which is available early prior to assigning clusters) or the number of instructions decoded.

2.3. Non-Adaptive Methods

We have investigated both adaptive and non-adaptive methods. Non-adaptive methods use fixed policies that do not change during run-time. For example, always selecting the cluster with fewest instructions. Adaptive methods, on the other hand, base their decisions on dynamically collected information. For example, whether the cluster
assignment for a particular instruction resulted in a stall last time it was executed. In this Section, we are concerned with non-adaptive methods. Further information on adaptive methods is given in Section 2.4.

We have investigated a variety of non-adaptive heuristics with varying complexity and performance characteristics. Here we restrict our attention to the following representative subset: First-Fit (FF), Modulo (MOD₁ and MOD₃), Dependence-based (DEP), Slice (SLC), Branch-Cut (BC), Load-Cut (LC) and Dependence-Depth-based (DDB). The first two methods do not utilize program-related information, while the rest do. We have considered dependences, instruction types and dataflow depth as alternative sources of program-related information.

**First-Fit (FF):** In this method we assign instructions to the same cluster until it fills up completely. Then we move to the next cluster and do the same. The primary advantage of this method is its simplicity. A possible implementation comprises a per cluster global-AND of the occupied flags of the cluster’s reservation stations (assuming an RUU-like implementation [15]) and a global current-cluster pointer. An incoming instruction is assigned to the current cluster so long there is space available (the cluster’s global-AND signal is 0, i.e., there is at least one free slot available). Otherwise, the current-cluster pointer advances to the next in order cluster¹. The impact of this method on decode/dis-

---

¹ Using a global “is there a free slot available?” signal per cluster makes distribution a serial process; we have to wait until the first instruction is assigned before probing for slot availability for the second instruction. To do in-parallel cluster assignment of multiple instructions we may use a population count circuit per cluster. This does not have to be complete population count circuit as the number of instructions that can be assigned per cycle is limited (i.e., decode width). Accordingly, we only care whether up to that number of slots are available per cluster.
patch latency should be minimal as the information required is independent of the instructions themselves and it could be made available early in the pipeline. While simple, this method makes no explicit attempt to minimize neither communication- nor issue-induced stalls. Nevertheless, dependent instructions tend to be close in the instruction stream. This often helps control communication-induced stalls.

**Modulo Methods (MODₙ):** As we will see in Section 2.3, the first-fit method fails to use issue-bandwidth efficiently. To improve issue-bandwidth utilization while keeping complexity at a minimum, we have investigated a variety of modulo n (MODₙ) methods. In these methods, instructions are assigned to clusters in a modulo n fashion where n is a small integer. For example, in the modulo 3 (MOD₃) method the first three instructions are assigned to cluster 0, the next three to cluster 1 and so on. Compared to FF, these methods distribute instructions more fairly among clusters resulting in a better utilization of issue-bandwidth. We have experimented with a variety of values for n and found that the optimal value differs per program. Here, we restrict our attention to MOD₁ and MOD₃. As with first-fit, the information required by modulo methods can be made available early in the pipeline. While fairly simple, MOD₃ performs surprisingly well.

**Dependence-based (DEP):** Neither of the methods described so far leverages program-related information. The dependence-based method uses data-dependence information in an attempt to reduce communication-induced stalls. In this method we aim to assign dependent instructions to the same cluster. This is done as follows: When decoding an instruction, we attempt to assign it to the same cluster as its parents. If an instruction has multiple parents that are assigned to different clusters we pick the cluster holding the min-
imum number of instructions. Also, if the parents have long committed, we just pick the cluster with the fewest instructions. The data-dependence information required by this method can be made available via the register renaming mechanism.

**Slice (SLC):** Using the DEP method, we often find that the parents of an instruction are assigned to different clusters. This is the result of the limited, forward-dependence-based scope of the DEP method. To further reduce communication-induced stalls it would be better to assign all parents and their consuming child to the same cluster. This is the goal of the slice method. To do so, we employ the method proposed by Canal, Parcerisa and González [4]. An auxiliary, PC-indexed table (the slice table) is used to re-construct the data-flow graph on the fly. Eventually, a common tag is assigned to all instructions belonging to the same *slice*. This tag is used to assign all dependent instructions to the same cluster the next time they are encountered. If no space is available in that cluster we pick the cluster with the fewest instructions. This method reduces communication stalls since instructions within a slice will reside mostly in the same cluster. Moreover, our results show that issue bandwidth is used efficiently. However, these improvements come at the expense of an auxiliary table. Compared to DEP, the slice-table-provided tag can be made available much earlier than the register-dependence information (since the slice table is PC-indexed). We classify this technique as non-adaptive as it does not utilize explicit information about the success of past cluster assignment decisions. We assume infinite slice tables in our experiments.

**Branch- and Load-Cut (BC and LC):** While DEP and SLC offer superior performance they may be too complex or costly to implement depending on the implementation specifics. Accordingly, we investigated methods that leverage other program-related informa-
tion that can be easily extracted at run-time. In particular, we investigated methods that utilize instruction-type information. In the branch-cut method we assign consecutive instructions to the same cluster till we reach a branch instruction. The intuition behind this heuristic is that instructions within a basic-block are mostly dependent. We also investigated variations of the branch-cut method where we changed clusters only on backward branches. In doing so, we were motivated by work in thread-level speculation where loop iterations may be assigned to separate clusters for parallel execution (see Section 2.6). However, we did not observe significant performance improvements. Accordingly, we restrict our attention to the general, all branch-cut method.

We also experimented with a load-cut method where instructions are assigned to the same cluster until a load is encountered. The load and the instructions that follow (till the next load) are then assigned to the next available cluster. The intuition behind this method is that loads often lead a chain of dependent instructions. Accordingly, the hope is that changing clusters upon encountering a load should force mostly dependent instructions to the same cluster, while distributing independent instructions across clusters. Whenever a sequence of adjacent loads is encountered we do not change clusters.

**Dependence-Depth-based (DDB):** Finally, in this method we categorize instructions based on its position (depth) in the DFG (Data Flow Graph). Only instructions currently active in the instruction window are considered in this process. If an instruction has no incomplete parents alive in the window, it belongs to depth 0. If it has only its direct parents alive it belongs to depth 1, and so on. We assign an instruction to the cluster having the least number of instructions of the same level while taking dependence information also into account (when a choice exists, we will assign to the same cluster as its closest
The intuition behind this method is that in a centralized configuration, instructions at the same level would probably issue around the same time (ignoring cache misses and other multi-cycle operations). Therefore by distributing them among clusters we could use the available issue bandwidth more efficiently. While this method may be fairly complex to implement, we include it as it approximates a resource-based scheduling algorithm.

### 2.3.1. Performance

In this Section, we present our analysis of the non-adaptive methods. Clustering is viable only if it results in higher performance. For the methods we studied, this can only be determined if we can measure the performance slowdown.

**Figure 2.2.**

(a) Performance slowdown of non-adaptive heuristics over a non-clustered architecture. Lower is better.

(b) Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reportedly, DDB, MOD1, FF, DEP, LC, BC, SLC and MOD3 from left to right per benchmark (same order as in part (a) with the addition of FF).
be the result of higher operating frequency. It is desirable to know how much faster the
clock rate of the clustered architecture has to be (vs. the centralized architecture’s clock
rate) to result in higher performance. Accordingly, we report performance slowdowns
compared to a non-clustered architecture assuming the same clock frequency. These slow-
downs can serve as bounds on how much faster the clock cycle of the clustered implement-
tation must be. For this reason, we compare each method with a non-clustered architecture
with the same overall resources.

While performance is our ultimate metric, it is desirable to get additional insight
on how each method attacks issue-bandwidth restrictions and inter-cluster communication
delays. To do so, we use a two tiered approach. First we report the fraction of instructions
that are delayed as a result of communication or of issue bandwidth limitations. However,
the two performance degrading factors interact with each other making it difficult to iso-
late their impact. Accordingly, we also study each performance degrading factor indepen-
dently (more on this later on).

Figure 2.2(a) reports relative performance for DDB, MOD1, FF, DEP, LC, BC,
SLC and MOD3 from left to right. The base configuration is a non-clustered architecture.
We can see that on the average and for most benchmarks, DEP does best (average slow-
down is 8%). However, for some of the benchmarks (i.e., mcf, gcc and mes), modulo
methods outperform the rest (average slowdown for MOD3 and MOD1 is 9.4% and
9.5%). On the average FF performs worst (16.4% average performance slowdown).
Instruction-type-based heuristic LC (9.7% average slowdown) offers better performance
compared to BC (11.4% average slowdown). SLC performs similar across all benchmarks.
Average performance slowdown is 11.1% for this method. Finally, while DDB shows a
10.9% performance slowdown.

*Ammp* seems to benefit most from the various methods compared to the other benchmarks. This is mostly due to the relatively very low parallelism (amm’s IPC is around 0.3) for this benchmark which results in a relatively small number of simultaneously active resources. Consequently, this benchmark is less sensitive to limited number of per cluster resources. A similar reasoning may explain why mes and equ lose more performance in our quad-cluster architecture. Mes and equ have the two highest IPCs (5.2 and 3.8 for a non-clustered 8-way machine) and therefore can potentially exploit many resources simultaneously. Apparently, resource per-cluster restriction harms these benchmarks more than others. Figure 2.2(b) reports the fraction of committed instructions that are delayed waiting for a result from a different cluster (lower bar) or because issue-bandwidth was unavailable (upper bar). Here an observation can be made about the relative fractions of instructions delayed due to communication or issued-bandwidth and performance. For the worse performing method (FF), most instructions are delayed due to insufficient issue-bandwidth (upper bar). As we distribute instructions to better utilize issue-bandwidth, communication delays start to become more common (lower bar). Note that a higher fraction of delayed instructions does not necessarily imply lower performance. In some cases, while fewer instructions are delayed, these are more critical and hence performance is worse.

In a realistic clustered configuration, issue-bandwidth restrictions and communication delays interact making it difficult to draw conclusions. Accordingly, we introduce four machine models: **NI-NC, I-NC, NI-C and I-C**. In this notation, **I** indicates that the model includes per cluster issue bandwidth restrictions, while **C** indicates that communi-
cation delays are incurred. The inverse notations, NI and NC, indicate that the model does not include per cluster issue-bandwidth restrictions or inter-cluster communication delays respectively. The NI-NC model corresponds to a non-clustered architecture while the I-C model corresponds to a realistic, clustered architecture. The two other models do not correspond to realistic architectures. However, they provide additional insight on the effectiveness of each method. The NI-C model shows how well we could have done if no issue-bandwidth restrictions were applicable (total issue bandwidth is still limited to 8 instructions per cycle, however, these instructions can come from any cluster, possibly all from the same one). Similarly, the I-NC model shows how well the heuristic performs in attacking issue bandwidth restrictions (no communication stalls possible).

