© 2009 Brian L. Greskamp
IMPROVING PER-THREAD PERFORMANCE
ON CMPS THROUGH TIMING SPECULATION

BY
BRIAN L. GRESKAMP
B.S. Clemson University, 2003
M.S. University of Illinois at Urbana-Champaign, 2005

DISSERTATION
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:
Professor Josep Torrellas, Chair
Shekhar Borkar, Intel Corporation
Assistant Professor Deming Chen
Professor Sanjay Patel
Assistant Professor Craig Zilles
Abstract

The future of performance scaling lies in massively parallel workloads, but less-parallel applications will remain important. Unfortunately, future process technologies and core microarchitectures no longer promise major per-thread performance improvements, so microarchitects must find new ways to address a growing per-thread performance deficit. Moreover, they must do so without sacrificing parallel throughput. To meet these apparently conflicting demands, this dissertation proposes a Timing Speculation (TS) system for multicores that boosts core clock frequencies past their normal limits when an application demands per-thread performance and operates efficiently at nominal frequency when it demands throughput. This work’s contributions are organized into three interlocking proposals.

This work begins by introducing *Paceline*, the first TS microarchitecture designed specifically for multicores. Paceline enables two cores to work together to execute a single thread at high speed under TS or independently to execute two threads at the rated frequency. In single-thread mode, one core in the pair — the “Leader” — executes at higher-than-normal frequency, while a “Checker” runs at the rated, safe frequency. The Leader runs the program faster but may experience timing errors. To detect and correct these errors, the Checker periodically compares a hash of its architectural state with that of the Leader. The Leader helps the Checker keep up by passing it branch results and prefetches.

Next, this dissertation enhances Paceline with *BlueShift*, a circuit design method for TS architectures that improves a circuit’s common-case delay rather than focusing on worst-case delay like traditional design flows. BlueShift profiles a gate-level design as it runs real benchmark applications to identify the frequently-exercised circuit paths and then applies speed optimizations.
to those paths only. These optimizations can be implemented in a way that can be enabled and
disabled at run-time so that they do not exact a power cost when they are not needed (i.e. when the
processor is executing a throughput workload).

Finally, this work presents LeadOut, a multicore design that combines Paceline with an ad-
ditional per-thread performance enhancement: the ability to increase core supply voltage above
nominal. LeadOut evaluates the performance gains that are possible with Paceline alone, voltage
boosting alone, and both together. It shows major gains from applying the two techniques together
when feasible and also shows that, in many cases, future multicores have power and tempera-
ture headroom to exploit still more per-thread enhancements as long as they can be enabled and
disabled dynamically according to application demand.
To my parents, Mike and Debbie Greskamp.
Acknowledgements

My parents, Mike and Debbie, have been most instrumental in this dissertation. Who else would have bought a fourth-grader thousands of dollars worth of electronic components and test equipment to encourage a burgeoning interest in electronics design? Their continued support and encouragement ultimately gave me the audacity to attempt the Ph.D, and their counsel throughout kept me in mental health to see it through.

For sparking my interest in electronics, I’d like to thank Wayne Weise, my first great mentor. Thanks, Wayne, for taking the time to put together lessons and kits for a student who was only beginning to understand. For introducing me to the joys (and trials) of research and pointing me toward graduate school, I thank Prof. Ron Sass, who welcomed me into his research group as a still-naïve undergraduate.

Next, I owe a great debt to my peer–mentors: Smruti R. Sarangi, Abhishek Tiwari, James Tuck, Radu Teodorescu, Luis Ceze, Karin Strauss, Wonsun Ahn, and Pablo Montesinos. Thanks for your critiques and insights, but above all else, thanks for your example. Pablo deserves special thanks for sharing his experiences, hints, and encouragement during an extremely difficult job search. Thanks also to the rest of the i-acoma group — past and present — for the discussion (whether research-related or not) and the occasional levity.

All students owe a large credit to their advisors, but I owe even more than most. Above all else, Prof. Torrellas taught me to stay focused and to develop a mental toughness that helps me remain motivated even after repeated setbacks. He also taught me to work analytically — to systematically exhaust a problem and to distill insights instead of getting lost in the complexities of implementation. In none of these lessons have I succeeded completely, but under his guidance,
I have progressed.

As for this document itself, the work is not solely my own. For their significant contributions, I thank my co-authors: Lu Wan, Jeffrey J. Cook, Ulya Karpuzcu, Prof. Deming Chen, and Prof. Craig Zilles. They provided invaluable assistance at all levels: developing the seminal BlueShift ideas (Jeffrey, Craig, Lu, and Deming); performing implementation (Lu), and tirelessly questioning and verifying my methods and assumptions (Ulya).

Of course, graduate school isn’t all about academics, and I’d like to thank my room-mates and friends, Lee Baugh and Kiran Lakkaraju, for putting up with my idiosyncrasies and providing a relaxing and fulfilling “home” life away from the lab. I’d also like to thank my friend and colleague, Ulya Karpuzcu, for her continued support — both academic and personal.
# Table of Contents

List of Tables  
List of Figures  

## Chapter 1 Introduction  
1.1 The Per-Thread Performance Crisis  
1.2 Barriers to Per-Thread Performance  
  1.2.1 Technology Barriers  
  1.2.2 Core Microarchitecture Barriers  
1.3 A Case for Configurable Multicores  
1.4 Timing Speculation  
1.5 Contributions and Overview  

## Chapter 2 Paceline: Timing Speculation for Multicores  
2.1 Introduction  
2.2 Characterizing Overclockability  
  2.2.1 Grading Artifacts  
  2.2.2 Safety Margins  
  2.2.3 Error Tolerance  
  2.2.4 Exploiting Overclockability  
2.3 Paceline Architecture  
  2.3.1 Overview of the Microarchitecture  
  2.3.2 Types of Errors  
  2.3.3 Detailed Microarchitecture  
  2.3.4 Additional Issues  
  2.3.5 Implementation Feasibility  
2.4 Analytical Performance Model  
2.5 Summary  

## Chapter 3 BlueShift: Designing Pipelines for Timing Speculation  
3.1 Introduction  
3.2 Taxonomy of Design for TS  
  3.2.1 Classification of TS Microarchitectures  
  3.2.2 General Approaches to Enhance TS  
  3.2.3 Putting It All Together  

vii
### Chapter 3.3 A Common-Case Optimization Flow

- **3.3.1 The BlueShift Framework** .......................... 36
- **3.3.2 Example BlueShift Techniques** .................. 40

### Chapter 4 LeadOut: Combining Timing Speculation with V–f Boosting

- **4.1 Introduction** ........................................... 45
  - **4.1.1 V–f Boosting** ..................................... 46
- **4.2 V–f Boosting and TS are Synergistic** .............. 47
  - **4.2.1 Power Increase in VBoost and Paceline** ... 47
  - **4.2.2 Composing the Techniques** .................... 48
  - **4.2.3 A Highly-Configurable Multicore** .......... 51
- **4.3 Dynamic Controller Design** .......................... 51
  - **4.3.1 Problem Formulation** ............................ 52
  - **4.3.2 Thread Controllers** ............................... 53
  - **4.3.3 Global Controller** ................................. 55
- **4.4 Summary** ............................................... 55

### Chapter 5 Paceline Evaluation

- **5.1 Experimental Setup** ................................ 57
- **5.2 Results** .................................................. 57
  - **5.2.1 Performance** ....................................... 57
  - **5.2.2 Dynamic Power** ................................... 60
  - **5.2.3 Power Density** ..................................... 62
  - **5.2.4 Sensitivity to Errors** ............................. 63

### Chapter 6 BlueShift Evaluation

- **6.1 Experimental Setup** ................................ 66
  - **6.1.1 Modeling Overview** ............................. 67
  - **6.1.2 Technology Model** ................................. 68
  - **6.1.3 BlueShifted Module Implementation** ....... 69
  - **6.1.4 Module-Level P_E and Power** ................ 71
  - **6.1.5 Microarchitecture-Level P_E and Power** ... 72
- **6.2 Results** .................................................. 74
  - **6.2.1 Error Curve Transformations** ................ 74
  - **6.2.2 Paceline+OSB Performance and Power** .... 76
  - **6.2.3 Razor+PCT Performance and Power** ....... 78
  - **6.2.4 Computational Overhead** ....................... 80

### Chapter 7 LeadOut Evaluation

- **7.1 Experimental Setup** ................................ 81
- **7.2 Results** .................................................. 83
  - **7.2.1 Performance** ....................................... 83
  - **7.2.2 Power–Performance Tradeoffs** ............... 87
  - **7.2.3 Sensitivity Analysis** ............................. 88
## List of Tables

1.1 Microprocessor evolution over the past 15 years. ........................................... 3

2.1 Types of errors considered. .......................................................... 19
2.2 Estimated versus measured Paceline speedups. ........................................ 28

3.1 Classification of existing proposals of TS microarchitectures. ...................... 33
3.2 How TS microarchitectural choices impact what TS-enhancing approaches are most appropriate. .......................................................... 35

4.1 Multicore constraint regimes. ......................................................... 50

5.1 Microarchitecture parameters. ......................................................... 58

6.1 BlueShift TS microarchitecture parameters. .............................................. 67
6.2 Technology model. ................................................................. 69
6.3 BlueShift parameters. ............................................................... 70
6.4 OpenSPARC modules used to evaluate BlueShift. ..................................... 70
6.5 Static power consumption ($P_{sta}$) and switching energy per cycle ($E_{dyn}$) for each module implementation in 130nm. .................................................. 72

7.1 Variation parameters. ................................................................. 83
7.2 Power/thermal environments for the sensitivity study. ................................. 90
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Top SPECint92 score by year. Dotted trend-line shows pre-2003 scaling. Solid line estimates future scaling.</td>
</tr>
<tr>
<td>1.2</td>
<td>A homogeneous multicore without reconfigurability (a) gives constant per-thread performance regardless of system loading. A configurable homogeneous multicore (b) provides enhancements to improve per-thread performance when few threads are running.</td>
</tr>
<tr>
<td>1.3</td>
<td>Error rate (a) and performance (b) versus frequency under TS.</td>
</tr>
<tr>
<td>2.1</td>
<td>Qualitative comparison of the potential of the three architectural approaches to exploit over-clocking.</td>
</tr>
<tr>
<td>2.2</td>
<td>Paceline multicore with 16 cores. Added components are highlighted in red.</td>
</tr>
<tr>
<td>2.3</td>
<td>The Paceline microarchitecture.</td>
</tr>
<tr>
<td>2.4</td>
<td>Speedup traces for <code>parser</code> (first row) and <code>ammp</code> (second row) with a leader over-clocking factor of 1.3.</td>
</tr>
<tr>
<td>3.1</td>
<td>General approaches to enhance TS by reshaping the $P_E(f)$ curve. Each approach shows the curve before reshaping (in dashes) and after (solid), and the operating point of a processor before (a) and after (b).</td>
</tr>
<tr>
<td>3.2</td>
<td>Circuit annotated with net transition times, showing two overshooting paths for this cycle.</td>
</tr>
<tr>
<td>3.3</td>
<td>The BlueShift optimization flow.</td>
</tr>
<tr>
<td>3.4</td>
<td>On-demand Selective Biasing (OSB): application to a chip (a) and pseudo code of the algorithm (b).</td>
</tr>
<tr>
<td>3.5</td>
<td>Transforming a circuit to reduce the delay of $A \rightarrow Z$ at the expense of that of the other paths. The numbers represent the gate size.</td>
</tr>
<tr>
<td>4.1</td>
<td>Qualitative depiction of increases in processor $f$ when the VBoost and Paceline techniques are applied.</td>
</tr>
<tr>
<td>4.2</td>
<td>Thread controller examples.</td>
</tr>
<tr>
<td>4.3</td>
<td>Global controller.</td>
</tr>
<tr>
<td>5.1</td>
<td>Application speedups with overclocking factors $(oc)$ from 1.1–1.4 without a BQ.</td>
</tr>
<tr>
<td>5.2</td>
<td>Application speedups with overclocking factors $(oc)$ from 1.1–1.4 with a BQ.</td>
</tr>
</tbody>
</table>
5.3 Dynamic power breakdown in two base cores sharing an L2 (leftmost bar $U$ in each group) and in Paceline mode for overclocking factors of 1.1 – 1.3 (right three bars in each group). .......................................................... 61

5.4 Error rate and geometric mean of the Paceline speedup in SPECint applications as the overclocking factor changes. .......................................................... 64

6.1 Whole-pipeline $P_E(f)$ curves for the four implementations. The frequencies are given relative to the Rated Frequency ($f_r$) of Paceline Base. .............................. 74

6.2 Performance (a) and power consumption (b) of Paceline before (Paceline Base) and after (Paceline+OSB) BlueShift. BS and NonBS refer to BlueShiftable and non-BlueShiftable modules, respectively. ........................................ 77

6.3 Performance (a) and power consumption (b) of Razor before (Razor Base) and after (Paceline+OSB) BlueShift. BS and NonBS refer to BlueShiftable and non-BlueShiftable modules, respectively. ........................................ 79

7.1 Performance improvements relative to Unoptimized at various multicore loading levels. .......................................................... 84

7.2 Relative incidence of limiting constraints across all applications and die samples. The numbers above the bars show the average power consumption of the enhanced thread on each system. .......................................................... 86

7.3 Per-thread power consumption vs performance for each technique under different load conditions. .......................................................... 88

7.4 Sensitivity of performance improvements to the thermal environment (a), process guardband (b), and power grid design (c). .......................................................... 89

7.5 Percentages of performance improvements experienced by an $S$ thread when the application demands different numbers of $R$ and $S$ threads. For each technique, there is an area of infeasible configurations. Iso-performance contours are shown in the feasible regions. .......................................................... 92
Chapter 1

Introduction

1.1 The Per-Thread Performance Crisis

Semiconductor process technology scaling has driven microprocessor performance for the past thirty years. Until very recently, each new technology generation brought exponentially more and exponentially faster transistors, which microarchitects put to innovative use in building ever-larger and faster processor cores. These designs furnished predictable, exponential performance gains on existing single-threaded compute intensive applications. For example, Figure 1.1 shows the scaling of SPECint performance since 1993, where each data point is the highest score reported in that quarter. As the dashed trendline shows, SPECint (and essentially all compute-intensive applications) enjoyed aggressive performance improvements until about 2003, when technology scaling and microarchitecture innovation began to falter. Since then, per-thread performance scaling (solid trendline) has fallen far — and probably permanently — behind the historical rate. The grand challenge for the computing industry today is to find and implement an alternative method of scaling application performance that matches the aggressive historical trend.