Figure 2.3 reports performance improvements over the base clustered configuration that uses the FF method. Due to space limitations, we restrict our attention to BC, MOD3, and the dependence-based DEP. For the BC method (part (a)) and for most of the benchmarks, I-NC does better the NI-C. This suggests that the inter-cluster delay causes more performance loss compared to the issue slot restriction. This is consistent with figure 2.2 where most stalled instructions are stalled due to cluster communication delay for this method. In part (b) we report the same set of models for MOD3. Here we witness a mixed behavior. While NI-C performs better than I-NC for some benchmarks (e.g., gcc) its the other way around for others (e.g., mes). Finally for DEP (part (c)), NI-C outperforms I-NC suggesting the issue slot restriction causes more performance loss compared to cluster communication delay. Again this is consistent with figure 2.2. As reported in 2.2, quite often, most stalled instructions are stalled because of per cluster issue slot restriction for
This can also be explained by the fact that DEP primarily aims at reducing cluster restrictions. Figure 2.3. How well some of the non-adaptive methods attack issue-bandwidth and communication restrictions. Four models are simulated per method. The four models are derived by selectively modeling issue-bandwidth and communication restrictions. The models are labeled with an X-Y notation, where X is either I or NI and Y is either C or NC. I indicates that issue bandwidth restrictions are imposed, while NI that they are not. Similarly, C and NC indicate that inter-cluster communication delays are modeled or that they are not respectively. NI-NC corresponds to a non-clustered architecture, while I-C corresponds to a realistic clustered one. Relative performance is reported over the base clustered configuration that utilizes the FF (first-fit) method. Higher is better.

DEP. This can also be explained by the fact that DEP primarily aims at reducing cluster
communication by assigning dependent instructions to the same cluster. Therefore, stalls caused by cluster communication are more effectively reduced than those caused by issue slot restrictions.

In this Section, we have discussed and evaluated the performance of various non-adaptive heuristics. We have found that it is possible to significantly improve performance over the simplistic first-fit method. However, we have also found that for many benchmarks, there is still a sizeable gap in performance compared to a centralized architecture operating at the same frequency. In the next section we propose methods that aim at reducing this performance gap.

2.4. Adaptive Methods

In this section, we present and evaluate a number of adaptive methods. The intuition behind these methods is that programs tend to exhibit non-random behavior. Accordingly, it may be possible to learn and avoid inefficient cluster assignments. We have investigated two classes of adaptive techniques. The first class is based on voting, while the second attempts to improve over the fixed modulo techniques we described in the previous section.

Voting-based Methods (CNT-X): The idea behind these methods is to identify problematic instruction assignments and try to avoid them the next time the same instructions are encountered. For example, these methods can improve instruction distribution whenever a program follows the same path repeatedly. In these methods we start with an underlying non-adaptive technique. Upon executing an instruction we record information about the success or failure of the current cluster assignment in a Cluster Prediction Table (CPT).
We experimented with PC-indexed CPTs so that they can indexed early the pipeline. A CPT entry contains four 2-bit up/down saturating counters one per cluster. The counters indicate how appropriate a cluster might be for the matching instruction, with 11 being the best and 00 the least. Initially, all counters are set to 01, indicating that all clusters are equally appropriate. As soon an instruction becomes ready we update the corresponding counter in the CPT. If the instruction can issue immediately, we increment the counter, otherwise and if the instruction is delayed because of a clustering-induced stall, we decrement it. The next time the same instruction is encountered, the CPT is accessed in parallel with the non-adaptive method. The instruction is then assigned to the cluster with the highest counter value (most appropriate based on past experience). If there are more than one qualifying clusters, we use either the non-adaptive method’s recommendation (so long as it is one the clusters with the highest counter values) or choose the cluster with less instructions. We tried different non-adaptive techniques. We only report CNT-DEP here since it is performs best.

**Adaptive-Modulo (MOD₃);** As we have seen in Section 2.3 for many benchmarks MOD3 performed best among the non-adaptive techniques. We have also noted that the best modulo value varied per benchmark, with 3 being a good enough comprise across all benchmarks we studied. Motivated by these observations we have developed the adaptive-modulo method. In this method, we start with an initial modulo value of 3. However, as execution progresses we keep statistics on how often instructions are stalled. After a pre-specified number of instructions have executed (1 million in our experiments) we try a dif-
ferent modulo value (e.g., increase to 4). If the new modulo value results in fewer instructions being stalled, we continue changing the modulo value (e.g., move to 5). Otherwise, we alter our direction of change (e.g., decrement as opposed to increment). Using this policy, the modulo value is dynamically adjusted to one that offers better performance. As described, this policy can get stuck to a local maximum since it relies on comparisons between adjacent values. Accordingly, we have also tried a different policy where we sweep over a pre-specified range of modulo values (i.e., 1 to 16) before deciding on the best one (this scan is repeated at regular intervals, i.e., 1M instructions). However, we did not observe significant performance improvements.

The primary advantage of this method is that it offers some adaptability without requiring many additional resources. A similar method was proposed for selecting an appropriate history-depth for branch prediction [8].

2.4.1. Performance

We report results assuming infinite cluster prediction tables. We have also experimented with finite prediction structures and found that 16K-entry non-tagged, counter-based prediction tables perform very close and sometimes better than the infinite table (better accuracy is possible via constructive interference). As with the adaptive techniques, we first compare their performance with a non-clustered machine assuming the same clock rate. Moreover, we report a breakdown of stalled instructions and use our four models (presented in Section 2.3) to isolate issue-bandwidth and communication related stalls.

We have experimented with various voting-based methods. Here we restrict our
attention to voting-based extensions to DEP. We also study the adaptive-modulo technique. Figure 2.4(a) shows relative performance slowdowns over the non-clustered architecture. For ease of comparison, the relevant non-adaptive methods are also included (repeated from Figure 2.2). As expected, for most cases the adaptive-techniques improve performance over the underlying non-adaptive method. On the average, the performance slowdowns over a non-clustered machine are approximately 7.1% and 8.3% for CNT-DEP and MODa respectively. CNT-DEP has narrowed the gap down compared to DEP. The same is true for MODa compared to MOD3. In absolute terms, the improvements appear relatively minor. However, they are sizeable when compared to the original gap between the best non-adaptive method and the centralized architecture (about 1% off the maximum

Figure 2.4.(a) Adaptive Method Performance. Performance improvements are over the non-clustered machine. For ease of comparison we also include the corresponding non-adaptive methods. Lower is better. (b) Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reported: DEP, CNT-DEP, MOD3 and MODa from left to right per benchmark.
Figure 2.4(b) shows a breakdown of delayed instructions for the adaptive methods. The lower part of each bar reports the fraction of committed instructions that were delayed due to inter-cluster communication. The upper part shows the fraction of committed instructions delayed due to issue-bandwidth unavailability. As shown, CNT-DEP has less instructions stalled due to per cluster issue restrictions compared to DEP. Meantime, it shows an increase in the number of instructions stalled due to cluster communication compared to DEP. While in DEP the number of instructions stalled because of issue slot
restriction delay out weights the number of those stalled due to cluster communication, in CNT-DEP the situation is reversed. MODa also has a slightly different stall distribution compared to MOD3. For many benchmarks, the adaptive nature of MODa causes opposite changes in each kind of stall. Often, an increase in the number of cluster communication stalls comes with a decrease in the issue restriction stalls. Our study shows that while increasing the modulo number reduces the number of cluster communication stalls, in increases the issue slot stalls.

In figure, 2.5 we report four models of issue-bandwidth and communication (see Section 2.3) to determine how sensitive each adaptive method is to each of these restrictions. Here the base case is the clustered FF-based configuration. The general trends with respect to issue-bandwidth and communication restrictions have not changed by much. However, the gap between NI-NC and the other models has been reduced, suggesting that adaptive methods improve performance for both the realistic and theoretical models.

In this section, we have shown that it is possible to further improve performance using adaptive techniques.

2.5. Sensitivity Analysis

In this section, we investigate the sensitivity of some methods to key architectural parameters. In Section 2.5.1, we vary inter-cluster communication latency. In Section 2.5.2, we investigate how our mechanisms react to shorter front-end pipelines. In Section 2.5.5, we study how restricting the number of cluster input ports to per cluster issue width changes the results.
2.5.1. Inter-Cluster Communication Latency

Figure 2.6 reports performance when the inter-cluster communication delay is increased to two cycles. We report slowdowns over the default centralized configuration operating at the same frequency. No additional communication delays are imposed for the base configuration. As expected, the performance gap increases. CNT-DEP is the best performing method being about 8.9% slower than the base. DEP remains the best non-adaptive method with an average slowdown of 10.1%. While MODa still improves over the MOD3, still CNT-DEP performs best.

While DEP loses 2.1% performance when cluster communication delay is increased to two from 1 cycle, MOD3 loses around 6%. Increasing cluster communication delay adds to the importance of cluster communication stalls compared to issue restriction stalls. Therefore, methods like DEP that reduce such stalls more effectively show less sensitivity to cluster communication delay.

2.5.2. Front-End Latency

We also take a look at how decreasing the number of front-end pipeline stages impacts some of the best performing methods. Figure 2.7 reports how performance varies for a one cycle decode stage. We restrict our attention to DEP, CNT-DEP, MOD3, and MODa. We report slowdowns with respect to the default centralized configuration with the same number of decode stages. Inter-cluster communication latency is one cycle. Overall, the performance gap has decreased. However, the relative trends do not change by much. The adaptive methods still perform better than the non-adaptive ones, with CNT-DEP being the best one.
2.5.3. Cluster Input Ports

Finally, we study how our techniques react to a restricted number of cluster input ports. So far we have assumed that each cluster has 8 write ports which corresponds to the aggregate issue bandwidth of our processor. This makes writing up to 8 instructions to every cluster at every cycle possible. Here we experiment how limiting the number of write ports to cluster issue width impacts performance. Having 8 input ports per cluster results in higher complexity and wider buses, therefore we study the effect of lower number of input ports. Accordingly, each cluster can receive up to two instructions per cycle. This makes many methods (e.g., MOD3, MODA, BC and LC) impossible to implement. Such methods, quite often if not always, require assigning more than two instructions to a single cluster at every cycle. Therefore we restrict this study to DEP, CNT-DEP, DDB, SLC and MOD1. Figure 2.8 reports how performance varies when every cluster has only two write ports. We report slowdowns with respect to the default centralized configuration. Inter-cluster communication latency is one cycle. Overall, the relative trends do not change by much, the CNT-DEP remains the best adaptive and the DEP method remains the best non-adaptive method. Compared to the configurations that had no restrictions on the number of cluster input ports, methods such as MOD-1 which distribute instruction uniformly across clusters remain un-affected. Interestingly, restricting the number of input ports improves performance for some benchmarks for SLC. This is a result of a more uniform instruction distribution. Consequently, this makes better resource utilization possible resulting in performance improvements.
A plethora of studies have investigated partitioning as a way of scaling over existing, centralized dynamically-scheduled superscalar architectures. A class of methods aims at extracting parallelism by making non-continuous or large prediction-based steps in the dynamic instruction stream, e.g., [1,7,13,14,16]. Here we restrict our attention to works that investigated partitioning a traditional architecture.

**Figure 2.6.** Performance of some methods when inter-cluster communication latency is increased to 2 cycles. The default centralized configuration without (no communication delays) is the base. *Lower is better.*

**Figure 2.7.** Performance of some methods with shorter front-end pipelines. Slowdowns are reported over the default non-clustered architecture without any additional front-end stages. *Lower is better.*

### 2.6. Related Work

A plethora of studies have investigated partitioning as a way of scaling over existing, centralized dynamically-scheduled superscalar architectures. A class of methods aims at extracting parallelism by making non-continuous or large prediction-based steps in the dynamic instruction stream, e.g., [1,7,13,14,16]. Here we restrict our attention to works that investigated partitioning a traditional architecture.
Palacharla, Jouppi and Smith studied the delay characteristics of key processor structures [12]. They demonstrated that it will not be possible to naively scale existing designs without adverse effects on clock cycle. They proposed using clustering as a solution and studied various non-traditional scheduling mechanisms for dual-clustered architectures (e.g., FIFO-based schedulers) and also used dependences to optimize cluster assignment. Due to the limited space available, an investigation of these alternative scheduler organizations is beyond the scope of this paper.

Farkas, Chow, Jouppi and Vransevic proposed and studied a dual-clustered architecture along with a cluster-aware static scheduling technique [6]. Canal, Parcerisa and González studied a variety of non-adaptive instruction distribution methods also for a non-uniform dual-clustered architecture [4]. They also proposed the slice-based method and explained how slice information can be extracted dynamically. We simulated a variant of their method. Finally, the ALPHA 21264 already employs a dual-cluster micro-architecture [9].

Fields et al. [35] used a criticality predictor to improve performance in clustered architectures. They used criticality-based information and modified the DEP method. Accordingly, when an instruction has two non-complete parents, they assign it to the cluster of the critical predecessor. In addition, they schedule critical instructions to execute before non-critical instructions. While their method performs better than our’s, it comes with additional complexity.

2.7. Conclusion

Clustering provides a potentially viable path for wider and deeper instruction win-
dows and higher operating frequencies. In this work, we have studied a variety of instruction distribution methods for quad-cluster processors. We studied non-adaptive methods and adaptive techniques with varying complexity and cost requirements. These methods utilized various types of information, including instruction-type, dependences and past history to better distribute instructions across clusters.

We have found that a non-adaptive dependence-based method, DEP offers performance within 8% of a non-clustered organization operating at the same frequency. Moreover, we have seen that it is possible to reduce this gap down to about 7.1% via a counter-based prediction scheme. While in absolute terms this is a minor improvement, it does represent a sizeable reduction in relative terms. We have also investigated the sensitivity of our methods to inter-cluster communication latency, front-end pipeline depth and the number of cluster input ports. We found that performance is much more sensitive to inter-cluster communication for the better performing methods. The performance gap for the best performing method increased to 10.2% when inter-cluster communication latency was increased two cycles. In contract, when using a shorter pipeline front-end stages this gap was 8.2%.

While we studied a reasonable set of configurations and methods, there is still a plethora of design points and possible other methods that warrant further study. There are multiple directions for further experimentation, including non-uniform cluster organizations, restrictions on inter-cluster communication bandwidth, the effect of previously proposed compiler optimizations [6] and alternative scheduler designs such as those appearing in [12]. Of particular interest are organizations where execution clusters (i.e., functional units, register files and cache ports) and schedulers are decoupled. In such a
design, an instruction is first assigned to a scheduler, and then, based on input operand availability is sent to the appropriate execution clustered.

Figure 2.8. Performance when the number of cluster input ports is limited to cluster issue width. All clustered configurations incur 1 cycles for inter-cluster communication. The base, centralized configuration does not incur any communication delays. Lower is better.
3. Instruction Flow-Based Front-End Throttling

3.1. Introduction

Successive high-performance processor generations have thus far relied on higher frequencies and more transistors to improve performance. Unfortunately, using more transistors clocked at higher frequencies requires more power. As a result, power dissipation in modern processors is now quickly approaching alarming levels jeopardizing further advances in performance. Finding ways to control further increases in power is imperative for future generation processors with billion transistors operating at multi-GHz frequencies.

In this work we are concerned with power reduction techniques at the architectural level that are complementary to low-level circuit techniques. Our goal is to revisit existing architectural decisions revising where necessary so that energy is used more efficiently. This is possible, as existing architectural decisions ignored power for the most part, focusing instead on performance, complexity and cost trade-offs.

A large fraction of power in modern high-performance processors is dissipated by the front-end, that is by instruction fetch and decode. For example, instruction decode and fetch dissipate 28% of overall (average) processor power in the Intel P6 [17] and as much as 36% for the less complex MIPS R3000 [21].

In this chapter, we are concerned with power-aware instruction fetch/decode designs for dynamically scheduled, superscalar processors. Previous work in this area exploits energy inefficiencies that are caused by incorrect control flow speculation (or, in
the interest of space, *speculation*) [17]. Incorrect speculation does not improve performance while it leads to extraneous instructions passing through the pipeline consuming additional energy (as we show in section 3.3, these instructions can be as much 4 times more than those executed by the program). Previous power-aware front-end proposals assign confidence to speculation decisions. They turn-off (or *gate* or *throttle*) the front-end on low confidence speculation decisions (i.e., when it is highly likely that the front-end is processing instructions down an incorrectly speculated control flow path). We explain their operation in detail in section 3.2. Power is reduced significantly since the number of extraneously processed instructions is reduced. Of course, currently it is not possible to always predict when the front-end has wandered off to an incorrect control flow path.

In this work we introduce a new class of power-aware front-end throttling methods that are complementary to previously proposed methods. Our methods are oblivious to control flow speculation. They rely on another well understood inefficiency of conventional front-end designs: Existing designs process instructions with the maximum possible rate since this maximizes the chances of exploiting as much parallelism as possible. However, this does not always improve performance significantly. Our methods exploit this inefficiency by slowing down the front-end (they turn it off for a couple of cycles) when they predict that this will *not* reduce performance. This prediction is based on local information about the instructions passing through various pipeline stages, or *instruction flow* or *traffic*. Since our methods are oblivious to control flow speculation they also reduce the number of extraneous instructions appearing on miss-speculated control flow paths. For this reason, they also reduce power demands. We make the following contributions:
• A minor contribution of this work is that we demonstrate that the previously proposed speculation confidence-based methods remain effective for future, highly-aggressive processor designs.

• A major contribution of this work is that we introduce a new class of power-aware front-end throttling mechanisms that rely on instruction-flow information. We demonstrate that these methods can complement existing speculation confidence-based methods, further decreasing power dissipation.

The rest of the chapter is organized as follows. In section 3.2, we explain the rationale of our approach and discuss the various heuristics. In section 3.3, we discuss and report both performance and power related results. Finally in section 3.4, we summarize our findings and offer concluding remarks.

3.2. Power-Aware Front-End Throttling

Conventional front-end designs rely heavily on control flow speculation to forge ahead into the dynamic instruction stream. Speculation allows a processor to guess the target of a branch without waiting for it to execute. Consequently, the processor can speculatively fetch and start executing the target instructions. Eventually, when the branch executes, the processor verifies its guess. While speculation greatly improves performance, it also increases power dissipation. In particular, instructions that execute due to incorrect speculation consume power while they do not contribute to performance. As we also show in section 3.3, the number of extraneous instructions can account for as much as 80% of all instruction passing through the front-end stages. Unfortunately, not using speculation in most modern processors is not an option as the performance penalty is too high.
Moreover, as we move toward deeper pipelines and wider/deeper instruction windows, the need for speculation increases.

Previous work on power-aware front-end designs focused on using speculation confidence mechanisms to reduce the number of extraneously executed instructions. Speculation confidence mechanisms predict when the front-end is fetching instructions down an incorrectly speculated control flow path. If so, they turn the front-end off until confidence is regained, an action referred to as *gating* or *throttling*. Of course, if a perfect confidence mechanism was possible, then perfect speculation would be possible too.

In this work we propose a new class of front-end throttling methods that are orthogonal to existing confidence-based methods. Our methods attack another well known power inefficiency of modern processor designs. In particular, conventional processors strive to fetch and decode as many instructions as possible and as quickly as possible. However, it is well understood, that the additional parallelism that may be exposed (if

---

**Figure 3.1.** Decoding as fast as possible may not improve performance. (a) Four instructions in program order. (b) Fetch, decode and execute in a conventional 4-way processor. Fetching and decode proceeds as quickly as possible. (c) Decode is gated for one cycle. It still takes 5 cycles to process all four instructions even though some of them are delayed.
any) does not always improve performance. Often times, the processor is stalled waiting for some other instructions to complete. To understand this, let us consider the example of figure 3.1 and let us focus on decode power. Shown in part (a) is sequence of four instructions, $a$ through $d$. Instructions $a$ and $b$ are independent, while $d$ depends on $c$. In part (b) we see how these instructions may be fetched, decoded and executed in a conventional 4-way processor. Even though the processor has the decode resources to process all four of them in a single cycle, the fetch stage delivers them in pairs in two consecutive cycles. This can happen, for example, if the four instructions appear on two different cache blocks (which are accessed in separate cycles). As a result it takes a total of 5 cycles to execute these instructions. In part (c) we turn off decode for one cycle (we omit the fetch stage in the interest of space). As a result, we decode all four instructions at the same time (third cycle). Even though $a$ and $b$ are decoded a cycle later than they did in part (b), notice that it still takes a total of 5 cycles to process all four instructions.