As of 2009, the processor industry’s near-unanimous answer is a many-core roadmap that provides exponentially more cores at each technology generation with relatively little increase in per-thread performance. To keep up with this hardware scaling model, software implementations must also scale to ever increasing levels of parallelism. While many important applications (eg. enterprise workloads and visual computing) scale readily to large numbers of cores, others do not. Even for problems that are at least in theory scalable, highly-parallel implementations are frequently more costly (ie. effort-intensive) than less-parallel ones, meaning that program
Figure 1.1: Top SPECint92 score by year. Dotted trend-line shows pre-2003 scaling. Solid line estimates future scaling.

scalability will often be limited by practical considerations such as development cost and time-to-market. This dissertation attempts to hedge the risk inherent in the many-core roadmap by proposing simple, low-overhead microarchitecture enhancements that improve per-thread performance on multicores without compromising parallel throughput.

1.2 Barriers to Per-Thread Performance

To address per-thread performance, we must first understand its technology and microarchitecture sources. As a starting point, consider the two microarchitectures shown in Table 1.1 released roughly fifteen years apart and bracketing a period in which SPECint performance improved by 168x. The table shows a roughly 35x improvement in clock rate between the two designs, half of which has come from deeper pipelining and the remainder from the technology (faster gates). Assuming that compiler quality remained constant over the interval, the rest of the 168x improvement (about 4.9x) has come from gains in IPC even as memory latency increased.

1A gross approximation, but for integer applications, the impact of the compiler is expected to be far less than that of the process and microarchitecture.
### Table 1.1: Microprocessor evolution over the past 15 years.

<table>
<thead>
<tr>
<th></th>
<th>Intel Pentium</th>
<th>Intel Core i7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year Introduced</td>
<td>1994</td>
<td>2009</td>
</tr>
<tr>
<td>Process</td>
<td>600 nm</td>
<td>45 nm</td>
</tr>
<tr>
<td>Die Size</td>
<td>147 mm²</td>
<td>263 mm²</td>
</tr>
<tr>
<td># Cores</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td># Transistors</td>
<td>3.2 M</td>
<td>731 M</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>100 MHz</td>
<td>3.46 GHz (w/Turbo)</td>
</tr>
<tr>
<td>FO4 per stage</td>
<td>46</td>
<td>23 (est.)</td>
</tr>
<tr>
<td>SPECint92</td>
<td>130</td>
<td>21.8 K</td>
</tr>
<tr>
<td>Total Power</td>
<td>11 W</td>
<td>130 W</td>
</tr>
</tbody>
</table>

Together, the 2x deeper pipelining\(^2\) and 4.9x higher IPC mean that microarchitecture innovation has contributed a roughly 10x performance improvement over the last fifteen years. The remaining 17x is due to technology advancement. The following subsections examine the technology and microarchitecture limitations on per-thread performance and explain why, despite this impressive history, further major improvements are unlikely.

#### 1.2.1 Technology Barriers

Classical technology scaling \(^1\)[18] exponentially increases the number and speed of devices on a fixed-size die while maintaining constant power density. For the seven process generations from 1994 (600nm) to 2009 (45nm), classical scaling predicts a 178x increase in transistor density and a 13x increase in device speed. While actual scaling closely matched those projections (128x and 17x, respectively, according to the data in Table 1.1), this was achieved at the cost of a 7x increase in power density — not the constant density classical scaling predicted. The main cause for the discrepancy is that supply voltage did not decrease as rapidly as the classical models require. Even worse, supply voltage scaling has reached an impasse below 1V and is expected to decrease by only 1.6x over the next thirteen years \(^1\). This is even less than the 3x reduction that occurred over the preceding thirteen year period (which saw an explosion in power density).

\(^2\)The Core i7’s per-stage FO4 delay is estimated to be the same as that of the preceding Core 2 microarchitecture \(^1\)[16], which is half that of the Pentium.
Meanwhile, the difficulty of precisely manufacturing nanoscale transistors has given rise to worsening process variation: the deviation of transistor parameters from their nominal specification \[68, 11\]. This variation affects transistor switching delay and leakage power \[41, 45, 57\], with the microarchitectural consequence that some cores on a multicore are slower or leakier than designed. In general, process variation erodes per-thread performance by forcing engineers to design defensively, sacrificing performance and efficiency. For example, it forces device engineers to employ higher threshold voltages than would otherwise be optimal in order to control static power consumption and forces circuit designers to include larger guardbands to accommodate timing uncertainty. Process variation thus greatly complicates the tradeoffs between power, performance, and reliability.

Because of the power density problem, process variation, and a host of other interrelated causes, the International Technology Roadmap for Semiconductors now projects that circuit delay will scale down by only 7% per year \[1\] instead of the historical 17% per year. This means that processor microarchitects can expect at most a 2.6x increase in clock frequency due to technology over the next thirteen years. The actual increase may be far less if designers decide to use the new technology nodes to recover power density that has been lost in the course of scaling instead of pushing clock frequency.

### 1.2.2 Core Microarchitecture Barriers

Although microarchitecture innovation has historically contributed major IPC and frequency gains (nearly 5x and 2x cumulative gain, respectively, since 1994 according to Table \[1.1\]), tightening power constraints severely limit future microarchitectures. Existing approaches for extracting higher IPC involve some combination of wider instruction issue and deeper instruction windows, both of which inherently degrade power efficiency. Wider issue increases power consumption by requiring more heavily-ported structures and forcing operands to travel larger physical distances on die for bypassing. Deeper instruction windows share similar problems at the circuit level and also raise the spectre of aggressive speculation, which is necessary to fill the
window with work but inherently wastes more power on incorrect, extraneous instructions. Deep pipelines (the traditional approach to high-frequency design) are also unattractive due to their higher degree of speculation and the power consumption of the additional pipeline latches themselves.

Heterogeneous designs [40] that contain multiple core types (i.e., a few large, fast cores and many simple, efficient cores) mitigate the power concern somewhat because most of the cores will be idle during the sequential sections in which the large cores are active, freeing more room for microarchitecture innovation on the large cores. However, they incur extra design cost to implement several core types and force a static partitioning of the silicon area between the types. While heterogeneous designs may indeed prevail, this dissertation assumes a simpler homogeneous multicore environment where all of the cores are microarchitecturally identical and of moderate size.

1.3 A Case for Configurable Multicores

The challenge taken up in this dissertation is to boost per-thread performance on future homogeneous multicores where clock frequency and IPC are stagnant. Moreover, this must be done with minimal area and power cost to avoid compromising performance on the highly-parallel applications that drive the new scaling roadmap. These criteria are in apparent contradiction because they seem to demand something (per-thread performance) for nothing (no compromise in peak throughput), but this is not so. The key to resolving the dilemma is to note that workloads do not simultaneously demand both per-thread performance and throughput from a given set of threads. Instead, at any given instant, high per-thread performance is needed only for a subset of threads. For example, parallel applications tend to alternate between highly-parallel phases that demand throughput and less-parallel phases that demand per-thread performance from a few threads. Additionally, multiprogrammed workloads may contain a mix of latency-sensitive (e.g., interactive) and throughput-sensitive programs.
What is really required then is the ability to configure the microarchitecture on-the-fly to match the workload’s current demands (ie. high throughput, per-thread performance, or some combination of the two). This dissertation refers to this ability to dynamically trade-off per-thread performance and throughput (which others have called reconfigurability [35], dynamism [27], or composability [36]) as configurability. In the most general terms, configurability allows the amount of resources (eg. core pipelines and power budget) dedicated to each thread to vary according to the workload demand. A configurable multicore speeds up performance-critical threads by allocating them more than their fair share of resources, for example by executing them on multiple cores employing Thread Level Speculation [39, 67], allowing adjacent execution pipelines to fuse together to achieve wider issue [35], or by raising their core clock rate and supply voltage [34]. During parallel phases, resources are allocated uniformly to all threads and each core executes in its most power-efficient mode.

To see the benefits of configurability more clearly, consider an application that has fewer active threads than the chip has cores. On a traditional (non-configurable) homogeneous multicore, no matter how many of the on-chip cores are idle, the per-thread performance of the active cores is unchanged as shown in Figure 1.2(a). The result is that, for applications with limited parallelism, adding more cores has no impact on performance. A configurable system, on the other hand, is able to reallocate resources from the idle cores and put them to work on the active threads. Figure 1.2(b) shows performance curves for two hypothetical configurable microarchitectures. Note that the configurable systems provide higher performance when few threads are active while matching the throughput of a non-configurable system when the chip is running its maximum number of threads.

An important observation from Chapter 4 is that microarchitectural enhancements for configurability can be composable. In other words, two different configurable enhancements such as those shown in Figure 1.2(b) can combine to give greater performance than either alone. Since the system still retains the ability to use only a subset (or none) of the per-thread enhancements at

---

3Assuming the power and area costs of the configurability enhancements are negligible.
any give time, performance is never worse than a non-configurable system or a configurable system with only a subset of the enhancements. Configurability provides a “knob” that the runtime environment or operating system can turn to trade off performance and power among the threads in the system. The more configurability the system provides, the more useful the knob becomes.

1.4 Timing Speculation

This dissertation proposes novel configurable enhancements that recover some of the clock frequency lost to process variation and guardbands in recent technology generations by employing Timing Speculation (TS). TS safely operates a core above its nominal, rated frequency by detecting and correcting any timing faults. The main contribution of this dissertation is to demonstrate a configurable, low-overhead TS microarchitecture that is composable with other configurable enhancements for near-future multicores.

To understand how TS can increase clock frequency, note that the processor’s rated frequency $f_r$ is conservative because it includes guardbands that guarantee safe operation under worst-case conditions of aging, temperature, supply voltage, and process variation even though the processor rarely if ever operates under such extreme conditions. Under nominal conditions, clock frequency can increase beyond $f_r$ without generating timing faults simply by consuming the guardband. As frequency increases further, it eventually reaches a Limit Frequency $f_0$ at which
the guardband has been fully consumed and beyond which faults begin to occur. As frequency continues to increase, the number of timing faults per cycle $P_E$ grows exponentially \cite{57}. Figure 1.3(a) shows how the rate of timing faults varies with frequency. Without TS, a processor can typically only operate at point (a) or at best, point (b). TS enables operation at point (c).

![Figure 1.3: Error rate (a) and performance (b) versus frequency under TS.](image)

TS results in an overall performance improvement when the gains from increased clock frequency outweigh the execution time overhead of fault recovery. To see how, consider the performance $perf(f)$ of a processor clocked at frequency $f$, in instructions per second:

$$
perf(f) = \frac{f}{CPI_{norc}(f) + CPI_{rc}(f)} = 
\frac{f}{CPI_{norc}(f) \times (1 + P_E(f) \times rp)} = 
\frac{f \times IPC_{norc}(f)}{1 + P_E(f) \times rp}
$$

(1.1)

where, for the average instruction, $CPI_{norc}(f)$ are the cycles taken without considering any time lost to fault recovery, and $CPI_{rc}(f)$ are the number of cycles per instruction lost to recovery from timing errors. In addition, $P_E$ is the probability of error (or error rate), measured in errors per non-recovery cycle. Finally, $rp$ is the microarchitecture’s recovery penalty per error, measured in cycles.

Figure 1.3(b) illustrates the performance tradeoff. The plots show three regions. In Region 1, $f < f_0$, so $P_E$ is zero and $perf$ increases consistently, impeded only by the application’s increasing memory CPI. In Region 2, errors begin to manifest, but $perf$ continues to increase because the
recovery penalty is small enough compared to the frequency gains. Finally, in Region 3, recovery overhead becomes the limiting factor, and perf falls off abruptly as \( f \) increases. The goal of TS is to operate as close as possible to point (c).

1.5 Contributions and Overview

This dissertation makes three interlocking contributions to the design of configurable multicores at the microarchitecture and circuit levels. First, Chapter 2 introduces Paceline, the first TS microarchitecture designed specifically for multicores. Paceline enables two cores to work together to execute a single thread at high speed under TS or independently to execute two threads at the rated frequency. In single-thread mode, one core in the pair — the “Leader” — executes at higher-than-normal frequency, while a “Checker” runs at the rates, safe frequency. The Leader runs the program faster but may experience timing errors. To detect and correct these errors, the Checker periodically compares a hash of its architectural state with that of the Leader. The Leader helps the Checker keep up by passing it branch results and prefetches.

Next, Chapter 3 proposes BlueShift, a circuit design method for TS architectures that improves a circuit’s common-case delay rather than focusing on worst-case delay like traditional design flows. BlueShift profiles a gate-level design as it runs real benchmark applications to identify the frequently-exercised circuit paths and then applies speed optimizations to those paths only. These optimizations can be implemented in a way that can be enabled and disabled at runtime so that they do not exact a power cost when they are not needed (ie. when the processor is executing a throughput workload).

Chapter 4 introduces LeadOut, a multicore design that combines Paceline with an additional per-thread performance enhancement: the ability to increase core supply voltage above nominal. LeadOut evaluates the performance gains that are possible with Paceline alone, voltage boosting alone, and both together. It shows major gains from applying the two techniques together when feasible and also shows that, in many cases, future multicores have power and temperature head-
room to exploit still more per-thread enhancements as long as they can be enabled and disabled
dynamically according to application demand.

Chapters 5 through 7 evaluate the three contributions in turn. Finally, Chapter 8 compares
Paceline, BlueShift, and LeadOut with a the large body of related work, and Chapter 9 concludes.
Chapter 2

Paceline: Timing Speculation for Multicores

2.1 Introduction

This chapter presents Paceline, a configurable TS microarchitecture for multicores. Paceline improves per-thread performance by running a thread redundantly on two cores of a multicore chip, called Leader and Checker. The leader is “overclocked” to a higher-than-rated frequency while still running at the rated, nominal supply voltage. It thus exploits the safety margin for frequency and even experiences occasional timing errors. Meanwhile, the checker core runs at the rated, safe frequency. The leader prefetches data into the L2 cache that it shares with the checker and also passes branch outcomes to the checker. This improves the checker’s IPC and allows it to keep up with the accelerated leader. The result is that the thread executes faster than on a single baseline core.

We envision a Paceline multicore containing multiple of these leader–checker core pairs. Each core pair shares an L2 and includes simple hardware to periodically compare state and to pass branch outcomes. This hardware requires only small design modifications with respect to a contemporary multicore — mostly confined to the interface between the L1 and L2 caches — and can be easily disabled, returning the core pair to the standard multicore operating mode. Our simulation results show that Paceline substantially improves the performance of a thread without significantly increasing the power density or the hardware design complexity of the chip. If the leader can overclock by 30%, Paceline improves SPECint and SPECfp performance by a geometric mean of 21% and 8%, respectively.
2.2 Characterizing Overclockability

Like all TS schemes, Paceline exploits three sources of “overclockability” in the leader core that allow it to operate above the rated frequency $f_r$: grading artifacts arising from the way processors are binned and marked for sale; process and environmental safety margins; and error tolerance at frequencies beyond the safe one. This section characterizes these sources.

2.2.1 Grading Artifacts

After fabrication, each die is tested for functionality and speed. The latter process is called speed binning, where the objective is to assign one of several pre-determined speed grades to each part. For example, a manufacturer might offer 4, 4.5, and 5 GHz speed grades, and will label each part with the highest speed grade at which it can safely and reliably operate under worst case conditions.

The binning process introduces overclockability in two ways. The first arises from the fact that bin frequencies are discrete. For example, under the scheme described above, a part that passes all tests at 4.8 GHz will be placed in the 4.5 GHz bin. As of early 2007, the Intel E6000 series and the AMD Athlon 64 X2 series space their bin frequencies by 7-14% [6, 32]. If we assume a spacing of 10% as an example, a processor can safely run on average 5% faster than its binned frequency specification.

Secondly, binning contributes to overclockability because within-die process variation causes some cores on a given multicore to be faster than others. In current practice, the slowest core dictates the bin for the entire die. While it is possible to bin each core on the die individually, this may not be cost-effective. If multicores continue to be binned according to the slowest core, each die will contain many underrated cores.

To get a feel for this effect, using the process variation model of Teodorescu et al. [57] with $\sigma/\mu = 9\%$ for the threshold voltage, we find that the $\sigma/\mu$ of on-die core frequency is 4%. Monte Carlo simulations show that this corresponds to an average 16% difference in frequency between
the fastest and the slowest cores on a 16-core die. As another example, Humenay et al. \cite{29} estimate a 17% difference in frequency between the fastest and slowest core on a 9-core 45nm die.

### 2.2.2 Safety Margins

Process and environmental margins for device aging, operating temperature, and supply voltage can also be exploited for overclocking. For example, device aging due to Negative Bias Temperature Instability (NBTI) \cite{51} and Hot Carrier Injection (HCI) \cite{71} causes critical path delays to increase over the lifetime of the processor — and this is especially severe in near-future technologies. A typical high-performance processor’s operational lifetime is seven to ten years \cite{4}. According to \cite{51}, the delay increase due to NBTI alone over that period is 8% in 70nm technology, and HCI adds additional slowdown on top of that. Since processors are guaranteed to operate at the rated frequency for the entire design lifetime, significant overclockability exists in fresh processors where aging has not yet run full course.

Processors are typically rated for maximum device junction temperatures in the 85–100°C range (e.g., \cite{31}) even though operating temperatures are often lower. At lower temperatures, transistors are faster, so overclocking is possible. For example, consider a chip where the hottest unit is 20°C below the maximum temperature. In this case, the analysis in \cite{28} and data from \cite{38} show that the safe frequency increases by approximately 5% for 180nm technology.

Finally, while a processor’s off-chip voltage regulator typically has a tight tolerance, on-chip logic is subject to sizable local voltage drops. These drops are due to rapid current changes that cause $L \frac{dI}{dt}$ supply noise and sustained periods of high activity that cause $IR$ drops. Since these drops are difficult to predict at design time, designers assume conservative margins to guarantee safe operation. For example, the IBM POWER4 processor is designed to provide voltages within ±10% of nominal at all points in the worst case \cite{78}. The logic must therefore be designed to operate correctly under a 10% supply voltage droop even though this is not the common case. This fact can offer some limited room to overclock the processor.
2.2.3 Error Tolerance

Timing Speculation can push the core’s clock frequency past the Limit Frequency \( f_0 \) into a regime where it will experience occasional timing errors. The paths within a processor have a variety of delays, and not all paths are exercised with a given input. Consequently, as the frequency climbs past \( f_0 \), error rate increases continuously as more and more paths fail. As long as only a few paths fail, the error rate will be tolerable, and TS can extract performance gains. The slope of the \( P_E \) vs \( f \) curve in the vicinity of \( f_0 \) varies widely from design to design. Some designs exhibit a “critical path wall” where many paths have long delay, and these are antagonistic to TS. Others exhibit a gradual error onset that results in a tolerable error rate even at frequencies far past \( f_0 \).

For example, the Razor project fabricated 180nm in-order Alpha microprocessors and measured the error rates under different frequencies and voltages [17]. It showed the error rate versus voltage for two specimens at 120 MHz and 140 MHz. If the supply voltage is reduced so that the chip begins to experience timing errors at 120 MHz (i.e., all safety margins have been removed), increasing the frequency to 140 MHz yields less than one error per ten thousand cycles for one specimen and one error per million cycles for the other. This corresponds to a 17% frequency improvement in exchange for an error rate of roughly \( 10^{-6} \) – \( 10^{-4} \) per instruction. On the other hand, experiments with the OpenSPARC [69] pipeline in Chapter 6 of this dissertation show a much more rapid error onset, allowing less overclocking.

2.2.4 Exploiting Overclockability

Exploiting the above factors for overclocking requires various levels of microarchitecture support. Removing the Grading Artifacts requires the fewest microarchitecture changes: The manufacturer must speed test each core individually and populate a one-time-programmable, architecturally-visible table with the core frequencies. The OS can then use this table to set operating conditions. We call this solution Fine Grain Binning (FGB).

At the next level of microarchitecture complexity, Timing Error Avoidance (TEA) schemes
(e.g., \[4, 12, 38, 76\]) can estimate the maximum safe frequency dynamically as the processor runs. These techniques either embed additional “canary” critical paths \[76\], which have delays slightly longer than the actual critical paths, or they directly monitor the delay of the existing critical paths \[4, 12, 38\]. Either way, they are able to determine when the clock period is too close to the actual critical path delay. Although in theory TEA can push the clock frequency arbitrarily close to \(f_0\), practice requires that some safety margins be maintained to avoid errors. Consequently, TEA schemes can remove the Grading Artifacts and some but not all of the Safety Margins.

With an additional increase in microarchitecture complexity, we can exploit all three factors for overclocking, including TS, where the processor experiences occasional errors. The required support is an error detection and correction mechanism. Some examples are Razor \[20\], DIVA \[7\], TIMERRTOL \[75\], and X-Pipe \[77\]. In practice, all of these schemes require fairly invasive microarchitectural changes, either by modifying all processor pipeline latches and the corresponding control logic or by adding a specialized checker backend for the processor core. In the next section, we describe the proposed Paceline microarchitecture, which exploits the three factors for overclocking while requiring minimal modifications to the processor cores and caches.

Figure 2.1 compares the factors for overclocking exploited by each microarchitecture approach. The figure shows only qualitative data because the factors described above are hard to quantify and not fully orthogonal.

![Figure 2.1: Qualitative comparison of the potential of the three architectural approaches to exploit overclocking.](image-url)
2.3 Paceline Architecture

Paceline is a leader–checker architecture that improves the performance of a thread by running it redundantly on two cores of a multicore. The leader core is clocked at a frequency higher than nominal, exploiting the three factors for overclocking described in Section 2.2. Meanwhile, the checker core runs at the rated, safe frequency. The leader prefetches data into the L2 cache that it shares with the checker, and also passes branch outcomes to the checker. This improves the checker’s IPC and allows it to keep up with the accelerated leader. The hardware periodically compares the architectural state of the leader and the checker, and is able to detect and recover from errors due to overclocking or other effects. The result is that the thread executes faster than on a single baseline core.

As it executes, the overclocked leader dissipates higher power than a baseline core. Meanwhile, the checker, by leveraging the prefetched data and branch outcomes from the leader, spends less power than a baseline core. To be able to sustain higher than baseline speed without overheating, the two cores periodically alternate the leader position. The result is that the chip’s power density and maximum temperature are not expected to increase substantially over a baseline system. Intuitively, the operation is analogous to a paceline of two bicycle riders where riders take turns to lead. The leader expends more effort while sheltering the other rider.

A Paceline multicore contains multiple of these leader–checker cores as shown in Figure 2.2. Each core pair shares an L2 cache and includes simple hardware to periodically compare state and to pass branch outcomes. This hardware requires only very small core modifications and can be easily disabled, returning the core pair to the standard multicore operating mode.

Overall, a Paceline pair speeds up a thread (of a serial or a parallel program) without significantly increasing multicore power density or hardware design complexity. In the following, we first give an overview of the microarchitecture and characterize the types of errors it can encounter. Then, we present two different Paceline variations, each specialized for handling different types of errors.
2.3.1 Overview of the Microarchitecture

In Paceline, the leader and the checker cores operate at different frequencies, with the checker lagging behind and receiving branch outcomes and memory prefetches into the shared L2 from the leader. Figure 2.3 shows the microarchitecture. The region in the dashed boundary is overclocked, while everything else runs at the rated, safe frequency.

The shaded components are the new hardware modules added in Paceline. Specifically, the outcomes of the leader’s branches are passed to the checker through the Branch Queue (BQ). Moreover, the hardware in both leader and checker takes register checkpoints every $n$ instructions and saves them locally in ECC-protected safe storage. In addition, the hardware hashes the checkpoints into signatures and sends them to the ECC-protected Validation Queue (VQ). As execution continues, the VQ checks for agreement between the hashed register checkpoints of the leader and the checker. The VQ sits in the cache hierarchy between the L1 and L2 caches. The L1 caches operate in write-through mode as in the Pentium 4 [62], so the VQ sees all the memory writes in order. Such capability allows it to provide extra functionality that depends on the types of errors handled. We will see the details in Section 2.3.3.

Although the leader and checker cores redundantly execute the same thread, they do not execute in lock-step. Since they are out-of-order processors, they execute instructions in different orders and even execute different instructions — due to branch misprediction. However, in the absence of errors, their retirement streams are identical.
Given a dynamic write instruction in a program, the leader and the checker will issue the store to the L1 at different times, when each retires the write instruction. The VQ will only allow one of the two stores to propagate to the L2, possibly after performing some validation. For reads, however, there is no such filtering. A load issued by the leader that misses in the L1 is immediately sent to the L2. If and when the checker issues the corresponding load, it also sends a read request to L2. The advantage of this approach is that it does not require any read buffering at all and, therefore, it is easy to support in hardware. However, it may result in the two loads returning different values — if the location being read is modified in between the reads by, for example, a write from another thread, a DMA action, or a fault. Smolens et al. [64] call this problem the Input Incoherence Problem. This approach is also used in Slipstream [53] and Reunion [64].

2.3.2 Types of Errors

To design Paceline, we consider the three potential sources of error shown in Table 2.1: timing errors due to overclocking, errors due to the input incoherence problem, and soft errors. Table 2.1 characterizes them based on (1) whether they re-appear as the code section with the error
is rolled back and re-executed, and (2) what part of the Paceline architecture they can affect.

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>Repeats in Re-Execution?</th>
<th>Can It Affect...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing Error Due to Overclock</td>
<td>Likely</td>
<td>Leader?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Incoherence</td>
<td>Possibly</td>
<td>Yes: Register state Data read into L1 Data written to L1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Soft Error</td>
<td>No</td>
<td>Yes: Register state Data read into L1 Data written to L1</td>
</tr>
</tbody>
</table>

Table 2.1: Types of errors considered.

These errors behave differently. Timing errors are likely to repeat after rollback because the same critical paths are likely to be exercised during re-execution. Input incoherence errors may re-occur when other threads repeatedly update the location that the leader and checker are trying to read [64]. Indeed, one can construct a pathological scenario where, in each re-execution, a third processor updates the location between the leader and checker reads. On the other hand, soft errors do not re-occur.

From the last two columns, we see that both timing and incoherence errors can affect the same parts of Paceline. Specifically, they can affect the state in the leader’s registers, and the state in the leader’s L1 that has been read from memory or been written by the processor. However, they cannot affect the checker. On the other hand, soft errors can affect both the leader and the checker — both the register state and the data written by the processor into the L1.

A final key difference not shown in the table is that soft errors are much less frequent that the other two types. For example, based on data in [57], this dissertation will estimate in Chapter 5 that the expected timing error rate at optimal performance is approximately one per $10^5$ instructions. A similar incidence of incoherence errors is shown in the environment considered by Reunion [64]. On the other hand, the soft error rate may be in the range of one per $10^{15}–10^{20}$ instructions.
2.3.3 Detailed Microarchitecture

Based on the previous discussion, we propose two different levels of microarchitecture support for Paceline. The first one is a simple design that targets only the frequent types of errors — namely the timing and input incoherence errors. The second level is a high-reliability design that targets all three types of errors. We call these designs Simple and High-Reliability, respectively. They can both be supported by a single microarchitecture, with some of the features being disabled for the Simple design.

There is a key difference between the two designs. Specifically, since timing and incoherence errors can only affect the leader, Simple can recover without rollback — it recovers simply by copying the checker’s state to the leader’s. On the other hand, since soft errors can affect both cores, High-Reliability must recover by rolling back. Unfortunately, the timing and incoherence errors will likely or possibly repeat during re-execution. To avoid this, the re-execution has to be performed under different conditions. As we will see, this complicates the High-Reliability design over the Simple one. Overall, these two designs correspond to different cost-functionality tradeoffs.

In this section, we describe the two designs. In both cases, the Paceline microarchitecture must provide two mechanisms: one to periodically compare the state of the leader and the checker, and another to repair the state of the leader-checker pair when the comparison mismatches.

The Simple Design

In this design, the checker is assumed to be always correct. Ensuring that the leader is also correct is a performance issue, not a correctness one. If the leader diverges from correct execution, it will not help speed up the checker; the leader will prefetch useless data into the L2 and pass useless branch outcomes to the checker, whose branch outcomes will not match the leader’s predictions.

Since correctness is not at stake, the checkpoint signature can be short — e.g., four bits. The
VQ has a *Signature Queue* that contains signatures that cores deposit when they take checkpoints and hash them. The VQ hardware compares signatures from the two cores corresponding to the same point in the program. Moreover, writes from the checker’s L1 are immediately routed to the L2, while those coming from the leader’s L1 are discarded. If the leader and checker checkpoints are different but hash into the same signature due to aliasing, correctness does not suffer. Moreover, since the divergence is likely to be detected in subsequent signature comparisons, the leader will not continue on a wrong path for long.