We can exploit this phenomenon to throttle the front-end whenever we predict that fetching/decoding as quickly as possible will not lead to significant performance improvements. As a result, instructions are fetched just in time for exploiting parallelism. This applies to both correctly and incorrectly speculated instructions. In the latter case, since fetch/decode will proceed at a slower pace, we essentially reduce the number of incorrectly speculated instructions that enter the pipeline and hence we reduce power. Our methods work by selectively disabling fetch and decode on a cycle by cycle basis. These methods reduce average dynamic power. In the sections that follow we review the heuristics we have investigated.
3.2.1. Flow-Based Heuristics

Our methods are instruction flow-based or flow-based. They rely on information about the instructions that flow through each pipeline stage. They use this information to estimate the amount of instruction level parallelism currently present in the processor. They stall fetch/decode when they estimate that sufficient parallelism is already available. In this case, the assumption is that introducing additional instructions would not impact performance significantly. The decision to throttle the front-end is done on a cycle by cycle basis. When so determined, we simply disable fetch and decode for 3 cycles (the decode stage actually occupies 3 pipeline stages in our configuration which is heavily pipelined). Among the various method we tried, we report the best three: Decode/Commit Rate, Dependence-based and Adaptive.

Decode/Commit Rate (DCR): This technique estimates that sufficient parallelism exists when the number of instructions passing through decode exceeds significantly the number of instructions that commit. Intuitively, branch miss-speculations will result in more instructions being decoded than committed. Also, when there is little parallelism we will temporarily observe many instructions being decoded (while filling up the window) and few being committed.

This technique simply compares the number of instructions decoded and committed during each cycle. Best results where obtained when we throttled the front-end when three times as much instructions were decoded than committed.

Dependence-Based (DEP): As another estimate of available parallelism this method simply inspects the instructions currently being decoded counting the depen-
dences among them. Whenever the number of dependences exceeds a pre-specified threshold we stall the front-end during the next three cycles. We tried different threshold values and the best results were possible with the threshold was set to half the decode width. The intuition is that a high number of dependences is an indication of a long and probably critical computation path. Consequently, it is unlikely that introducing additional instructions as quickly as possible will have a significant impact on performance. We have also experimented with taking into account dependences with the instructions already in the window. However, the results were not promising. This heuristic is also fairly straightforward to implement. Since dependence checking is done during decode, the necessary information is already available. Accordingly, the power overhead should be negligible.

**DEP-and-DCR (DAD):** As we will show in section 3.3.1, sometimes DEP works better than DCR and vice versa. To capture the best of both worlds, we developed an adaptive technique. DAD combines DCR and DEP and stalls the front-end if both methods instruct us to do. While this reduces the number of turn-off times compared to the previous two techniques, in all cases, performance is either better or very close to the best of the two underlying methods.

### 3.2.2. Confidence-Based Heuristics

Previously proposed *control flow confidence-based* heuristics, or simply *confidence-based* heuristics aim at reducing the number of erroneously executed instructions due to control flow miss-speculations. Of course, it is not possible to identify miss-predictions early enough to avoid fetching and decoding instructions down the miss-predicted path. Otherwise, perfect branch prediction would be possible and there wouldn’t be any
miss-speculations to start with. For this reason, these methods rely on confidence mechanisms to identify low confidence speculation decisions. In investigating these heuristics we draw from the suggestions and experience reported in [17]. In this work, we investigate the both-strong (BoS) method. This method employs a simple mechanism for identifying low confidence branches. These are branches for which we have experienced low prediction accuracy. To throttle the front-end we count the number of low confidence branches in the pipeline. We stall the front-end when this number exceeds a pre-specified threshold. BoS deems a branch as low confidence when any of the two underlying branch predictors (we use a combined, McFarling predictor) is not strongly biased. While Manne et al. [17] suggest additional confidence-based methods, BoS performed the best for our processor model. Therefore, and in the interest of space, we restrict our attention to this confidence-based method.

3.2.3. Combining Confidence and Flow

The last method we report, combines the BoS and the best flow-based heuristic (DAD). This is the BDAD method which combines BoS and DAD. Here we throttle the front-end when either BOS or DAD instruct us to do so. We will show that this method outperforms all others in energy reduction while offering competitive performance.

3.3. Results

In this section, we present our analysis of the various front-end throttling methods. We report performance results in section 3.3.1. We report energy- and power-related measurements in sections 3.3.2 through 3.3.4.
3.3.1. Performance

Reducing the rate of fetch/decode can negatively impact performance. Accordingly, we first investigate how the various methods impact performance. Figure 3.2 reports performance for all methods. Performance is reported relatively to the base configuration. The base configuration never throttles the front-end. Numbers lower than one represent slowdowns. We include an ideal confidence mechanism (IDEAL) that never fetches miss-speculated instructions. While this is impractical, it provides us with insight on what it would be possible with a perfect confidence-based method. Interestingly, in some cases (i.e., equ, gcc and prs), ideal outperforms the base suggesting that miss-speculated instructions may not have positive side-effects (e.g., prefetching). Both ideal and BoS show an average slowdown of 1.1%.

The flow-based DEP and DCR are not robust. For example, DEP does not perform well for the two benchmarks (bzp and mes). On the other hand, while DCR provides better performance for bzp and mes (where DEP fails) it does not perform well for equ (where DEP provides good performance). Fortunately, the DAD method indeed captures the best of both underlying techniques offering acceptable performance across the board. DAD performs either better than or close to the best of two underlying techniques. On the average, DAD, DCR and DEP are within 0.8%, 1.5% and 2.1% of the base case respectively. Finally The combined method BDAD, performs within 1.9% of the base.

3.3.2. Traffic and Energy Overhead

We start by establishing that control-flow miss-speculations increase the number of instructions passing through various pipeline stages and these instructions result in a
considerable energy overhead. Figure 3.3 reports the traffic and energy consumption due to control miss-speculation. We report traffic for the fetch, decode, issue and complete stages. Part (a) reports the number of extraneous instructions passing through different stages. This is expressed as a fraction of all instructions passing through that stage (e.g., 0.7 means that 3 in 10 instructions passing through that stage are actually committed). Part (b) reports the total energy overhead due to miss-speculations. Here, extra traffic is higher for decode and fetch since fewer miss-speculated instructions get to issue or complete. There is a correlation between the percentage of extra instructions for each benchmarks and its energy overhead. The exact overhead of course varies as it depends on the actual instruction mix. On average, 24.4% of the total energy is wasted down the wrong instruction path. The extraneous instructions that consume this energy on the average represent 51%, 39%, 26% and 26% of the instructions passing through the fetch, decode, issue and complete stages respectively.
Having shown that miss-speculations greatly increase instruction traffic and result in a considerable energy overhead, in this section we study how well the various throttling methods perform. Specifically, we look at both instruction traffic and energy reduction. We report energy instead of power (energy over run-time) to focus on savings achieved by intelligent throttling (rather than power savings caused by longer runtimes).

Figure 3.4 reports *traffic reduction* for all methods we studied. Reported is the total number of instructions passing through each stage measured as a fraction of the total number of instructions passing through the same stage in the original non-power-aware configuration. Both correctly speculated and extraneous instructions are included in this metric.
Figure 3.4. Traffic reduction for the bos, dep, dcr, dad, bdad methods. Higher is better.
For example, ideally we could see a fetch traffic reduction of 72% in gcc. Parts (a) through (e) show the results for BOS, DEP, DCR, DAD and BDAD respectively. As shown, in all pipeline stages, BDAD shows more traffic reduction than other methods. On average, BDAD reduces traffic by 11%, 15.1%, 6.8% and 6.8% at the fetch, decode, issue and complete stages.

Figure 3.5 reports energy consumption reduction for all methods. Part (a) reports total energy reduction measured over all stages. Here BDAD reduces total energy by 12.2%, DEP is second with a 10.5% energy reduction. Part (b) reports front-end energy reduction. On average BDAD reduces front-end energy requirements up to 17.8%. Part (c) reports issue energy reduction. Issue is the major energy consumer in the back-end of the processor. On average, BDAD reduces issue energy by 8.3%. As shown, the highest energy savings are observed for mcf while they are miniscule for amm and mes. This is consistent with figure 2(a) and 2(b) where amm and mes show lower energy overheads while mcf shows the highest. This is due to the high branch prediction rate for these two benchmarks.

3.3.4. Power Reduction and Breakdown

Finally, we report power measurements. These measurements complement the energy measurements by also taking into account the variation in cycle count introduced by the various throttling methods. We restrict our attention to the best performing method, BDAD.

Figure 3.6 reports power (energy per cycle) consumption reduction for BDAD. Here bars from left to right report total, front-end and issue power dissipation reduction.
On average BDAD saves 15.3\%, 19.3\% and 10.3\% of the total, front-end and issue power dissipation respectively. As shown, power savings are higher than energy savings. This is explained by the increased cycle count caused by front-end throttling. While this increase results in 1.9\% average performance loss for BDAD, it improves power savings. Throttling methods not only reduce the energy consumption, but also expend the reduced energy in a longer period of time.