When a mismatch is detected, we know that a timing or incoherence error has happened and that its effect is confined to the leader’s processor and L1 cache. At this point, the VQ state is invalidated and we need to repair the register and L1 cache states of the leader. To repair the register state, the checker’s checkpoint is copied to the leader’s registers, effectively rolling the leader forward past the error. To repair the L1 state, the simplest option is to invalidate the leader’s L1 contents. As execution resumes, the leader will re-fetch the correct data from the L2 on demand. This approach does not involve a large performance overhead because all the lines invalidated from the L1 are in the L2.

A more sophisticated option, which we do not adopt, is to have the VQ record the read miss and write addresses emanating from the leader L1 during each checkpoint interval and to selectively invalidate only those lines when a signature mismatches. This approach is not guaranteed to repair the L1 state, so an incoherence may resurface in future signature comparisons. If a certain number of signature comparisons in a row are found to mismatch, the recovery procedure can revert to flushing the leader’s L1, which eliminates any lingering incoherence.

As an optimization, the VQ may also include a *Write Queue* that buffers all leader writes until they are performed by the checker and sent to the L2. This prevents the leader from suffering incoherence when a recently-written line is evicted from its L1 and read again before the checker commits the corresponding write; instead, the correct value for the read is found in the Write Queue. As an alternative, a standard victim cache could serve the same purpose.
The High-Reliability Design

In this design, the leader or the checker may suffer an error. We need to make sure that any divergence between the leader and the checker states is detected. Missing a divergence leads to a correctness problem, since erroneous data can then propagate to the L2.

Paceline compares the register and L1 state of leader and checker. The register state is checked by comparing the checkpoint signatures of the cores, which are stored in the VQ’s Signature Queue. In the High-Reliability design, the signature has more bits (e.g., sixteen) than in the Simple design in order to minimize aliasing. In addition, the L1 state is checked by buffering the leader writes in an in-order circular Write Queue in the VQ. Each Write Queue entry contains the address and data of a leader write, together with a Validated bit. When a checker write arrives, the hardware compares its address and data to the next non-validated entry in the Write Queue and, if they match, the Validated bit for the entry is set. Note that writes issued by each core arrive at the VQ in program order. For a checkpoint interval to check successfully, the leader and checker register signatures at the end of the interval must match, and the VQ must have successfully validated all writes in the interval. When this is the case, the Validated writes are removed from the VQ and released to the L2 cache.

If, instead, the signatures mismatch or the writes are not all validated, Paceline rolls the checkpoint interval back and re-executes it. Rolling back the interval involves invalidating the VQ’s state, restoring the register state of the last successfully-compared checkpoint in both cores, and repairing the L1 state in both cores. To repair the L1 state, there are two alternatives. The simplest is to invalidate both L1 caches as in Simple. In this case, as the leader and checker re-execute, they will naturally re-populate their L1 caches with lines from the L2. A more advanced alternative, which we do not adopt, is to re-fetch only those lines accessed during the failing checkpoint interval (i.e., those which could have contributed to the failure). This can be done by operating the two L1 caches in a special mode during the re-execution that explicitly forces all accesses to miss and re-fetch from the L2. At the next successful checkpoint comparison, re-execution is
complete, and the L1s may return to normal mode.

Regardless of which L1 cleanup method is chosen, the VQ operation during re-execution is no different than its normal-mode operation; it buffers leader writes and sends load requests to the L2 as usual. However, some additional precautions are needed to handle timing errors and persistent input incoherence errors. Specifically, recall from Table 2.1 that a timing error will reappear in a normal re-execution. To avoid this, in a re-execution, the leader is clocked at a lower, safe frequency until the next checkpoint. We envision the high-frequency clock and the safe-frequency clock to be always available to the leader. When the leader enters re-execution mode, the high-frequency clock is disconnected from the leader and the safe-frequency one is fed to it.

A persistent input incoherence error is one that repeatedly occurs every time that the interval is re-executed [64]. Since this event is expected to be rare, the High-Reliability design uses a simple approach to handle it. Specifically, when the hardware detects that an interval has been re-executed more than a few times in a row, an interrupt is sent to the other cores to stall them until the local core pair successfully proceeds past the next checkpoint.

2.3.4 Additional Issues

Special Accesses

There are a few types of memory accesses, including atomic Read-Modify-Write (RMW) operations such as atomic increment, and non-idempotent reads such as I/O reads, that require special support in a paired architecture such as Paceline. These accesses are called serializing operations in [64]. In this section, we briefly describe the basic support required.

At the simplest level, serializing operations can be implemented by checkpointing the leader before issuing the operation and then stalling the leader until the checker reaches the same point. Once this happens, the checker also checkpoints. Then, the checkpoint interval is validated by comparing the checkpoint signatures and, in the High Reliability design, also comparing all the
writes in the interval. After that, both cores issue the serializing operation, which is merged in the VQ and issued as a single access to memory. Let us call $r$ the data that this operation returns, such as the data read in the RMW operation or in the non-idempotent read. The value $r$ is provided to both cores and also temporarily buffered in the VQ. After that, both cores take a new checkpoint and the checkpoint interval is validated. In case of a match, the cores continue.

If, instead, a mismatch occurs, an error has happened since the previous checkpoint — for example a soft error has affected one of the register files. In this case, the hardware uses the same recovery procedures as before except that the serializing operation cannot be redone. Instead, the buffered $r$ value is provided to the processors.

As a simple performance optimization in the case of atomic RMW operations, the leader can issue an exclusive prefetch for the RMW variable into the L2 before stalling to wait for the checker. With this support, by the time the leader and checker together issue the RMW operation, the data is likely to be in the desired state in the L2. Note that this optimization cannot be applied for non-idempotent reads.

It is possible to design higher performance implementations where leader and checker do not need to wait for one another. For example, in an RMW operation, the leader could issue the read and buffer the write in the VQ, rather than waiting for the checker. The checker will later perform its RMW operation and, if both operations agree, a single operation is made visible to the memory. Otherwise, the recovery procedure is followed. Since this dissertation focuses on compute-intensive sequential sections, we do not evaluate these issues.

**Interrupts and Exceptions**

Delaying interrupt delivery is correct and allowable as long as the delay is bounded. When an interrupt arrives, Paceline must find some point in the future when the leader and checker have identical architectural state and then deliver the interrupt simultaneously to both. Checkpoint boundaries provide ideal points at which to deliver interrupts. Consequently, when an interrupt signal arrives at the processor pair, it goes first to the VQ. The VQ marks a pending interrupt
bit and waits until the next checkpoint is released. It then initiates a rollback to the just-released checkpoint and asserts the interrupt line on both cores before restarting them. Both cores then wake up in the same architectural state and see the interrupt simultaneously. The VQ remembers the interrupt that was just delivered so that it can re-deliver the same interrupt in the event of a future rollback recovery.

We assume that synchronous, program-initiated exceptions have precise state semantics in the processor, as is the case in most current designs. Intuitively, an instruction with an exception of this type behaves like a conditional jump with the additional effect of setting one or more status registers. This does not demand any special handling from Paceline; exceptions are handled just as any other instruction. Barring an error, the exception code will execute identically in both cores just as all other instructions do. In the case of an error causing the execution of the exception code in one core only, the next checkpoint comparison will fail and initiate recovery.

2.3.5 Implementation Feasibility

Minimizing changes to the processor cores and the cache coherence system has been a key goal for Paceline, and we believe that the resulting microarchitecture can integrate easily in a commercial CMP. Note that the VQ, the principal Paceline structure, falls outside the core boundary and occupies the relatively non-critical path between the L1 and L2 caches. Moreover, cache coherence, which happens at the L2 level, is completely oblivious to Paceline. Although the core itself does require some modifications (shown shaded in Figure 2.3), they affect only the fetch and retirement stages. The renaming, scheduling, and execution logic of the core is completely unaffected. The checkpointing hardware required for Paceline will likely be present in a future baseline system by default because several proposals for future core features (e.g., thread-level speculation and transactional memory) rely on the ability to perform frequent register checkpoints.

Perhaps the most demanding feature of Paceline in terms of implementation and verification effort is the requirement that, at a given time, different cores be able to run at different frequen-
cies. Adding clock domains to a design complicates verification, so until recently, multicore designs ran all cores at the same frequency. However, pressure to improve power efficiency has forced chip makers to implement per-core frequency scaling, with the Barcelona microarchitecture from AMD being the first to support it [19]. In Paceline, clock domain crossings occur in the BQ, VQ, and (for loads) between the leader L1 and the L2.

2.4 Analytical Performance Model

In order to enjoy speedup under Paceline, an application must have two properties: (i) overclocking the leader must increase its performance; and (ii) the improved behavior of the branch predictor and cache subsystems in the checker must increase its IPC. A program that is totally memory-bound or limited by the L2 cache latency will not satisfy (i), while a program that fits completely within the L2 cache and already achieves high branch prediction accuracy without Paceline will not meet requirement (ii).

Here, we present an intuitive method of estimating how much speedup Paceline can generate for a given overclocking factor and application. The method uses a standard (not Paceline-enabled) microarchitecture, so it can easily be added to early architectural design-space explorations. To get the required data, we instrument the simulator to output an IPC trace that records the average IPC for each chunk of ten thousand dynamic instructions. We then run the application on the simulator under two configurations. The first represents an overclocked leader with a perfect (infinite performance) checker, while the second represents a checker in the presence of a perfect (infinite performance) leader.

To obtain these leader and checker configurations, we start with the baseline core configuration. The leader configuration differs from the baseline only in being overclocked, so that the latencies of all memory components beyond the L1 cache increase by the overclocking factor. The checker configuration differs from the baseline only in having a perfect branch predictor and a perfect L2 cache (thanks to the BQ and prefetching effects). Running the simulator with each
configuration produces IPC traces. From these, we obtain performance traces by multiplying the entries in the leader IPC trace by the leader frequency and those of the checker trace by the checker frequency. Finally, we normalize these numbers to the performance of a baseline core. The two resulting speedup traces give the leader’s speedup \( L_i \) and the checker’s speedup \( C_i \) relative to a baseline core during the \( i \)th dynamic instruction chunk. For example, the leftmost two columns of Figure 2.4(a) show the \( L_i \) and \( C_i \) traces for the SPECint application \texttt{parser}.

The overclocking factor used for the leader is 1.3.

\[ L_i \quad C_i \quad \min(L_i, C_i) \quad S_j \]

The speedup \( S_j \) over the interval from program start to chunk \( j \) is then given by the harmonic mean of all \( P_i \) in the interval as shown in Equation 2.1. Referring to the last column of Figure 2.4, we see how \( S_j \) evolves as the application executes.

\[ S_j = \frac{j}{\sum_{i=1}^{j} \frac{1}{P_i}} \quad (2.1) \]
For an example of an application on which Paceline performs well, see parser in Figure 2.4(a). Here, the $L_i$ speedup trace is consistently near the maximum possible value of 1.3 because the leader experiences few L1 cache misses, so almost all of the execution takes place inside of the overclocked domain. The $C_i$ speedup in the checker from perfect branch and cache subsystems is high most of the time, allowing the checker to keep up with the leader most of the time. Consequently, the speedup converges to 1.25 with an overclocking factor of 1.3.

In contrast, ammp in Figure 2.4(b) does not show much potential for speedup. Early in the execution, the application has good cache and branch behavior in a baseline processor and, therefore, the $C_i$ speedups are often modest. The result is that the checker limits the Paceline speedup. Later, the application enters a memory-bound phase, in which $L_i$ speedups are low, and the leader limits the Paceline speedup.

We have predicted the Paceline speedups for our applications using this model. Table 2.2 compares the predicted speedups for the SPECint applications to speedups measured on the cycle-accurate Paceline-enabled simulator of Chapter 5. Given the strong agreement between the two sets of numbers, we believe that our trace-based model provides an attractive alternative to custom simulator implementation when evaluating Paceline.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Estimated</th>
<th>Actual</th>
<th>Appl.</th>
<th>Estimated</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>22%</td>
<td>21%</td>
<td>mcf</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>crafty</td>
<td>26%</td>
<td>25%</td>
<td>parser</td>
<td>25%</td>
<td>24%</td>
</tr>
<tr>
<td>gap</td>
<td>22%</td>
<td>20%</td>
<td>twolf</td>
<td>29%</td>
<td>29%</td>
</tr>
<tr>
<td>gcc</td>
<td>28%</td>
<td>28%</td>
<td>vortex</td>
<td>16%</td>
<td>13%</td>
</tr>
<tr>
<td>gzip</td>
<td>15%</td>
<td>18%</td>
<td>vpr</td>
<td>29%</td>
<td>29%</td>
</tr>
</tbody>
</table>

Table 2.2: Estimated versus measured Paceline speedups.

2.5 Summary

This chapter demonstrated a substantial potential for overclocking through TS in near-future multicore. It introduced the Paceline Simple microarchitecture, which provides a safe means of exploiting this overclockability to improve the performance of a thread (of a serial or a parallel ap-
plication). It also presented a *High-Reliability* variant that additionally provides tolerance to transient faults such as soft errors. In either variant, Paceline has minimal impact on multicore power density and hardware design complexity. Moreover, it is a configurable scheme that can be enabled or disabled on-demand at runtime. When disabled, it has no impact on power consumption and a very small die area footprint. With an overclocking factor of 1.3, Chapter 5 shows SPECint and SPECfp speedups of 21% and 8%, respectively, from outfitting a typical near-future core design with Paceline.
Chapter 3

BlueShift: Designing Pipelines for Timing Speculation

3.1 Introduction

A key limitation of current TS proposals (including Paceline) is that they assume traditional design methodologies, which are tuned for worst-case conditions and deliver suboptimal performance under TS. Specifically, existing methodologies strive to eliminate slack from all timing paths in order to minimize power consumption at the target frequency. Unfortunately, this can create a critical path wall that impedes overclocking. If the clock frequency increases slightly beyond the target frequency, the many paths that make up the wall fail all at once. The error recovery penalty then quickly overwhelms any performance gains from higher frequency.

In this chapter, we present a novel approach where the processor itself is designed from the ground up for TS. The idea is to identify the most frequently-exercised critical paths in the design and speed them up enough so that the error rate grows much more slowly as frequency increases. The majority of the static critical paths, which are rarely exercised, are left unoptimized or even deoptimized — relying on the TS microarchitecture to detect and correct the infrequent errors in them. In other words, we optimize the design for the common case, possibly at the expense of the uncommon ones. We call our approach and design optimization algorithm BlueShift.

This chapter also introduces two techniques that, when applied under BlueShift, improve processor performance. These techniques, called On-demand Selective Biasing (OSB) and Path Constraint Tuning (PCT), utilize BlueShift’s approach and design optimization algorithm. Both techniques target the paths that would cause the most frequent timing violations under TS and add slack by either forward body biasing some of their gates (in OSB) or by applying strong timing
constraints on them (in PCT).

Later, Chapter 6 evaluates BlueShift by applying it with OSB and PCT on modules of the OpenSPARC T1 processor. It finds that applying BlueShift with OSB to a Paceline core contributes an 8% speedup over a traditionally-designed core with little inherent overclockability. PCT achieves similar speedups, but targets a different microarchitecture. Although both techniques come with a significant power overhead, they are remarkable for providing a means of scaling logic delay that is orthogonal to voltage scaling. Moreover, OSB is a fully configurable technique that can be disabled at runtime. When disabled, it imposes no power overhead.

3.2 Taxonomy of Design for TS

This chapter aims to develop general design techniques that are compatible with a variety of TS schemes — not just Paceline. To that end, this section begins by outlining the key differences and similarities of previous TS proposals. Next, it constructs a taxonomy to encompass existing and future TS microarchitectures. Finally, it uses this microarchitecture taxonomy to develop a set of general approaches to enhance TS, each of which applies to different points in the taxonomy.

Stage-Level TS Microarchitectures

Razor [20], TIMERRTOL [75], CTV [59], and X-Pipe [77] detect faults at pipeline-stage boundaries by comparing the values latched from speculatively-clocked logic to known good values generated by a checker. This checker logic can be an entire copy of the circuit that is safely clocked [59, 75]. A more efficient option, proposed in Razor [20], is to use a single copy of the logic to do both speculation and checking. This approach works by wave-pipelining the logic [15] and latching the output values of the pipeline stage twice: once in the normal pipeline latch, and a fraction of a cycle later in a shadow latch. The shadow latch is guaranteed to receive the correct value. At the end of each cycle, the shadow and normal latch values are compared. If they agree, no action is taken. Otherwise, the values in the shadow latches are used to repair the pipeline state.
Another stage-level scheme, Circuit Level Speculation (CLS) [42], accelerates critical blocks (rename, adder, and issue) by including a custom-designed speculative “approximation” version of each. For each approximation block, CLS also includes two fully correct checker instances clocked at half speed. Comparison occurs on the cycle after the approximation block generates its result, and recovery may involve re-issuing errant instructions.

**Leader-Checker TS Microarchitectures**

In multicores, two cores can be paired in a leader-checker organization, with both running the same (or very similar) code, as in Paceline, Slipstream [70], and Reunion [64]. The leader runs speculatively and can relax functional correctness. The checker executes correctly and may be sped up by hints from the leader as it checks the leader’s work. Because the two cores are loosely coupled, configurability comes naturally; they can be disconnected and used independently in workloads that demand throughput.

One type of leader-checker microarchitecture sacrifices configurability in pursuit of higher frequency by making the leader core functionally incorrect by design. Optimistic Tandem [47] achieves this by pruning infrequently-used functionality from the leader. DIVA [7] can also be used in this manner by using a functionally incorrect main pipeline. This approach requires the checker to be dedicated and always on.

### 3.2.1 Classification of TS Microarchitectures

We classify existing proposals of TS microarchitectures according to: (1) whether the fault detection and correction hardware is always on (Configurability), (2) whether functional correctness is sacrificed to maximize speedup regardless of the operating frequency (Functional Correctness), and (3) whether checking is done at pipeline-stage boundaries or upon retirement of one or more instructions (Checking Granularity). In the following, we discuss these axes. Table 3.1 classifies existing proposals of TS microarchitectures according to these axes.
Table 3.1: Classification of existing proposals of TS microarchitectures.

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Configurability</th>
<th>Functional Correctness</th>
<th>Checking Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Razor [20]</td>
<td>Always-on</td>
<td>Correct</td>
<td>Stage</td>
</tr>
<tr>
<td>Paceline</td>
<td>On-demand</td>
<td>Correct</td>
<td>Retirement</td>
</tr>
<tr>
<td>X-Pipe [77]</td>
<td>Always-on</td>
<td>Correct</td>
<td>Stage</td>
</tr>
<tr>
<td>CTV [59]</td>
<td>Always-on</td>
<td>Correct</td>
<td>Stage</td>
</tr>
<tr>
<td>TIMERRTOL [75]</td>
<td>Always-on</td>
<td>Correct</td>
<td>Stage</td>
</tr>
<tr>
<td>CLS [42]</td>
<td>Always-on</td>
<td>Relaxed</td>
<td>Stage</td>
</tr>
<tr>
<td>Slipstream [70]</td>
<td>Always-on</td>
<td>Relaxed</td>
<td>Retirement</td>
</tr>
<tr>
<td>Optim. Tandem [47]</td>
<td>Always-on</td>
<td>Relaxed</td>
<td>Retirement</td>
</tr>
<tr>
<td>DIVA [7]</td>
<td>Always-on</td>
<td>Relaxed</td>
<td>Retirement</td>
</tr>
</tbody>
</table>

Configurability

The checker hardware that performs fault detection and correction can be kept Always-on or just On-demand. If per-thread performance is crucial all the time, the processor will always operate at a speculative frequency. Consequently, an Always-on checker suffices. This is the approach of most existing proposals. However, future multicores must manage a mix of throughput- and latency-oriented tasks. To save power when executing throughput-oriented tasks, it is desirable to disable the checker logic and operate at $f_r$. We refer to schemes where the checker can be engaged and disengaged with minimal power and area overhead as On-demand checkers.

Functional Correctness

Relaxing functional correctness can lead to higher clock frequencies. This can be accomplished by not implementing rarely-used logic, such as in Optimistic Tandem [47] and CLS [42], by not running the full program, such as in Slipstream [70], or even by tolerating processors with design bugs, such as in DIVA [7]. These Relaxed schemes suffer from errors regardless of the clock frequency. This is in contrast to Correct schemes, which guarantee error-free operation at and below the Limit Frequency $f_0$.

Relaxing functional correctness imposes a single (speculative) mode of operation, demanding an Always-on checker. Correctness at the Limit Frequency $f_0$ and below is a necessary condition for checker schemes based on wave pipelining [15] like Razor [20], or On-demand checker
schemes like Paceline.

Checking Granularity

Checking can be performed at pipeline-stage boundaries (Stage) or upon retirement of one or more instructions (Retirement). In Stage schemes, speculative results are verified at each pipeline register before propagating to the next stage. Because faults are detected within one cycle of their occurrence, the recovery entails, at worst, a pipeline flush. The small recovery penalty enables these schemes to deliver performance even at high fault rates. However, eager fault detection prevents them from exploiting masking across pipeline stages.

The alternative is to defer checking until retirement. In this case, because detection is delayed, and because recovery may involve heavier-weight operations, the recovery penalty is higher. On the other hand, Retirement schemes do not need to recover on faults that are microarchitecturally masked, and the loosely-coupled checker may be easier to build.

3.2.2 General Approaches to Enhance TS

Given a TS microarchitecture, Equation 1.1 shows that we can improve its performance by reducing $P_E(f)$. To accomplish this, we propose four general approaches. They are graphically shown in Figure 3.1. Each of the approaches is shown as a way of reshaping the original $P_E(f)$ curve of Figure 1.3(a) (now in dashes) into a more favorable one (solid). For each approach, we show that a processor that initially worked at point $a$ now works at $b$, which has a lower $P_E$ for the same $f$.

Delay Trading (Figure 3.1(a)) slows-down infrequently-exercised paths and uses the resources saved in this way to speed up frequently-exercised paths for a given design budget. This leads to a lower Limit Frequency $f'_0$ when compared to the one in the base design $f_0$ in exchange for a higher frequency under TS.

Pruning or Circuit-level Speculation (Figure 3.1(b)) removes the infrequently-exercised paths from the circuit in order to speed-up the common case. For example, the carry chain of the adder is only partially implemented to reduce the response time for most input values [42]. Pruning
results in a higher frequency for a given $P_E$, but sacrifices the ability to operate error-free at any frequency.

*Delay Scaling* (Figure 3.1(c)) and *Targeted Acceleration* (Figure 3.1(d)) speed-up paths and, therefore, shift the curve toward higher frequencies. The approaches differ in which paths are sped-up. Delay Scaling speeds-up largely all paths, while Targeted Acceleration targets the common-case paths. As a result, while Delay Scaling always increases the Limit Frequency, Targeted Acceleration does not, as $f'_0$ may be determined by the infrequently-exercised critical paths. However, Targeted Acceleration is more energy-efficient. Both approaches can be accomplished with techniques such as supply voltage scaling or body biasing \cite{73}.

### 3.2.3 Putting It All Together

The choice of a TS microarchitecture directly impacts which TS-enhancing approaches are most appropriate. Table 3.2 summarizes how TS microarchitectures and TS-enhancing approaches relate.

<table>
<thead>
<tr>
<th>Microarchitecture Characteristic</th>
<th>Implication on TS-Enhancing Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurability</td>
<td><em>Delay Trading</em> is undesirable with On-demand microarchitectures</td>
</tr>
<tr>
<td>Functional Correctness</td>
<td><em>Pruning</em> is incompatible with Correct microarchitectures</td>
</tr>
<tr>
<td>Checking Granularity</td>
<td>All approaches are applied more aggressively to <em>Stage</em> microarchitectures</td>
</tr>
</tbody>
</table>

Table 3.2: How TS microarchitectural choices impact what TS-enhancing approaches are most appropriate.
**Configurability** directly impacts the applicability of Delay Trading. Recall that Delay Trading results in a lower $f_0$ than the base case. This would force On-demand checking architectures to operate at a lower frequency in the non-TS mode than in the base design, leading to sub-optimal operation. Consequently, Delay Trading is undesirable with On-demand checkers.

The **Functional Correctness** of the microarchitecture impacts the applicability of Pruning. Pruning results in a non-zero $P_E$ regardless of the frequency. Consequently, Pruning is incompatible with Correct TS microarchitectures, such as those based on wave pipelining (e.g., Razor) or on-demand checking (e.g., Paceline).

**Checker Granularity** dictates how aggressively any of the TS-enhancing approaches can be applied. An approach is considered more aggressive if it allows more errors at a given frequency. Since Stage microarchitectures have a smaller recovery penalty than Retirement ones, all the TS-enhancing approaches can be applied more aggressively to Stage microarchitectures.

### 3.3 A Common-Case Optimization Flow

Our goal is to design processors that are especially suited for TS. Based on the insights from the previous section, we propose: (1) a novel processor design methodology that we call *BlueShift* and (2) two techniques that, when applied under BlueShift, improve processor frequency. These two techniques are instantiations of the approaches introduced in Section 3.2.2. Next, we present BlueShift and then the two techniques.

#### 3.3.1 The BlueShift Framework

Conventional design methods use timing analysis to identify the static critical paths in the design. Since these paths would determine the cycle time, they are then optimized to reduce their latency. The result of this process is that designs end up having a *critical path wall*, where many paths have a latency equal to or only slightly below the clock period.

We propose a different design method for TS processors, where it is fine if some paths take
longer than the period. When these paths are exercised and induce an error, a recovery mechanism is invoked. We call the paths that take longer than the period *Overshooting* paths. They are not critical because they do not determine the period. However, they hurt performance in proportion to how often they are exercised and cause errors.

Consequently, a key principle when designing processors for TS is that, rather than working with static distributions of path delays, we need to work with *dynamic* distributions of path delays. Moreover, we need to focus on optimizing the paths that overshoot most frequently *dynamically* — by trying to reduce their latency. Finally, we can leave unoptimized many overshooting paths that are exercised only infrequently — since we have a fault correction mechanism.

BlueShift is a design methodology for TS processors that uses these principles. In the following, we describe how BlueShift identifies dynamic overshooting paths and its iterative approach to optimization.

**Identifying Dynamic Overshooting Paths**

BlueShift begins with a gate-level implementation of the circuit from a traditional design flow. A representative set of benchmarks is then executed on a simulator of the circuit. At each cycle of the simulation, BlueShift looks for latch inputs that change after the cycle has elapsed. Such endpoints are referred to as overshooting. As an example, Figure 3.2 shows a circuit with a target period of 500ps. The numbers on the nets represent their switching times on a given cycle. Note that a net may switch more than once per cycle. Since endpoints $X$ and $Y$ both transition after 500ps, they are designated as overshooting for this cycle. Endpoint $Z$ has completed all of its transitions before 500ps, so it is non-overshooting for this cycle.

Once the overshooting endpoints for a cycle are known, BlueShift determines the path of gates that produced their transitions. These are the overshooting paths for the cycle, and are the objects on which any optimization will operate. To identify these paths, BlueShift annotates all nets with their transition times. It then backtraces from each overshooting endpoint. As it backtraces from a net with transition time $t_n$, it locates the driving gate and its input whose transition
Figure 3.2: Circuit annotated with net transition times, showing two overshooting paths for this cycle.

at time $t_i$ caused the change at $t_n$. For example, in Figure 3.2, the algorithm backtraces from $X$ and finds the path $b \rightarrow c \rightarrow e$. Therefore, path $b \rightarrow c \rightarrow e$ is overshooting for the cycle shown.

For each path $p$ in the circuit, the analysis creates a set of cycles $D(p)$ in which that path overshoots. If $N_{cycles}$ is the number of simulated cycles, we define the Frequency of Overshooting of path $p$ as $d(p) = |D(p)|/N_{cycles}$. Then, the rate of errors per cycle in the circuit ($P_E$) is upper-bounded by $\min(1, \sum_p d(p))$. To reduce $P_E$, BlueShift focuses on the paths with the highest frequency of overshooting first. Once enough of these paths have been accelerated and $P_E$ drops below a pre-set target, optimization is complete; the remaining overshooting paths are ignored.

**Iterative Optimization Flow**

BlueShift makes iterative optimizations to the design, addressing the paths with the highest frequency of overshooting first. As the design is transformed, new dynamic overshooting paths are generated and addressed in subsequent iterations. This iterative process stops when $P_E$ falls below target. Figure 3.3 illustrates the full process. It takes as inputs an initial gate-level design and the designer’s target speculative frequency and $P_E$.