Figure 3.5.(a) Total Energy reduction (b) Front end Energy Reduction (c) Issue Energy Reduction for the BOS, DEP, DCR, DAD, BDAD methods. Higher is better.
In figure 3.7 we report average breakdown of power dissipation for various key processor structures measured for the same subset of SPEC benchmarks used through this chapter. We report how our methods impact power dissipation in different processor sections. We do so for the base and the power-aware processors. We also report a breakdown of the power savings per structure. In part (a) we report power distribution in the base machine. The two most power hungry structures are the instruction window and the ALU. They consume 20.8% and 19.7% total power respectively. In part (b) we report the fraction of total power savings per structure. As shown, the data cache and the branch predictor account for 25.8% and 23.6% of the total power saved. This is 15.6%, 8.9% and 8.5% for the instruction window, the result bus and the clock. In part (c) we report absolute power consumption for both machines. The left bar reports absolute power consumed in the base machine. The right bar reports the absolute power in the power-aware machine. As expected, most of the power savings are due to structures that mostly interact with front-end instructions.
3.4. Conclusion

We extended previous work in the area of power-aware front-end throttling by introducing flow-based front-end throttling techniques and evaluating these methods using a power model of a modern high-performance processor. Flow-Based methods are
orthogonal to previously proposed methods. In addition they are independent of the branch predictor organization used by the processor. More importantly our methods rely on power-efficient structures and impose very little power overhead. We showed that our methods complement previously proposed methods and make further power savings possible. We showed that 12.2% of the energy (and 15.3% of power) consumed could be saved by exploiting an intelligent method that combines confidence-based and the recently proposed flow-based front-end throttling methods. This considerable reduction in energy is possible with a minor 1.9% performance loss. Moreover, we studied how front-end throttling techniques affect power consumption in different processor sections and structures. As expected, structures interacting mostly with front-end instructions benefit more from the savings than those in the issue- and complete-logic.
4. Asymmetric-Frequency Clustering: A Power-Aware Back-End

4.1. Introduction

Frequency and voltage scaling are two commonly used circuit-level techniques that offer a trade off between latency and switching (dynamic) power. These techniques have been used extensively to reduce power in the non-critical circuit paths of modern high-performance processors. Further power benefits may be possible when these circuit-level techniques are combined with architectural-level methods. One possibility is to use lower power and consequently performance units for executing non-critical instructions. These are instructions whose execution time can be prolonged without impacting performance. There are two main challenges: (1) Developing power-aware heuristics that can identify non-critical instructions with sufficient accuracy and little power overhead, and (2) building pipelines that successfully mix different supply voltage and frequency domains.

In this work, we focus on how the processor’s back-end can be changed to exploit frequency and voltage scaling in combination with architectural methods for identifying non-critical instructions. The back-end comprises the structures that perform instruction issue, execution, completion and commit. There are two reasons why we focus on the back-end: First, there is great potential for reducing overall power since a large fraction of power in high-performance processors is dissipated by their back-end. By using WATTCH [40] we estimated that the power consumed by the back-end of a superscalar, 6-way issue, 96-entry window processor is about 54% of the total processor switching power. Second,
most of the front-end power is dissipated by SRAM-like structures where a reduction in supply voltage may compromise cell stability and correct operation.

We introduce asymmetric-frequency clustering (AFC) as a power-aware back-end that leverages frequency and supply voltage scaling for reducing power dissipation. We study an AFC processor comprising two clusters: one that is a performance oriented conventional cluster and another low-power oriented cluster. The fast cluster operates at twice the frequency that the slow cluster operates at. This frequency differential facilitates the use of a lower supply voltage at the slower cluster resulting in further power benefits. We argue that AFC simplifies the design of dual-frequency, dual-voltage processors as it localizes the two frequency/voltage domains into two coarse grain blocks defining a clear architectural and physical interface between them. Moreover, the AFC maintains the frequency advantage of clustering [32][33]. Key to the success of AFC is the ability to predict those non-critical instructions that can be safely executed twice as slow without reducing performance. We present three heuristics for doing so that are based on local information. One of the heuristics is a modification of a previously proposed criticality predictor [27].

Our contributions are: (1) We introduce new, simple to implement and effective non-criticality detection heuristics, and (2) we demonstrate that when these heuristics are used with AFC they can significantly reduce power while maintaining high performance as compared to various important alternatives.

The rest of the chapter is organized as follows: In section 4.2, we present the rationale of our approach and discuss various heuristics for determining non-critical instructions. In section 4.3, we commend on related work. In section 4.4, we report performance
and energy results. Finally in section 4.5, we summarize our findings and offer concluding remarks.

4.2. Asymmetric-Frequency Clustering

The goal of AFC is to use units operating at a lower voltage supply and frequency \textit{intelligently} only for those instructions that are non-critical. Previous work has shown that there are many instructions that could be delayed without impacting performance [25]. Processors that use dynamic voltage/frequency scaling exist. However, adjustments to voltage/frequency are done at a coarse grain (e.g., thousands of cycles) since the circuits that facilitate this scaling incur considerable delay and power overheads that have to be amortized over long periods of time. AFC aims at providing the flexibility to use lower supply voltage and frequency units on a \textit{per instruction basis}. In AFC, the processor’s resources are \textit{statically} divided into frequency/voltage domains at a coarse level. Switching power can be reduced by \textit{intelligently} distributing instructions to the different voltage/frequency domains without requiring dynamic voltage/frequency scaling.

In general, mixing multiple voltages and frequencies on the same circuit is a challenging task, e.g., [37]. Different power supply lines are required, the design of the clock distribution network becomes more complex and noise considerations increase. Of particular concern is efficient interfacing between resources operating under different voltages and frequencies. For example, driving a higher supply voltage circuit using the output of a lower supply voltage circuit requires conversion of the voltage levels. Such circuits impose area overheads, can be slow and may dissipate relatively high power since the lower voltage levels may not be sufficient to completely turn-off transistors operating at a
higher supply voltage. Finally, using a lower voltage supply is highly questionable in some cases. This includes the various SRAM-based structures (e.g., caches and branch predictors).

In this work, we argue that clustering \cite{32,33} can be used to minimize the complexities associated with designing and interfacing circuits operating at different frequencies and supply voltages. To illustrate the potential of this method, we statically divide the processor’s back-end into two different clusters. Each cluster operates under a different frequency/supply voltage domain. We devote one cluster to power efficient resources (low voltage and frequency) while the other cluster uses performance oriented resources (high voltage and frequency). We refer to such clustering as frequency-clustering. With frequency-clustering, we restrict interfacing to cluster boundaries. In AFC, voltage shifting is limited to a set of clearly identified buses that connect the two clusters. These buses are used for inter-cluster communication. In addition, by localizing both frequency and voltage domain we reduce the associated voltage/frequency distribution network. Frequency-clustering saves energy both by executing a fraction of instructions at a lower frequency and voltage and by distributing what otherwise would be large centralized resources (e.g., a larger central scheduler as opposed to two smaller ones). The latter benefit is inherent to any clustered architecture \cite{34}.

4.2.1. AFC Architecture

In figure 4.1 we show our AFC processor. The processor’s back-end comprises a 4-way performance-oriented cluster (FASTC) and a 2-way power-aware cluster (SLOWC). FASTC runs at a frequency that is twice faster than that of SLOWC, has a 64-
entry window and can issue up to four instructions per cycle. SLOWC has a window size of 32 instructions that can issue up to two instructions per cycle. All memory references are performed in the fast cluster. This is done, so that we do not worry about caches operating at different voltages. Inter-cluster communication latency is at least two cycles.

FASTC has six writeback ports since it may receive up to four results from itself and two from SLOWC. Since SLOWC is twice as slow, inter-cluster communication is limited to even cycles. FASTC can receive up to four results during odd cycles and up to six during even cycles. SLOWC receives results only on even cycles. Since FASTC may be producing results during odd cycles too, a FIFO buffer is provided between the two clusters. This *inter-cluster communication buffer* also serves to minimize the number of writeback ports in SLOWC as follows: In the worst case, SLOWC may need to receive up to 10 results in a single cycle: two from itself at eight from FASTC if the latter committed four instructions during the last two cycles. That would require a total of 10 writeback ports at SLOWC. We found that doing so greatly increases power at SLOWC. Accord-
ingly, we limit the number of results that can be simultaneously received by SLOWC to six. The inter-cluster communication buffer serves to smooth out communication. In our studies, we used a 16-entry buffer. If this buffer is full we stall FASTC until the SLOWC frees up four entries. An alternative would be to leverage control-misspeculation hardware to squash and restart execution from the instruction that overflows the buffer.

4.2.2. Cluster Assignment Heuristics

The goal of the cluster assignment heuristics is to identify those non-critical instructions than can execute twice as slow. We present a number of heuristics that utilize local information only. Since our goal is to reduce power consumption, in the evaluation we take into account the power dissipated by the auxiliary structures required by the heuristics. With all heuristics, we initially assign all instructions to FASTC. We assign instructions to SLOWC when instructed by the heuristic. Our goal is not to develop the best heuristic possible. Rather, we aim at demonstrating that even simple heuristics can result in significant power improvements. We have experimented with various heuristics and present the best ones here.

**Generation-Time Gap (GTG):** This heuristic is a combination of two separate heuristics. The GTG heuristic deems an instruction as non-critical if any of the two underlying heuristics does so. The first, looks at the gap in cycles between the time an instruction writes its result and the time its children, if any, issue. The intuition here is that if an instruction writebacks a result and its children do not issue immediately, then they are probably waiting for some other instruction to finish. Since it takes 2 cycle to communicate results across clusters and since SLOWC is twice as slow we use four cycles as our
threshold (two cycles for executing most instructions, plus two for inter-cluster communication). The second heuristic, uses an approximation of dynamic instruction distance between an instruction and its children. If an instruction is dispatched much earlier than its children, this is an indication that it appears much earlier in the instruction stream. Empirically, we found this to be a good indicator that the parent can be delayed. In this study we used two cycles as the threshold. That is, if no children are dispatched in the next two cycles after an instruction is dispatched the latter is deemed as non-critical.

In both cases, the determination that an instruction is non-critical is available after the instruction was dispatched and assigned to a cluster. Accordingly, we use a table to predict if future instances of the same instruction are non-critical. In our experiments we used a 4k table. Every entry contains six bits. These are used as a six-bit saturating counter. Entries are allocated only for critical instructions as they commit. If an entry exists, it is also updated at commit time. The entry is initially set to 32. When we detect critical instructions we increment the counter by eight, otherwise we decrement by one. We assign an instruction to FASTC unless this table deems an instruction as non-critical. An instruction is deemed non-critical at dispatch if the table-entry found is below or equal a threshold (16 in our case).

**Young in Queue (YIQ):** This method is based on the QOLD criterion proposed by Tune et al.,[27]. The QOLD method marks the oldest instruction in the instruction queue as critical during each cycle. Once the oldest instruction is marked as critical, we also mark its parents as critical. Upon commit, we update a prediction table as we did with the GTG method. This method also uses a prediction table similarly to the GTG method. Again we assign instructions to FASTC unless they are deemed as non-critical. An
instruction is deemed as non-critical if the prediction table contains an entry that is below the pre-specified threshold.

**Complete Time Estimation (CTE):** If we had an oracle and we knew in advance when an instruction will complete, then a heuristic for identifying non-critical instructions during dispatch would be as follows: Compare your complete time with the maximum complete time of all preceding instructions that are currently in the window. If this instruction will be completing earlier, it will be forced to wait for that other instruction to commit. Hence, it may be safe to delay this instruction. Of course, we cannot have such an oracle. Instead, we estimate the completion time of instructions as they are dispatched. This is straightforward: We associate a completion time with every register. As instructions are dispatched we predict how long they will take to execute once issued (that is, we “predict” the latency of the functional unit they will be executing). We also obtain the maximum completion time for its source operands. Adding the two we obtain an estimate for when this instruction will complete. We then compare this estimate with the maximum completion time seen thus far (only one entry is required). Modern processors already predict instruction latencies for dynamic scheduling purposes.