At the head of the loop (Step 1), a physical-aware optimization flow takes a list of design changes from the previous iteration and applies them as it performs aggressive logical and physical optimizations. The output of Step 1 is a fully placed and routed physical design suitable for fabrication. Step 2 begins the embarrassingly-parallel profiling phase by selecting $n$ training
benchmarks. In Step 3, one gate-level timing simulation is initiated for each benchmark. Each simulation runs as many instructions as is economical and then computes the frequencies of overshooting for all paths exercised during the execution. Before Step 4, a global barrier waits for all of the individual simulations to finish. Then, the overall frequency of overshooting for each path is computed by averaging the measure for that path over the individual simulation instances. BlueShift also computes the average $P_E$ across all simulation instances.

BlueShift then performs the exit test. If $P_E$ is less than the designer’s target, then optimization is complete; the physical design after Step 1 of the current iteration is ready for production. As a final validation, BlueShift executes another set of timing simulations using a different set of benchmarks (the Evaluation set) to produce the final $P_E$ versus $f$ curve. This is the curve that we
use to evaluate the design.

If, on the other hand, $P_E$ exceeds the target, we collect the set of paths with the highest frequency of overshooting, and use an optimization technique to generate a list of design changes to speed-up these paths (Step 5). Different optimization techniques can be used to generate these changes. We present two next.

### 3.3.2 Example BlueShift Techniques

To speed-up processor paths, we propose two techniques that we call *On-demand Selective Biasing (OSB)* and *Path Constraint Tuning (PCT)*. They are specific implementations of two of the general approaches to enhance TS discussed in Section 3.2.2, namely Targeted Acceleration and Delay Trading, respectively. We do not consider techniques for the other approaches in Figure 3.1 because a technique for Pruning was already proposed in [47] and Delay Scaling is a degenerate, less energy-efficient variant of Targeted Acceleration that lacks path targeting.

**On-Demand Selective Biasing (OSB)**

On-demand Selective Biasing (OSB) applies forward body biasing (FBB) [73] to one or more of the gates of each of the paths with the highest frequency of overshooting. Each gate that receives FBB speeds up, reducing the path’s frequency of overshooting. With OSB, we push the $P_E$ versus $f$ curve as in Figure 3.1(d), making the processor faster under TS. However, by applying FBB, we also increase the leakage power consumed.

Figure 3.4(a) shows how OSB is applied, while Figure 3.4(b) shows pseudo code for the algorithm of Step 5 in Figure 3.3 for OSB. The algorithm takes as input a constant $k$, which is the fraction of all the dynamic overshooting in the design that will remain un-addressed after the algorithm of Figure 3.4(b) completes.

The algorithm proceeds as follows. At any time, the algorithm maintains a set of paths that are eligible for speedup ($P_{elig}$). Initially, at entry to Step 5 in Figure 3.3, Line 1 of the pseudo code in Figure 3.4(b) sets all the dynamic overshooting paths ($P_{oversh}$) to be eligible for speedup.
Figure 3.4: On-demand Selective Biasing (OSB): application to a chip (a) and pseudo code of the algorithm (b).

Next, in Line 2 of Figure 3.4(b), a loop begins in which one gate will be selected in each iteration to receive FBB. In each iteration, we start by considering all paths \( p \) in \( P_{elig} \) weighted by their frequency of overshooting \( d(p) \). We also define the weight of a gate \( g \) as the sum of the weights of all the paths in which it participates (\( \text{paths}(g) \)). Then, Line 3 of Figure 3.4(b) greedily selects the gate \( (g_{sel}) \) with the highest weight. Line 4 removes from \( P_{elig} \) all the paths in which the selected gate participates. Next, Line 5 adds the selected gate to the set of gates that will receive FBB \( (G_{FBB}) \). Finally, in Line 6, the loop terminates when the fraction of all the original dynamic overshooting that remains un-addressed is no higher than \( k \).

After this algorithm is executed in Step 5 of Figure 3.3, the design changes are passed to Step 1, where the physical design flow regenerates the netlist using FBB gates where instructed. In the next iteration of Figure 3.3, all timing simulations assume that those gates have FBB. We may later get to Step 5 again, in which case we will take the current dynamic overshooting paths and re-apply the algorithm. Note that the selection of FBB gates across iterations is monotonic; once a gate has been identified for acceleration, it is never reverted to standard implementation in subsequent iterations.

After the algorithm of Figure 3.3 completes, the chip is designed with body-bias signal lines that connect to the gates in \( G_{FBB} \). The overhead of OSB is the extra static power dissipated by the gates with FBB and the extra area needed to route the body-bias lines and to implement the body-bias generator [73].
In TS architectures with On-demand checkers like Paceline (Table 3.1), it is best to be able to disable OSB when the checker is not present. Indeed, the architecture without checker cannot benefit from OSB anyway, and disabling OSB also saves all the extra energy. Fortunately, this technique is easily and quickly disabled by removing the bias voltage. Hence the “on-demand” part of this technique’s name.

Path Constraint Tuning (PCT)

Path Constraint Tuning (PCT) applies stronger timing constraints on the paths with the highest frequency of overshooting, at the expense of the timing constraints on the other paths. The result is that, compared to the period $T_0$ of a processor without TS at the Limit Frequency $f_0$, the paths that initially had the highest frequency of overshooting now take less than $T_0$, while the remaining ones take longer than $T_0$. PCT improves the performance of the common-case paths at the expense of the uncommon ones. With PCT, we change the $P_E$ versus $f$ curve as in Figure 3.1(a), making the processor faster under TS — although slower if it were to run without TS. This technique does not intrinsically have a power cost for the processor.

Existing design tools can transfer slack between connected paths in several ways, exhibited in Figure 3.5. The figure shows an excerpt from a larger circuit in which we want to speed up path $A \rightarrow Z$ by transferring slack from other paths. Figure 3.5(a) shows the original circuit, and following to the right are successive transformations to speed up $A \rightarrow Z$ at the expense of other paths. First, Figure 3.5(b) refactors the six-input AND tree to reduce the number of logic levels between $A$ and $Z$. This transformation lengthens the paths that now have to pass through two 3-input ANDs. Figure 3.5(c) further accelerates $A \rightarrow Z$ by increasing the drive strength of the critical AND. However, we have to downsize the connected buffer to avoid increasing the capacitive load on $A$ and, therefore, we slow down $A \rightarrow X$. Figure 3.5(d) refines the gate layout to shorten the long wire on path $A \rightarrow Z$ at the expense of lengthening the wire on $A \rightarrow X$. Finally, Figure 3.5(e) allocates a reduced-$V_t$ gate (or an FBB gate) along the $A \rightarrow Z$ path. This speeds up the path but has a power cost, which may need to be recovered by slowing down another path.
The implementation of PCT is simplified by the fact that existing design tools already implement the transformations shown in Figure 3.5. However, they do all of their optimizations based on static path information. Fortunately, they provide a way of specifying “timing overrides” that increase or decrease the allowable delay of a specific path. PCT uses these timing overrides to specify timing constraints equal to the speculative clock period for paths with high frequency of overshooting, and longer constraints for the rest of the paths.

The task of Step 5 in Figure 3.3 for PCT is simply to generate a list of timing constraints for a subset of the paths. These constraints will be processed in Step 1. To understand the PCT algorithm, assume that the designer has a target period with TS equal to $T_{ts}$. In the first iteration of the BlueShift framework of Figure 3.3, Step 1 assigns a relaxed timing constraint to all paths. This constraint sets the path delays to $r \times T_{ts}$ (where $r$ is a relaxation factor), making them even longer than a period that would be reasonable without TS. When we get to Step 5, the algorithm first sorts all paths in order of descending frequency of overshooting at $T_{ts}$. Then, it greedily selects paths from this list leaving those whose combined frequency of overshooting is less than the target $P_E$. To these selected paths, it assigns a timing constraint equal to $T_{ts}$. Later, when the next iteration of Step 1 processes these constraints, it will ensure that these paths all fit within $T_{ts}$, possibly at the expense of slowing down the other paths.

At each successive iteration of BlueShift, Step 5 assigns the $T_{ts}$ timing constraint to those paths that account for a combined frequency of overshooting greater than the target $P_E$ at $T_{ts}$. Note that once a path is constrained, that constraint persists for all future BlueShift iterations.

Eventually, after several iterations, a sufficient number of paths are constrained to meet the target.
3.4 Summary

This chapter introduced BlueShift, a new design approach and optimization algorithm in which the processor is designed from the ground up for TS. The idea is to identify and optimize the most frequently-exercised critical paths in the design, at the expense of the majority of the static critical paths. It then proposed two specific optimization techniques that, when applied under BlueShift, improve processor performance: On-demand Selective Biasing (OSB) and Path Constraint Tuning (PCT). These techniques target the most frequently-exercised critical paths, and either add forward body bias to some of their gates or apply strong timing constraints on them.

When applied to modules from the OpenSPARC T1 processor, the evaluation in Chapter 6 shows that OSB and PCT provide average speedups of 8% and 6% on top of traditionally-designed Paceline and Razor cores, respectively. BlueShift thus provides a new way to speed up logic modules that is orthogonal to voltage scaling.
Chapter 4

LeadOut: Combining Timing Speculation with $V-f$ Boosting

4.1 Introduction

Having described a timing speculation microarchitecture and design methodology, we now expand our search for configurable techniques to enhance per-thread performance. This chapter focuses on combining TS with a voltage and frequency-boosting technique recently introduced in Intel’s Nehalem processor as Turbo Boost [34, 5]. The idea of is to increase the voltage and frequency of the active cores on a multicore beyond nominal until the chip reaches its power envelope. The hardware overhead is modest because it uses the same mechanisms as dynamic voltage-frequency scaling (DVFS), which is already widely deployed for power savings. This makes voltage–frequency boosting an especially attractive option for adding configurability to a multicore.

This chapter observes that individual application of either voltage–frequency boosting or TS is suboptimal. Alone, neither technique will be able to consistently bring the multicore to its power envelope or limit temperature. Instead, the technique will be limited by other factors, such as the maximum supply voltage (beyond which reliability suffers) or maximum error rate (beyond which performance drops due to frequent recovery), leaving some of the available power and temperature headroom untapped. This means that the two techniques can combine synergistically to unlock much higher levels of single-thread performance than either can alone. Moreover, we demonstrate a dynamic controller that co-optimizes the two techniques. Under a variety of loading conditions, the two techniques together bring the multicore to the extremes of its power and temperature envelope in an aggressive pursuit of performance without violating any con-
Chapter 7 evaluates the controlled combination of these techniques on a 16-core configurable desktop multicore where pairs of cores can be coupled into Paceline mode or decoupled, and where each core has its own voltage and frequency domain. The results show that when half of the cores are busy and the goal is to optimize a new performance-critical thread, the combined approach enables the thread to attain 34% higher performance than before while consuming 220% more power. Moreover, the configurability of the proposed multicore enables similar gains under various load conditions.

4.1.1 V–f Boosting

The Intel Core i7 (“Nehalem”) processor introduces the Intel Turbo Boost technology, where both supply voltage (\(V_{dd}\)) and frequency (\(f\)) are increased when high performance is desired on the active (non-idle) cores and the chip is operating below its power and thermal limits. To support this feature, the processor includes sensors and an on-chip controller that continuously monitors the power, temperature, and current in the chip.

When the operating system requests performance state P0 (highest performance) for one or more cores and places other cores into an inactive state, the on-chip controller transparently attempts to scale up \(V_{dd}\) and \(f\) on the active cores. As long as the on-chip sensors report safe conditions, the controller increases the frequency in steps of 133.33 MHz with a corresponding voltage increase. The maximum number of steps is determined by the number of inactive cores. If, at any time, the power, temperature, or current conditions reach unsafe values, the hardware automatically decreases \(V_{dd}\) and \(f\) one 133 MHz step.

Note that in Intel’s Turbo Boost, all active cores increase and decrease \(V_{dd}\) and \(f\) at the same time. In this paper, we use a more advanced design where each core increases its \(V_{dd}\) and \(f\) independently of the other cores. Moreover, the \(V_{dd}\) and \(f\) increases are not a fixed function of the number of inactive cores; instead, they continue until the application hits a power or temperature constraint or another limitation.
4.2 V–f Boosting and TS are Synergistic

The key insight of this chapter is that TS and voltage-frequency boosting are synergistic and multiply their speedups on per-thread performance. This provides great potential for boosting per-thread performance in multicores.

4.2.1 Power Increase in VBoost and Paceline

We call the technique described in Section 4.1.1 VBoost and compare it with Paceline from Chapter 2. Of the two, VBoost is simpler to implement, but Paceline still has lower complexity than other per-thread performance enhancements (e.g., thread-level speculation, core fusion, or very wide superscalars) because it requires few changes to the core. The modifications needed to integrate the BQ, VQ, and signature/checkpointing logic are concentrated in the fetch and retire stages, where they are unlikely to affect the critical path.

Our goal is to apply these two techniques to threads that demand per-thread performance until a core reaches its maximum allowed power ($P_{\text{max}}$) or temperature ($T_{\text{max}}$). These techniques have different power behaviors; per-core power (and, therefore, temperature) increases faster with VBoost than with Paceline. This arises from the fact that VBoost increases both $V_{\text{dd}}$ and $f$, while Paceline only increases $f$. We can see this in the formulas for static power due to leakage ($P_{\text{leak}}$) in Equation 4.6, dynamic power ($P_{\text{dyn}}$ in Equation 4.5), and logic gate delay [55] ($T_g$ in Equation 4.4).

\[ V_t = V_{t0} + K_{\text{temp}} (T - T_0) + K_{\text{DIBL}} (V_{\text{dd}} - V_{\text{dd0}}) \]  
(4.1)

\[ \mu \propto T^{-1.5} \]  
(4.2)

\[ I_{\text{leak}} \propto \frac{\mu}{L_{\text{eff}}} T^2 e^{-q V_t/(kTn)} \]  
(4.3)

\[ T_g \propto \frac{V_{\text{dd}} L_{\text{eff}}}{\mu (V_{\text{dd}} - V_t)^{\alpha}} \]  
(4.4)

\[ P_{\text{dyn}} \propto C V_{\text{dd}}^2 f \]  
(4.5)

\[ P_{\text{leak}} = V_{\text{dd}} I_{\text{leak}} \]  
(4.6)

As VBoost increases $V_{\text{dd}}$, both $P_{\text{dyn}}$ and $P_{\text{leak}}$ increase rapidly. Indeed, $P_{\text{leak}}$ depends directly
on $V_{dd}$ (Equation 4.6) and, through $I_{leak}$ (Equation 4.3), exponentially on $V_{dd}$ (since $V_t$ is a function of $V_{dd}$ in Equation 4.1). Meanwhile, $P_{dyne}$ has a quadratic dependence on $V_{dd}$ (Equation 4.5) and a linear dependence on $f$ (which itself depends on $V_{dd}$).