### 4.3. Related Work

Seng *et al.* [22], suggested exploiting slower functional units for processing non-critical instructions. Our measurements show that in addition to functional units, other back-end sub-sections consume a considerable part of processor back-end energy (*e.g.*, 76% for gcc in a 6-way, 96 instruction centralized machine). Therefore, we extend prior work to cover the processor back-end (*e.g.*, instruction selection and wake-up). G. Mosh-
nyaga suggested a complexity adaptive issue logic where voltage supply was changed dynamically [23]. We extend the voltage duality to the entire back-end and remove supply voltage switching and its associated costs by using clusters. Pyreddy and Tyson [24] studied how exploiting dual speed pipelines affects processor performance. They used heuristics to mark and send instructions through execution paths with varying latencies. Their heuristics used profiling methods for marking instructions, while we focus on dynamic mechanisms. In addition we take into account voltage reduction effects too. Casmira and Grunwald [25] defined and measured slack as the number of cycles than an instruction can wait before being issued and becoming critical. Semeraro also suggested using multiple frequency and voltage domains throughout the processor and showed that fine grain voltage/frequency scaling this can lead to significant power benefits [36]. In our work, we argue that clustering can alleviate the complexities and potential power overheads associated with fine-grain mixing of frequency/voltage domains. Fields et al. [42] studied how their slack-predictor can be used to save energy in a dual pipeline machine similar to ours. What makes our work different is that we take into account technological restrictions such as those associated with memory references and also power-overheads.

4.4. Results

In section 4.4.1, we study the performance of our power-aware dual-cluster organization relatively to a set of four other obvious alternatives. We will refer to our organization as AFC. In section 4.4.2 we study energy

4.4.1. Performance

Often times, to obtain higher performance disproportion ally large investments in
area and power were made in modern processor designs. Consequently, reducing power considerably may be possible by simply reversing these decisions. To determine whether our methods are indeed worthwhile, it is important to compare them with a number of obvious alternatives. We focus on the geometry of the instruction scheduler and compare with a number of obvious alternatives that vary instruction scheduler width, size and clustering. The scheduler geometry is a strong indicator of back-end power since it dictates the size and width of most all other structures. For example, the physical register file is also affected by the scheduler’s geometry: more physical registers are required for longer schedulers, and more read/write ports are required for wider schedulers.

Overall, our dual cluster architecture has a 96-entry window and can issue 6 instructions per cycle. Accordingly, we compare with four meaningful variations: (1) an identically organized, dual-cluster configuration where both clusters operate at the fast frequency (DUAL-UF), (2) a centralized 96-entry, 6-way issue scheduler (CNTR), (3) using just the fast cluster (64-entry, 4-way scheduler) (FAST-ONLY), and (4) a centralized processor with 4-way issue (same as the fast cluster) but having a scheduler with 96-entries (total number of entries in our configuration) (FAST-ONLY-WIDE). We explain why we included each configuration in our comparisons as we present the results.

Comparing with an identical dual-cluster, uni-speed processor (DUAL-UF):

In figure 4.2(a) we report performance compared to the DUAL-UF architecture. To distribute instructions to clusters, we start with the dependence method [33]. In the dual-frequency configuration, the distribution decisions are selectively overwritten by the criticality prediction mechanism. On the average, our dual-frequency processor’s performance is comparable to the conventional, uni-frequency clustered processor. For some
programs and the CTE method, the AFC architecture performs better. In some cases, assigning less critical instructions to the narrow cluster, reduces contention on the wide cluster. This may speed up the execution of critical instructions and hence improve performance even if the non-critical instructions take twice as much time to complete.

**Comparing with a centralized processor with the same aggregate issue width and depth (CNTR):** Figure 4.2(b) reports performance relative to the CNTR processor. As expected the dual-cluster, dual-frequency processor does not perform as well. On the average, our back-end energy saving methods degrade performance as much as 4.2%, 6%, and 5.7% for CTE, GTG, and YIQ respectively. This degradation is expected as our configuration is clustered and also uses a slow cluster.

**Comparing with using just the fast cluster either as is or with having the same overall scheduler entries:** Figure 4.2(c) reports the performance improvement over the FAST-ONLY-WIDE processor. This is a processor that uses just the fast cluster with the same overall number of scheduler entries. On the average, our AFC processor improves performance by 6.4%, 4.5%, and 4.8% for the CTE, GTG, and YIQ respectively. The performance improvement implies that the extra issue bandwidth provided by the slow cluster pays off. Figure 4.2(d) reports performance improvements over the FAST-ONLY processor. This is a processor that is equivalent to the fast cluster of our AFC configuration On the average, performance improvements exceed 6.5% for all heuristics.

We have shown that, while our AFC configuration degrades performance compared to a centralized processor with the same overall issue-width and instruction window size, this degradation is comparable and often smaller than that observed with an identical, uni-speed dual-cluster processor. We have also shown that our dual-speed processor
improves performance by as much as 8.6% on the average compared to an alternative

Figure 4.2. Relative Performance v.s. (a) DUAL-UF, and (b) CNTR processors. (c) & (d) Variations of the fast cluster only: (c) FAST-ONLY-WIDE, and (d) FAST-ONLY.
organization that uses only the fast cluster.

**Where is performance lost in the AFC processor:** To better understand the relative importance of clustering and the use of a slow cluster, in figure 4.3(a) we report the percentage of instructions assigned to the fast-cluster. On the average, GTG assigns the minimum number of instructions to the fast cluster (82%). YIQ assigns the maximum number of instructions to the fast cluster (90% on the average). Finally, CTE assigns 83% of the instructions to the fast cluster.

![Figure 4.3](image-url)  
*Figure 4.3. Understanding where performance is lost in the AFC processor: (a) The fraction of instructions assigned to the fast cluster. (b) Relative performance for a non-realistic machine which loses performance only due to instructions running slower (no cluster-induced stalls).*

Figure 4.3(b) provides additional insight on the inner-workings of our heuristics. Shown is the slowdown compared to CNTR when the non-critical instructions as identified by each heuristic are artificially forced to run twice as slow. This measurement reveals how effectively our heuristics identify non-critical instructions. Combining the
results of figures 4.2 and 4.3 we can observe that CTE and GTG manage to assign a large number of instructions to the slow cluster while avoiding penalizing many instructions that are critical to performance.

4.4.2. Energy Reduction

Chandraskan et al., have shown that even though the exact determination of voltage reduction under performance constraints is complex and technology specific, it is possible to closely predict it [26]. Their methods predicts that when the frequency is twice as slow, the supply voltage can be reduced even to less than 50%. Accordingly, in figure 4.4 we report energy savings when the slow cluster operates with a voltage supply that is 70% of that used in the fast cluster. We use a 0.7x factor as opposed to 0.5x to pessimistically account for voltage conversion latency overheads. The energy results are shown in figure 4.4. Figure 4.4(a) reports back-end energy reduction compared to the DUAL-UF architecture. This energy includes all structures at the back-end (e.g., register files, schedulers, functional units). It does not include the data cache. Here on average, energy savings are 16.8%, 15.2%, and 14.2% for CTE, GTG, and YIQ respectively. As shown in figure 4.2(a) this energy reduction comes with virtually no performance cost when the CTE method is used. Figure 4.4(b) reports energy reduction compared to the CNTR processor. On the average, energy savings are 34.6%, 33.2%, and 32.6% for CTE, GTG, and YIQ respectively. Figure 4.4(c) reports energy savings compared to the FAST-ONLY-WIDE processor. On the average, we reduce energy by 9.7%, 7.9%, and 7% with CTE, GTG and YIQ. These energy savings are significant if one considers that the AFC configuration also performs slightly better. For ammp, our AFC processor consumes more energy. This is
because *ammp* exhibits very low IPC and very high branch prediction accuracy. Consequently, static power dissipation (estimated at 10% of worst case dynamic power) becomes more important. The AFC has collectively more resources than the FAST-ONLY-WIDE. Finally, figure 4.4(d) reports energy reduction compared to the FAST-ONLY processor. On the average, energy consumption increases by 4.3%, 6.4%, and 7.6% for CTE, GTG, and YIQ respectively. However, we have seen that the AFC processor improves performance. For example, as seen in figure 4.2(b), with the CTE method performance is 8% higher.

As shown, the AFC processor offers considerable energy savings compared to the first two alternatives (DUAL-UF and CNTR). Compared to either alternative, the AFC processor reduces energy requirements by running non-critical instructions in the slower cluster. Compared to the CNTR processor, the AFC architecture also reduces energy by exploiting smaller structures.

### 4.5. Conclusion

We introduced and evaluated an asymmetric dual-cluster, dual-frequency microarchitecture comprising a performance oriented cluster and a power-aware one. Our organization aims: (1) at reducing switching power by executing non-critical instructions slower, and (2) at maintaining performance by executing performance critical instructions as fast as possible. In our organization, non-critical instructions are meant to execute in the power-aware cluster that is narrow and uses a lower frequency and power supply. Performance critical instructions are meant to execute in the performance oriented cluster that is
Figure 4.4. Back-end energy reduction (when the slow cluster uses 0.7 x Vdd) compared to: (a) DUAL-UF processor, (b) CNTR processor (c) FAST-ONLY-WIDE processor (d) FAST-ONLY processor.

wide and uses a higher frequency and voltage supply. By localizing the two frequency/voltage domains, we mitigate many of the complexities associated with maintaining multi-
ple supply voltage and frequency domains on the same chip. Essential to the success of our technique are methods for distributing instructions across the two clusters. We have introduced two new distribution heuristics that are effective and power efficient. We have shown that it is possible to save up to 19% of back-end energy with only a 0.2% performance loss compared to a conventional, dual-clustered processor.
5. Branch Predictor Prediction: A Power-Aware Branch Predictor

5.1. Introduction

Dynamic power dissipation has emerged as a first class consideration in modern, high-performance processor design. In this work, we focus on energy-efficient and highly-accurate branch predictors. Branch predictors dissipate a considerable fraction of overall processor power. It was shown that for a typical high-performance processor, branch predictors dissipate about 10% of overall power [41].