In general, when speeding up a single thread with either of these techniques, $VBoost$ will reach power or temperature constraints at lower frequencies than Paceline — assuming, of course, that no other constraint limits the frequency increase earlier. One way to improve the effectiveness of $VBoost$ is to add activity migration [26], which moves the thread among nearby cores to reduce the formation of hotspots. We call the combination of $VBoost$ and migration with the neighboring core $VBo+Mig$.

### 4.2.2 Composing the Techniques

The two techniques are synergistic because, in principle, the application of each one is limited by a different constraint. Specifically, suppose that we have as much power and temperature headroom as we want and use one of the techniques to gradually increase the frequency of a core. If we apply Paceline, the limiting factor will be the value of the error rate $P_{Emax}$ beyond which the performance begins to drop because of the frequency of error recovery. Instead, if we apply $VBoost$, the limiting factor will be the value of the process maximum supply voltage ($V_{ddmax}$), beyond which the devices become unreliable.

Pictorially, this is shown in Figure 4.1. Figure 4.1(a) shows a typical $P_E$ versus $f$ curve for a core. The core works at $f$=A. If we apply Paceline, we eliminate guardbands and even allow the processor to tolerate some errors. We work at $f$=B in Figure 4.1(b), and accomplish an $f$ increase of $\Delta f_L$. Instead, if we apply $VBoost$ alone, the entire $P_E$ versus $f$ curve shifts to the right — while changing its slope in some way. Figure 4.1(c) shows that the processor can now work at $f$=C while still respecting the guardband. We have obtained an $f$ increase of $\Delta f_B$. Finally, if we combine both $VBoost$ and Paceline, we still have the same curve as Figure 4.1(c) but, as shown in Figure 4.1(d), we remove the guardband and tolerate some errors to operate at $f$=D for an $f$ increase of $\Delta f_{L&B}$.
Figure 4.1: Qualitative depiction of increases in processor $f$ when the VBoost and Paceline techniques are applied.

In practice, the core may reach its power or temperature limits before the two techniques are applied to their full extent. Figure 4.1(e) shows such a case where the $f$ reaches only $E$. Note that the core running the critical thread reaches the temperature or power constraint sooner or later depending, in part, on the load in the rest of the multicore. Specifically, if the load is high, the temperature of the chip and, therefore, of the core is higher to start with. Moreover, a higher temperature induces higher leakage power, reducing the room to $P_{\text{max}}$. Overall, we define three operational regimes, as shown in Table 4.1.

In the Individual regime, the application of Paceline alone is sufficient to bring a core its temperature or power limit. The combination of the two techniques — which we refer to as VBo+Pl — cannot deliver gains because the voltage of the Paceline cores cannot be increased without violating the temperature or power constraints. This regime has a higher chance to occur when the rest of the chip is heavily loaded, as the other active cores generate a background of high temperature that reduces the room to $T_{\text{max}}$ and $P_{\text{max}}$ in this core. Our evaluation finds this case to be rare.

In the Synergistic regime, Paceline alone does not bring a core to its power or temperature limits. Instead, it reaches the maximum timing error rate first. However, the application of both techniques together does bring the core to its temperature or power limits. In this regime, the combination of techniques is likely to deliver gains. Chapter 7 finds that this regime dominates when the chip is under high to moderate load.

Finally, in the Unfulfilled regime, even when both techniques are applied together, the pro-
<table>
<thead>
<tr>
<th>Regime</th>
<th>Bounding Constraints</th>
<th>Bounded by T or P?</th>
<th>Gain from combining?</th>
<th>Multicore load</th>
<th>When seen in our evaluation (Chapter 7)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>✓ ✓ ✓</td>
<td>Paceline bounded, combination bounded</td>
<td>No</td>
<td>Very High</td>
<td>Rarely</td>
</tr>
<tr>
<td></td>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synergistic</td>
<td>✓ ✓ ✓</td>
<td>Paceline not bounded, combination bounded</td>
<td>Likely</td>
<td>High to Moderate</td>
<td>2, 4, 8 proc. available for use</td>
</tr>
<tr>
<td></td>
<td>✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unfulfilled</td>
<td>✓ ✓ ✓</td>
<td>Combination not bounded</td>
<td>Yes</td>
<td>Low</td>
<td>16 proc. available</td>
</tr>
</tbody>
</table>

Table 4.1: Multicore constraint regimes.
cessor reaches the maximum supply voltage and limiting timing error rate before arriving at the temperature or power constraint. In this case, the combination of techniques delivers gains, but there is power available for per-thread performance improvement that goes unused and could be spent on additional techniques if they were available. Chapter 7 finds that this regime is common in lightly-loaded multicores.

### 4.2.3 A Highly-Configurable Multicore

To support \( VBoost \) and \( VBo+Mig \), we give each core from Figure 2.2 its own voltage domain. To support Paceline, each core already has its own frequency domain. When per-thread performance matters, the core’s voltage and frequency are ramped up. Under \( VBo+Mig \), the thread bounces between the core and its neighbor to equalize temperatures. In this case, each of the two cores oscillates between a high-voltage, high-frequency mode and a shut-down mode. While we could migrate the thread to a remote core, the benefits of a shared cache more than compensate for the slightly higher temperature resulting from lateral heat conduction between the adjacent cores.

The configurable multicore operates as follows. At any time, there are \( R \) threads where throughput matters and \( S \) threads where per-thread performance is paramount. The multicore can assign each of the \( S \) threads either to a single core using \( VBoost \), or to a core pair using \( VBo+Mig \), Paceline, or \( VBo+Pl \) — the decision is made by an optimizing controller. The \( R \) threads run normally on remaining cores, and any unused cores are shut down. As the load \((R, S)\) changes, the controller reconfigures the mode, \( f \), and \( V_{dd} \) of each core to maximize overall performance.

### 4.3 Dynamic Controller Design

To manage the reconfigurability of the multicore, we design a dynamic controller that runs in software. In this section, we formulate the problem, present the per-thread controllers, and finally present the global controller.
4.3.1 Problem Formulation

Our goal is to maximize the speed of the $S$ threads, which are performance-critical. We do this by consuming all the power and thermal headroom available in the cores that run them. To ensure reliable operation, we must respect two constraints: the per-core power consumption must be less than $P_{max}$, and the hottest point in the core must have a temperature less than $T_{max}$. Additionally, to simplify the control, when using the Paceline technique, the timing error rate $P_E$ is constrained to be less than $P_{Emax} = 10^{-5}$. Note that we use per-core constraints rather than chip-wide ones because our hardware has independent per-core power grids and supplies, with no sharing of the current load between cores.

In this environment, we propose a thread controller in software per $S$ thread (which may control one or two cores, depending on the technique used), and a simple global controller also in software that oversees the thread controllers. The inputs and outputs of each thread controller are:

**Inputs:** The controller reads a power and a temperature sensor in the core (or each of the two cores) that it controls. This is like the Intel Foxton [46]. Additionally, if it controls two cores running in leader-checker mode, it reads two other sensors. The first one is a counter with the rate of checkpoint rollbacks observed — which gives the error rate $P_E$. The second one is the average occupancy of the leader-checker coupling queues — which gives an indication of whether the checker is falling behind the leader.

**Outputs:** The controller sets the voltage and frequency of the core (or each of the two cores) that it controls. We assume that $V$ and $f$ can be modified arbitrarily at every control interval (1 ms) with a minimum granularity of 10 mV and 100 MHz, respectively. The controller also sets the microarchitectural configuration in the cores depending on what technique is to be applied ($V_{Boost}$, $V_{Bo+Mig}$, Paceline, $V_{Bo+Pl}$, or none).
4.3.2 Thread Controllers

A thread controller is implemented with two software modules: the \( f \)-subcontroller that sets the core frequency and the \( V \)-subcontroller that sets the core voltage. Next, we describe the way they work for each of the techniques.

**Thread Controller for VBoost:** In this technique, the thread controller increases the core’s \( f \) as much as possible while applying an elevated \( V \). Figure 4.2(a) shows the organization. The \( f \)-subcontroller drives the optimization by executing a simple search for the maximum \( f \) as it monitors the core’s \( P \) and \( T \) for constraint violations. In the absence of violations, it attempts to increase \( f \) at the rate of 100 MHz every 2 ms. Otherwise, it initiates exponential backoff starting with a reduction of 100 MHz for the next control step.

![Thread Controller Examples](image)

Figure 4.2: Thread controller examples.

Whenever the \( f \)-subcontroller generates a \( f \) change, it sends the new \( f \) to the \( V \)-subcontroller, which determines whether a feasible \( V \) exists to support it. If so, the \( V \)-subcontroller acknowledges the new \( f \) and instructs the core PLLs and supplies to move to the new operating point. Otherwise, it notifies the \( f \)-subcontroller that its \( f \) choice was infeasible and the \( f \)-subcontroller responds by proposing successively lower \( f \) until one is accepted. The \( V \)-subcontroller implementation is simple — just a lookup table indexed by \( f \). At manufacturing test time, the table is populated with \( V \) that guarantee error-free (fully guardbanded) operation of the core for each possible \( f \) setting. Frequencies that cannot be supported within the allowable \( V_{dd} \) range are marked as

![Global Controller](image)

Figure 4.3: Global controller.
infeasible.

**Thread Controller for VBo+PI:** In this technique, the thread controller drives the leader $f$ as high as possible while keeping the error rate below $P_{E_{\text{max}}}$. Simultaneously, the checker frequency is adjusted to ensure that the checker does not fall behind and limit performance. Both the leader and checker can receive elevated voltages. However, at all times, the checker is operated at a fully-guardbanded voltage that ensures error-free operation just like a VBoost core.

As shown in Figure 4.2(b), this thread controller uses an $f$-subcontroller and a $V$-subcontroller for each of the two cores. Compared to VBoost, the leader’s $V$-subcontroller ($V_{\text{ldr}}$) and the checker’s $f$-subcontroller ($f_{\text{ckr}}$) are slightly different. The reason is that the $V_{\text{ldr}}$-subcontroller provides a $V$ that accommodates a certain error rate, and the $f_{\text{ckr}}$-subcontroller has the goal of enabling the checker to catch up with the leader.

The controller works as follows. The leader uses the same $f$-subcontroller as in VBoost to continuously search for the highest $f$. However, the leader’s $V_{\text{ldr}}$-subcontroller provides a $V$ that accommodates the $f$ target with an acceptable error rate $P_{E}$. Like in VBoost, the $V_{\text{ldr}}$-subcontroller is implemented as a lookup table indexed by $f$, but unlike in VBoost, it learns its entries dynamically in the field — since they depend on process variation, application, and environmental parameters. At power-on, all entries default to a minimum $V_{dd}$. Whenever a frequency $f_{\text{viol}}$ is requested and $P_{E_{\text{max}}}$ is exceeded, all table entries for $f \geq f_{\text{viol}}$ increment their $V$ by 10 mV. In this way, the table entries monotonically converge toward the minimal $V$s that ensure $P_{E} < P_{E_{\text{max}}}$. However, because of the monotonicity, transients bursts of errors can drive $V$ to values that are unnecessarily high in steady state. To combat this tendency, all entries are decremented by 10 mV once per second.

The $f_{\text{ckr}}$-subcontroller in the checker, rather than maximizing the checker’s $f$, attempts to keep the leader–checker coupling queues no more than half and no less than one quarter full. It does so by treating the leader–checker pair as a GALS system and applying the *attack-decay* control algorithm [60] as follows. Whenever the queues are more than half full, the checker $f$ is increased (*attack*) by 200 MHz per ms. When they are less than one quarter full, the $f$ is decreased.
(negative attack) by 100 MHz per ms. Otherwise, the \( f \) is decreased by 100 MHz every 7 ms (decay).

**Thread Controllers for VBo+Mig and Paceline**: These controllers are special cases of the two organizations already presented. In VBo+Mig, each of the two cores has the \( f \)-subcontroller and \( V \)-subcontroller of VBoost. The only difference is that, in the inactive core, the \( V \)-subcontroller turns off the power supply to conserve power. The thread controller for Paceline is a degenerate case of VBo+Pl’s where the two \( V \)-subcontrollers veto any \( f \) that is infeasible at the nominal \( V \).

### 4.3.3 Global Controller

The global controller receives information from a higher level of the system (e.g., the operating system or the runtime) about which threads are \( S \)-threads. It then attempts to speed-up all of these threads uniformly. To keep the control simple, the global controller simply selects one technique, which all the thread controllers apply to the \( S \) threads — VBoost, VBo+Mig, Paceline, VBo+Pl, or none. All thread controllers are asked to apply the same technique. Figure 4.3 shows the overall organization. In the figure’s example, the global controller has decided to apply VBo+Pl to the \( S \) threads.

The algorithm used by the global controller is based on a heuristic that works well in our experiments of Chapter 7. Specifically, if the chip has enough idle cores to dedicate two cores to each \( S \) thread, then the global controller selects VBo+Pl for application. Otherwise, it selects VBoost. This heuristic is shown in Chapter 7 to be suboptimal only in rare cases when the chip is in the Individual regime of Section 4.2.2. Finally, the global controller reruns its algorithm every time that the number of \( S \) or \( R \) threads changes.

### 4.4 Summary

This chapter augmented a TS-enabled multicore with the ability to boost voltage and frequency of critical threads. In a large multicore with varying numbers of busy cores and critical threads,
This chapter observed that individual application of either TS or $V-f$ boosting alone is suboptimal. Due to supply voltage or error rate limitations, they are often unable to bring the multicore all the way to its power or temperature envelope. This chapter argued that the two techniques are complementary and can be synergistically combined to unlock much higher levels of single-thread performance. Moreover, it presented an example dynamic controller that co-optimizes the two techniques.

Among other experiments, Chapter 7 evaluates these techniques and their combination on a simulated 16-core, desktop-style configurable multicore where half of the cores are busy. When the goal is to run one additional thread at high performance, application of either Paceline or voltage boosting increases the thread’s performance by 13% or 20%, respectively, while increasing its power by 100%. However, if we combine both techniques, we unlock much higher single-thread performance. Specifically, the performance-critical thread attains 34% higher performance than before, while consuming 220% more power. The high configurability of a multicore with both techniques enables similar gains for various load conditions.
Chapter 5

Paceline Evaluation

Paceline is the central feature of the proposed configurable CMP. The evaluation therefore begins by considering Paceline by itself — without any of its subsequent BlueShift or voltage–frequency boosting enhancements.