Developing energy-efficient branch predictors is a challenging task due to the way branch prediction impacts overall energy. Specifically, branch prediction impacts energy both directly and indirectly. The predictor itself consumes energy. In isolation, it would seem that using smaller predictors would result in reduced energy consumption. However, this is not true since branch predictor accuracy indirectly impacts the amount of work and hence the amount of energy that is consumed by the rest of the processor [41]. Accordingly, in this work we focus on predictors that are both energy-efficient and highly-accurate.

We propose branch predictor prediction or BPP as an energy-efficient extension to the commonly used combined predictors. Combined predictors use three underlying sub-predictors that are all accessed for every branch. Out of the three prediction hints, one is used to select among the other two. BPP exploits the temporal and sub-predictor locality characteristics of typical branch streams to allow us to gate two out of the three sub-predictors for many branches. Specifically, we have observed that: 1) often, short sequences...
of branches tend to appear repeatedly, and 2) they tend to use the same sub-predictors. In BPP, a small buffer is introduced in the fetch stage. BPP entries record the sub-predictors used by recent branches. Each BPP entry is tagged by the PC of a recently seen branch and records the sub-predictors used by the two branches that followed it in the dynamic execution stream. By associating sub-predictor hints with a preceding branch, BPP avoids increasing prediction latency. The hints become available at least one cycle in advance of when the actual prediction needs to take place.

We show that our predictor can reduce energy by 13% over a 32K-entry, conventional, banked combined predictor (taking into consideration the BTB and the return address stack that our BPP does not optimize). Moreover, we show that a processor with BPP is always more energy efficient than one that uses any of the sub-predictors alone. Finally, we show that BPP reduces energy even for smaller predictors (range 16K-entries down to 1K-entries).

The rest of the paper is organized as follows. In section 5.2 we explain BPP. In section 5.3 we report performance, and energy savings. We report relative energy consumption for both the predictor and the entire processor. We compare our model to three different alternatives. In section 5.4, we study how BPP reacts to changes in predictor size and BPP configuration. In section 5.5 we review related work. Finally, in section 5.6, we summarize our findings.

5.2. Branch Predictor Prediction

Branch prediction is essential to sustaining high performance in modern high-end processors. The combined predictor is one of the most accurate predictors available and is
used in many high-performance designs. Combined predictors use three underlying sub-predictors. Two of the sub-predictors produce predictions for branches. They are typically tuned for different branch behaviors. The third sub-predictor is the selector and it keeps track of which of the two sub-predictors works best per branch. Typical configurations, use bi-modal predictors for one of the sub-predictors and the selector, and a pattern-based predictor like gshare for the last sub-predictor. Bi-modal predictors capture temporal direction biases (e.g., mostly taken) in branch behavior and have short learning times. However, they cannot capture complex repeating branch behaviors that do not exhibit a direction bias. Pattern-based predictors like gshare can successfully capture repeating direction patterns and correlated behaviors across different branches. However, these predictors require longer learning times and have larger storage demands. By using a selector predictor, combined predictors offer the best of both underlying sub-predictors: fast learning and prediction for branches that exhibit temporal bias, and slower but accurate prediction for branches with repeatable direction patterns.

While using three tables (gshare, bimod and selector) makes better prediction possible, it also increases energy consumption. Our analysis shows that for a 32K-entry predictor, the three sub-predictors consume close to 65% of the total predictor energy (the rest is consumed in the BTB and the return address stack).

BPP aims at reducing energy consumption compared to conventional combined predictors while maintaining accuracy virtually intact. One obvious way of reducing energy would be to use one of the sub-predictors. While this obviously reduces branch prediction energy consumption, it eventually results in much higher overall energy consumption because of reduced accuracy and consequent increase in energy wasted down
miss-speculated paths (see section 5.3.2).

BPP relies on typical program behavior, to gate two out of the three underlying sub-predictors for most branches. In particular, BPP exploits the following two phenomena: (1) Branch instructions show strong temporal locality. That is, looking over short periods of time, there is a small set of branches that account for the vast majority of predictions. Figure 5.1(a) quantifies the temporal locality in the dynamic branch stream by reporting how often a branch is within the last $n$ branches fetched, where $n$ varies from 8 to 64. On average, almost half of the branches appear within eight branches apart and about 83% of the branches appear within the last 64 branches fetched. (2) Often branches tend to use the same sub-predictor. Figure 5.1(b) shows how often two subsequent instances of the same branch use the same sub-predictor. On the average, a branch uses the same sub-predictor with a 91% probability.

![Figure 5.1(a) Temporal locality among branch instructions.](image-a)

![Figure 5.1(b) How often do branches use the same sub-predictor to predict the branch outcome.](image-b)
Based on the above we suggest BPP. At the core of BPP we use a FIFO buffer called the BPP-buffer. BPP-buffer is a small energy-efficient structure that stores information regarding the most recent branches encountered. The stored information is used to gate the selector and one of the two sub-predictors. This \( n \)-entry FIFO buffer records the \( n \) most recently fetched branches in sequence. Every buffer entry includes an address field and a two bit sub-predictor hint field. We use the latter to record the last sub-predictor used by the specific branch. Combinations 00 and 01 indicate that the gshare and bimodal predictors have been the last predictors used to speculate the branch outcome respectively. We assign combination 11 to indicate that the branch was miss-predicted last time it was speculated. We do not gate the sub-predictors if the miss-predicted branch reappears since we have no confidence on the predictor used last time.

The front-end fetches up to two branches every cycle for a total of eight instructions. Once all instructions are fetched we check the BPP buffer to see if the last branch fetched is among those recorded in the buffer. If no match is found, all sub-predictors are probed during the next cycle. If a match is found, we look at the sub-predictor hints of the next two buffer elements. Since branch sequences tend to repeat, we implicitly predict (assume) that these access hints are the right ones for the next two branches that may be fetched during the next cycle. We gate two of the three tables if the next (guessed by the BPP-buffer) dynamic branches have used (and therefore are predicted to use) the same sub-predictor and they have not been miss-predicted. Consequently, the only case where we gate two of the sub-predictors is when both hints are equal and not 11.

We access the BPP-buffer in parallel with the branch predictor. This is true for both branch lookup and update. We allocate BPP entries at fetch in order. We update the BPP-
buffer speculatively as soon as a branch calculates its direction. We do not flush the BPP entries that follow a miss-predicted branch. We instead mark the miss-predicted branch. This allows us to salvage sub-predictor locality down miss-predicted paths. For example, due to control independence some of the sub-predictor hints may be valid even though an earlier branch was miss-predicted. We lookup the BPP buffer to decide what sub-predictor to gate (if any) for the next cycle. While BPP uses branch temporal locality it is different from a branch predictor cache of shorts since it associates sub-predictor hints with preceding branches.

Provided that sufficient branch and sub-predictor locality exists, BPP has the potential for reducing branch prediction energy consumption. However, it introduces extra energy overhead and can, in principle, increase overall energy consumption if the necessary behavior is not there. We take into account this overhead in our study and show that for the programs we studied BPP is robust.

5.3. Results

In this section, we present our analysis of the BPP technique. We report performance results in section 5.3.1. We report energy measurements in section 5.3.2. While our study shows that power (energy per cycle) results follow the same trend of energy results, we do not report power in the interest of space.

For most of our experiments we used a combined predictor with 32K-entries per sub-predictor. In section 5.4.1, we demonstrate that BPP is still effective even with smaller predictors. We use the 32K-entry predictor after investigating predictors of different sizes studying performance and accuracy. Many metrics for summarizing energy vs. perfor-
mance trade offs exist. For example, one such metric is the energy, delay product. Since our primary goal was to maintain performance we choose instead on predictors that offer performance within 2% of the best predictor we studied. Figures 5.2(a) and 5.2(b) show how predictor size impacts performance and predictor energy consumption relatively to the same processor that uses a 64k-entry predictor. As shown performance-wise most benchmarks show no performance slowdown when a 32k predictor is used. Exceptions are gcc (2% slowdown) and prs (0.6% slowdown). The average slowdown stays within 2% even with the 16K-entry predictor. However, worst case slowdown with this predictor is 5% and much worse for predictors of smaller sizes. In figure 5.2(b) we report predictor energy consumption. Here relative energy consumption is 22%, 25%, 30%, 37%, 50% and 69% for 1k, 2k, 4k, 8k, 16k and 32k predictors when compared to a 64k predictor. Parikh et. al, showed that while using smaller predictors saves predictor energy, it also results in higher overall processor energy consumption[41].

5.3.1 Performance

BPP can negatively impact accuracy and hence performance. Accordingly, we first investigate how BPP impacts performance. To determine whether our technique is indeed worthwhile, it is important to compare it with a number of obvious alternatives. Accordingly, we compare with three alternative branch predictor organizations. Accordingly, we compare with three alternative branch predictors: (1) **Conventional Combined Predictor (CMB)** (2) **Bimodal-Predictor (BMD)** (3) **Gshare-Predictor (GSH)**. Comparing to CMB shows whether our energy savings are worth the possible performance loss, BMD and GSH comparisons explore the possible
In figure 5.3(a) we report performance for the processor that uses a 32-entry BPP (in section 5.4.2 we vary BPP size) compared to 3 different base cases. Bars from left to right report relative performance compared to CMB, BMD and GSH. Numbers lower than 100% represent slowdowns. On average, performance slowdown is 0.3% compared to CMB. In the worst case of mcf, it is only 1%. As it can be seen in table, mcf exhibits the worst branch behavior but seems to not be so sensitive to predictor size (figure 5.2). Accordingly, we expect that often branches change sub-predictors in this program impacting BPP accuracy. However, even then, the performance loss is only 1%, making BPP an
attractive alternative to a conventional combined predictor (CMB). Comparing to the BMD and GSH on average, BPP offers higher performance (7.1% and 3.1% respectively). The performance difference are amplified for some of the integer benchmarks (e.g., gcc, wlf and vpr), while they are negligible for most of the floating point benchmarks (e.g., mes and amm).

The results of figure 5.3(a) corroborate that for well behaved benchmarks a simple branch predictor may be a better choice. However, there are programs the benefit from more elaborate prediction schemes (e.g., gcc and equ). It is for this reason that combined predictors are used in virtually all modern, high-performance processors.

![Figure 5.3](image_url)

*Figure 5.3.(a) Performance (higher is better). Bars from left to right compare BPP to the CMB, BMD and GSH. (b) Percentage of cycles that two of the three tables are gated.*
5.3.2. Energy

As a first indicator of energy reduction in figure 5.3(b) we report how often BPP gates two out of the three sub-predictors per benchmark. As shown, on the average, we gate sub-predictors 54% of time. The maximum of 96% is observed for mes and the minimum of 34% for equ. This is consistent with figure 5.1(b) where equ and mes show minimum and maximum predictability in accessing sub-predictors respectively.

In the rest of the section we first report predictor energy consumption. Later, we also study overall energy consumption since it is very sensitive to branch prediction accuracy. Therefore, while exploiting a less complex branch predictor reduces predictor’s energy consumption, it may increase the total energy consumption. In our experiments we take into account the energy overhead associated with the BPP buffer. Moreover, we also study how BPP interacts with banked predictors (suggested by previous work[41]). Banking is a reasonable choice and particularly important for large predictors. We used four banks based on CACTI’s analysis. We make the following assumptions to pessimistically account for energy overheads and static energy: First, we assume that gated structures still consume 10% of their maximum energy consumption. Second, in banked predictors we assume that all banks are always precharged (i.e., banks need to be ready in case they are accessed by the next branch). It would be possible to use BPP to selectively precharge only the sub-predictor banks that we will use.

Figure 5.4 reports relative branch predictor energy consumption for non-banked (part (a)) and banked (part (b)) predictors. For each benchmark, bars from left to right show results when compared to CMB, BMD and GSH respectively. In figure 5.4(a) we
report relative energy consumption for non-banked predictors compared to a non-banked BPP. Compared to the CMB machine, on average, BPP reduces branch prediction energy consumption by 22% (average relative energy consumption is 78%). Compared to the BMD and GSH machine, as expected, our branch predictor consumes more energy. On average, BPP consumes 53% and 34% more energy compared to the BMD and GSH respectively. However, we later show that since BPP is more accurate, it reduces overall processor energy. In figure 5.4(b) we report relative energy consumption for banked predictors compared to a banked BPP. As expected, energy savings are lower but still considerable. When compared to CMB, BPP reduces energy 13% (average relative energy consumption is 87%). BPP consumes 35% and 22% more than BMD and GSH respectively.

In figure 5.5 we report relative total energy consumption. Again bars from left to right show results when compared to CMB, BMD and GSH respectively. In figure 5.5(a) we study processors with non-banked predictors. When compared to the CMB, BMD and GSH relative energy reduction is 1.7%, 5% and 1.7% (average relative energy consumptions are 98.3%, 95% and 98.3%). In figure 5.5(b) we study processors with banked predictors. Here when compared to the CMB, BMD and GSH relative energy reduction is 0.4%, 4.6% and 1.2%. respectively (average relative energy consumptions are 99.6%, 95.4% and 98.8%). This suggests that BPP reduces both the predictor’s energy and the total processor energy consumption. Moreover it shows that using a combined predictor both increases performance and saves energy when compared to using only one of its sub-predictors.
Figure 5.4. Relative branch prediction energy consumption for (a) non-banked BPP and (b) banked BPP. Bars from left to right compare to CMB, BMD and GSH.

Figure 5.5. Relative total energy consumption for a processor using a (a) non-banked BPP and (b) banked BPP. Bars from left to right compare to CMB, BMD and GSH machine.
5.4. Sensitivity Analysis

In this section, we study how BPP reacts to smaller predictor sizes and different BPP buffer sizes.

5.4.1. Sensitivity to Predictor Size

Here we present how BPP performs when smaller branch predictors are used. We report relative energy consumption compared to a non-banked CMB machine with a similar predictor size. We focus on non-banked predictors since banking is reportedly less effective for smaller predictors [41]. Figure 5.6 reports relative energy consumption for different predictor sizes. As expected, savings are less for smaller predictor sizes. Still, on average we save 4% even for a 1k predictor. Our average savings are 7.5%, 13%, 16% and 21% for predictor sizes 2k, 4k, 8k and 16k. In all results we assume a 32 BPP buffer size.

![Figure 5.6. Relative energy consumption compared to non-banked CMB and for different predictor sizes.](image-url)

5.4.2. Sensitivity to Buffer Size

We study how the number of BPP entries impacts energy consumption. We only report relative energy consumption. While our results shows that performance stays within 0.5% for all buffer sizes we do not report detailed performance results due to space limitations. We limit our attention to comparing with a conventional, banked combined predic-
Figure 5.7 reports the relative energy consumption when BPP is in use. On average, energy reduction is 11.3%, 13%, 10.4% and 8.7% for buffer sizes 64, 32, 16 and 8 respectively. As shown, savings are maximum for a 32-entry buffer. Apparently, a 32-entry buffer is large enough to store the required information. While 8- and 16-entry buffers appear to be too small, a 64-entry buffer imposes unnecessary energy overhead. In all results we assume a 32k predictor.

![Figure 5.7. Relative energy consumption for different buffer sizes. Bars from left to right report for buffers sizes 64, 32, 16 and 8.](image)

5.5. Related work

Previous work has introduced banking and the usage of prediction probe detector (PPD) as two techniques that reduce power dissipation without harming accuracy [41]. Banking reduces the active portion of the predictor. In this paper we studied how BPP interacts with banking and demonstrated that BPP can reduce energy even when the underlying predictors are banked. PPD uses pre-decoded bits (stored in the PDD table) to eliminate unnecessary BTB and predictor accesses. It aims at identifying 1) when a cache line has no conditional branches to avoid a lookup in the direction predictor and 2) when a cache line has no control-flow instructions at all so the BTB lookup could be eliminated. The PDD table stores the required information. While BPP does not target reducing the
BTB energy consumption it could be used on top of PDD to further reduce energy consumption by gating the sub-predictors when there are predictor lookups required.

5.6. Conclusion

We presented BPP, a technique for reducing energy while maintaining the accuracy advantage of combined branch predictors. We affirmed that it is possible to significantly reduce energy by exploiting a) the temporal locality amongst branch instructions, and b) the high predictability in their usage of sub-predictors. BPP reduces energy consumption up to a maximum of 31% (average of 22%) compared to a non-banked and up to 21% (average of 13%) compared to a banked conventional combined predictor. This comes with a negligible performance degradation. Our study covered a subset of both integer and floating point SPEC 2000 benchmarks. We have also shown that while BPP is more effective for larger predictors, it still reduces energy even for smaller branch predictors. More importantly, we have shown that when one considers the overall processor energy consumption, BPP-enhanced processors always dissipate less energy when compared to ones that use either just the conventional combined predictor, or just one of its underlying sub-predictors.

Because of the considerable energy savings and the relatively small cost, BPP is an attractive power-aware enhancement for modern, high-performance processors.
6. Conclusion

In this thesis we introduced and investigated performance and power optimization techniques for high performance processors.

As a performance optimization technique, we investigated clustering as a potentially viable solution to wide and deep pipelines. We studied a variety of instruction distribution methods, including adaptive and non-adaptive techniques, for quad-cluster processors. Our methods utilized various types of information, including instruction-type, dependences and past history to better distribute instructions across clusters.

Our study show that a non-adaptive dependence-based method, DEP offers performance within 8% of a non-clustered organization operating at the same frequency. Also, we found that it is possible to reduce this gap down to about 7.1% via a counter-based prediction scheme. We studied the sensitivity of our methods to inter-cluster communication latency, front-end pipeline depth and the number of cluster input ports. We found performance more sensitive to inter-cluster communication for the better performing methods.

While we studied a reasonable set of configurations and methods, there is still a plethora of design points and possible other methods that warrant further study. There are multiple directions for further experimentation, including non-uniform cluster organizations, the effect of previously proposed compiler optimizations [6] and alternative scheduler designs such as those appearing in [12]. Of particular interest are organizations where execution clusters (i.e., functional units, register files and cache ports) and schedulers are decoupled. In such a design, an instruction is first assigned to a scheduler, and then, based on input operand availability is sent to the appropriate execution clustered.
We also introduced three power optimization techniques. We introduced techniques to reduce power dissipation at processor’s front-end, processor’s back-end and at the branch predictor.

We extended previous work in the area of power-aware front-end throttling by introducing flow-based front-end gating techniques. Moreover, we evaluated these methods using a power model of a modern high-performance processor. We showed that 12.2% of the energy consumed could be saved by exploiting an intelligent method that combines confidence-based and the recently proposed flow-based front-end throttling methods. This considerable reduction in energy is possible with a minor 1.9% performance loss. Moreover, we studied how front-end throttling techniques affect power consumption in different processor sections and structures. As expected, structures interacting mostly with front-end instructions benefit more from the savings than those in the issue- and complete-logic.

As a power-aware processor back-end, we introduced and evaluated an asymmetric dual-cluster, dual-frequency microarchitecture comprising a performance oriented cluster and a power-aware one. We reduced switching power by executing non-critical instructions slower while maintaining performance by executing performance critical instructions as fast as possible. By localizing the two frequency/voltage domains, we mitigate many of the complexities associated with maintaining multiple supply voltage and frequency domains on the same chip. We also introduced efficient methods for distributing instructions across the two clusters. We have shown that it is possible to save up to 19% of back-end energy with only a 0.2% performance loss compared to a conventional, dual-clustered processor.
Finally as a power-aware branch prediction technique, we presented BPP. We showed that it is possible to significantly reduce energy by exploiting the temporal locality amongst branch instructions, and the high predictability in their usage of sub-predictors. Our power-aware branch predictor reduces energy consumption up to a maximum of 31% (average of 22%) compared to a combined predictor. This comes with a negligible performance degradation. We have also shown that BPP reduces energy across a range of branch predictor sizes. More importantly, we have shown that when one considers the overall processor energy consumption, BPP-enhanced processors always dissipate less energy when compared to ones that use either just the conventional combined predictor, or just one of its underlying sub-predictors.

There are several possible future extensions to our power optimization techniques. As our preliminary studies show, front-end power optimization techniques cluster instructions making more efficient usage of processor bandwidth possible. Potentials of such behavior may lead to further power savings.

As an extension to processor back-end power optimization techniques, we suggest investigating different processor configurations. Of particular interest is studying clustered architectures with wider power-aware clusters. Another possible future study could be reconfigurable power-aware architectures where the number of active power-war resources could be decided dynamically.

Finally, there are possible future extensions to power-aware branch predictors. While other types of information (e.g., based on different types of branches) can be used in improving predictor prediction techniques, our preliminary studies show potential for further improvements by using the BPP buffer more efficient (e.g., avoid storing redun-
dant information).
References