5.1 Experimental Setup

We use a modified version of the cycle-accurate Wattch-enabled [13] SESC [2] simulator to evaluate the performance and power impact of Paceline. The evaluation assumes a cache-coherent multicore comprising 16 high-performance out-of-order cores, configured as shown in Table 5.1. The core design is slightly more aggressive than current designs like the Intel Core i7 [33], and pairs of cores share an L2 cache. The bottom of the table shows the parameters for the Paceline features, not present in the baseline architecture.

5.2 Results

5.2.1 Performance

To get a feel for Paceline’s performance potential, we first consider performance in the absence of errors. In other words, we assume that the leader frequency can be increased by some overclocking factor \( oc \) above \( f_r \) without causing timing errors and see how much performance Paceline can extract. As the overclocking factor increases, performance relative to a single baseline core increases monotonically toward an asymptote. Figure 5.1 shows the speedups for the in-
<table>
<thead>
<tr>
<th>Core Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>32nm, Out-of-Order, 3.8 GHz</td>
</tr>
<tr>
<td>Pipe width</td>
<td>6-fetch, 4-issue, 4-retire</td>
</tr>
<tr>
<td>Pipe depth</td>
<td>Min. 13 cycle mispredict penalty</td>
</tr>
<tr>
<td>ROB size</td>
<td>152</td>
</tr>
<tr>
<td>Scheduler size</td>
<td>40 fp, 80 int</td>
</tr>
<tr>
<td>LSQ size</td>
<td>54 LD, 46 ST</td>
</tr>
<tr>
<td>Branch pred</td>
<td>80Kb local/global tournament, unbounded RAS</td>
</tr>
<tr>
<td>L1 I cache</td>
<td>16KB, 2 cyc, 2 port, 2 way</td>
</tr>
<tr>
<td>L1 D cache</td>
<td>16KB WT, 2 cyc, 2 port, 4 way</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2MB WB, 2 ns round trip, 1 port, 8 way, shared by two cores, has stride prefetcher</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Memory</td>
<td>80 ns round trip, 10 GB/s max per core pair</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Paceline Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Design evaluated</td>
<td>Simple (Section 2.3.3)</td>
</tr>
<tr>
<td>VQ write queue</td>
<td>64 entries, 2 ports per core, 8 cycles</td>
</tr>
<tr>
<td>Ckpt interval</td>
<td>100 instructions</td>
</tr>
<tr>
<td>Ckpt restore cost</td>
<td>100 cycles</td>
</tr>
<tr>
<td>VQ ckpt queue</td>
<td>5 entries</td>
</tr>
<tr>
<td>Migration policy</td>
<td>Swap leader and checker every 250μs</td>
</tr>
</tbody>
</table>

Table 5.1: Microarchitecture parameters.

Individual SPECint and SPECfp applications without a BQ as the leader overclocking factor (oc) varies from 1.1 to 1.4 in increments of 0.1. The figure also includes bars for the geometric mean of the integer and floating-point applications. The figure shows that the SPECint applications do not experience much speedup from prefetching alone. In fact, only gap and mcf achieve any speedup at all without a BQ; at an overclocking factor of 1.3, they reach speedups of 2% and 4%, respectively. The remaining SPECint applications experience no speedups or even small slowdowns. The reason is that almost all of the SPECint2000 working sets fit within the L2 cache. When there are few L2 misses, there can be no prefetching benefit. Consequently, the leader only wastes L2 bandwidth.

On the other hand, with a BQ, Figure 5.2 shows large but application-dependent gains from Pceline, with speedups for the oc-1.3 case ranging from 1.03 (equake) to 1.29 (vpr). Moreover, while some applications (e.g., vortex) top out at modest overclocking factors, others could benefit from factors in excess of 1.4. With the BQ, applications with low L2 miss rates are
at an advantage, since the leader is able to do useful work when overclocked rather than stalling for main memory accesses. The BQ is a major benefit for most SPECint applications, where the dynamic branch count is high and the branch prediction accuracy relatively low. Here, the near-perfect branch information from the BQ can vastly increase the checker IPC (sometimes by a factor of two or more) — as was shown in the parser example of Figure 2.4. The dramatic impact of the BQ is in accord with previous work [22], which reported similar speedups from perfect branch prediction.

In contrast, the SPECfp applications present behavior that is difficult for Paceline to exploit, whether a BQ is present or not. Most of the time, one of the following two conditions prevails: Either the program is memory-bound so that overclocking the leader has little effect, or the program is hitting in the L2 cache and generating good branch predictions so that branch and cache hints do not help the checker keep up with the leader. This was the behavior seen from ammp in Figure 2.4.
5.2.2 Dynamic Power

We compare the dynamic power consumed by a Paceline pair to that consumed by two baseline cores sharing an L2 and independently executing the same program. Figure 5.3 shows the absolute dynamic power consumption for a pair of cores under five different configurations. The leftmost bar ($U$) represents the base unpaired case of two cores independently executing the same application. Moving to the right, the following four bars represent a Paceline system running with an overclocking factor of 1.1, 1.2, 1.3, and 1.4, respectively. The three segments within each bar show the power consumed in the checker core ($Checker$), leader core ($Leader$), and in the added Paceline VQ, BQ, checkpointing, and hashing structures ($Extra$). The core powers include the L1 instruction and data cache access power. Off-chip power is not considered.

A key observation is that the power of a Paceline pair is usually approximately equal to that of two baseline cores. Due to its improved branch prediction, the checker fetches and executes far fewer instructions in the Paceline system than it would in the base system. This is consistent with results from other work [49], which indicate that for the SPECint applications, per-
Figure 5.3: Dynamic power breakdown in two base cores sharing an L2 (leftmost bar $U$ in each group) and in Paceline mode for overclocking factors of 1.1 – 1.3 (right three bars in each group).
fect branch prediction reduces the number of fetched instructions by 53% on average. Using this information, we can obtain an intuitive feel for the relationship between baseline and Paceline power.

Consider gzip, which attains a speedup $S_{1.3} = 1.18$ with an overclocking factor of 1.3. Our simulations show that the checker fetches 51% fewer instructions than a baseline core. Wattch shows that in total, it consumes 39% less energy than the baseline core. The leader core, meanwhile, is executing approximately the same number of total instructions as the baseline core. However, the number of wrong-path instructions increases slightly because the longer relative memory access time leads to slower branch resolution. Additionally, the lower IPC of the leader reduces its energy efficiency, as parts of the chip (especially the clock network) remain active even on idle cycles. The net result is that the leader consumes 1% more energy than the baseline core.

Given the preceding, we can easily compute the total dynamic core power for gzip under Paceline. When the application runs on the baseline core, let $E_B$ be the dynamic energy consumed and $T_B$ the time taken. According to the above, the energy of the checker is $E_C = 0.61 E_B$, and the energy of the leader is $E_L = 1.01 E_B$. However, due to the Paceline speedup, the leader and checker energies are dissipated over a shorter interval than the baseline execution. The total core power under Paceline is then $(E_L + E_C) S_{1.3}/T_B \approx 1.91 P_B$, where $P_B$ is the power of a baseline core. In this case, the total core dynamic power is less than that of two baseline cores executing the same application.

### 5.2.3 Power Density

Although total dynamic power may not increase under Paceline, the leading core power clearly does. This thermal imbalance is a concern because it could worsen hot spots that erode overclockability and reduce device lifetime. Paceline avoids this problem through activity migration \cite{26,52}, periodically switching the core on which the hot thread runs. In Paceline, core swapping allows both the leader and checker core temperatures to roughly equal the temper-
ture of a baseline core. This is because each core is dissipating approximately the same dynamic power as a baseline core on average, and the package’s thermal RC network has an averaging effect on temperature. Since static power is dependent only on temperature, chip static power also does not change after applying Paceline.

Swapping the leader and checker is trivial in Paceline, and it has negligible effect on performance even if swaps are performed roughly once per million instructions. To see why, consider the simplest swapping mechanism: A swap starts with both cores rolling back to the most recent checkpoint as though an error had just been detected. Then, a mode bit change in the VQ and in the BQ effectively switches the identities of the two cores. Finally, switching the leader and core clock frequencies completes the swap. In this simple scheme, both cores begin with an empty L1 after the swap, but they repopulate it quickly from the shared L2 [48, 52].

5.2.4 Sensitivity to Errors

Until now, we have assumed a zero error rate, which models the effect of overclocking up to the safe zero-guardband frequency $f_0$. To see what happens as frequency passes $f_0$, we adopt the timing error model from [57] using its default parameters, which predict a gradual error onset.

To model the rate of timing errors, we assume that the onset of timing errors occurs at an aggressive overclocking factor of 1.3, where we set the error rate to one per $10^{12}$ cycles. This represents a far-future process where guardbands are large, providing an ideal environment for Paceline. The data in [57] predicts that a 9% frequency increase pushes the error rate from one per $10^{12}$ to one per $10^{4}$ instructions. It also shows that the increase is approximately exponential. Consequently, using this information, we set the error rate at an overclocking factor of $1.3 \times 1.09 \approx 1.42$ to be one per $10^{4}$ cycles, and use a straight line in a logarithmic axis to join the $10^{-12}$ and $10^{-4}$ error rate points. The resulting error rate line is shown in Figure 5.4.

To estimate the impact of these errors on Paceline speedups, we test three different recov-

---

1The error onset predicted by the default VARIUS parameters is significantly more gradual than that observed in our experiments with OpenSPARC in the following section, but it serves to illustrate the concepts clearly.
Figure 5.4: Error rate and geometric mean of the Paceline speedup in SPECint applications as the overclocking factor changes.

These penalties are modeled as a number of cycles that we force the checker to stall after every timing error. The recovery penalties modeled are 100, 1K, or 10K cycles. In actuality, simulations show that the Paceline recovery time is dominated by the time required to re-populate the L1 caches following the flush at the start of recovery. For the simulated microarchitecture, this overhead averages $\approx 1K$ cycles amortized over the post-recovery execution.

Figure 5.4 shows the resulting Paceline speedups over the baseline processor for each of the three penalties. The speedups correspond to the geometric mean of the SPECint applications. From the figure, we see that at low overclocking factors, the speedups are roughly proportional to the factors. However, the exponential increase in error rates causes the linear region to give way to a “cliff” where speedup plummets if the overclocking factor is pushed even 1–2% past the optimal point. We need, therefore, to be careful not to increase overclocking past the optimal point. Later, we will show that the controller designed for LeadOut (Section 4.3) solves this problem adequately.

The figure also demonstrates that the peak Paceline speedup is only weakly dependent on the recovery penalty; the topmost point of the curves is similar across different recovery penalties.
Consequently, we conclude that — at least for the error rates targeted here — optimizing recovery time need not be a design priority.
Chapter 6

BlueShift Evaluation

The preceding chapter demonstrated that Paceline is capable of generating large speedups if the leader overclocking factor is large. The purpose of BlueShift is to ensure leader overclockability even when the process guardbands are small and when a traditional design exhibits a critical path wall that is antagonistic to TS. This chapter demonstrates significant performance enhancement from applying BlueShift to just such a design.

The BlueShift PCT and OSB techniques are both applicable to a variety of TS microarchitectures. However, to focus our evaluation, we mate each technique with a single TS microarchitecture that, according to Section 3.2.3, emphasizes its strengths. Specifically, an Always-on checker is ideal for PCT because it lacks a non-speculative mode of operation where PCT’s longer worst-case paths would force a reduction in frequency. Conversely, an On-demand microarchitecture is suited to OSB because it does have a non-speculative mode where worst-case delay must remain short. The PCT design, where TS is on all the time, adopts an extremely aggressive performance target, while the OSB one targets a more power-efficient, configurable environment. Overall, we evaluate a high-performance Always-on Stage microarchitecture (Razor [20]) for PCT and a power-efficient On-demand Retirement one (Paceline) for OSB. We call the resulting BlueShift-optimized designs Razor+PCT and Paceline+OSB respectively. These are compared with the Razor Base and Paceline Base designs, which are microarchitecturally identical but use a traditional design methodology.
6.1 Experimental Setup

The core microarchitecture parameters are the same as in the preceding chapter (Table 5.1) for both Razor+PCT and Paceline+OSB. Table 6.1 shows the timing speculation support assumed for the two example architectures. The target error rates for Paceline and Razor have been set so that the processor will spend at most 1% of its time in error recovery.

<table>
<thead>
<tr>
<th>Paceline+OSB Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design evaluated</td>
</tr>
<tr>
<td>VQ write queue</td>
</tr>
<tr>
<td>Ckpt interval</td>
</tr>
<tr>
<td>Ckpt restore cost</td>
</tr>
<tr>
<td>VQ ckpt queue</td>
</tr>
<tr>
<td>Total target $P_E$</td>
</tr>
<tr>
<td>Migration policy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Razor+PCT Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error recovery cost</td>
</tr>
<tr>
<td>Total target $P_E$</td>
</tr>
</tbody>
</table>

Table 6.1: BlueShift TS microarchitecture parameters.

6.1.1 Modeling Overview

To accurately model the performance and power consumption of a gate-level BlueShifted processor, we partition the modeling task into two loosely-coupled levels. The lower level comprises the BlueShift circuit implementation, while the higher level consists of microarchitecture-level power and performance estimation. At the circuit-modeling level, we sample modules from the OpenSPARC T1 processor [69]. We apply BlueShift to these modules and use them to compute $P_E$ and power estimates before and after BlueShift. At the microarchitecture level, we want to model a more sophisticated core than the OpenSPARC. To this end, we simulate the out-of-order core of Table 5.1 in the SESC timing simulator.

The main modeling challenge lies in incorporating the circuit-level $P_E$ and power estimates into the microarchitectural simulation. Our approach is to assume that the modules from the OpenSPARC are representative of those in any other high-performance processor. In other words,
we assume that BlueShift would induce roughly the same $P_E$ and power characteristics on the out-of-order microarchitecture that we simulate as it does on the in-order processor that we can measure directly. With this assumption, we annotate the microarchitectural power models with the power parameters derived from the low-level circuit analysis to accurately capture the power impact of BlueShift. We also combine the module-level $P_E$s to estimate the frequency at which the whole pipeline would meet the target error rate and perform microarchitecture simulation at that frequency.

The following subsections first describe the technology model (needed to accurately capture leakage power). Next, they show how to generate the BlueShifted circuits. Finally, they detail how $P_E$ and power estimates are extracted from these circuits and used to annotate the microarchitectural simulation.

### 6.1.2 Technology Model

We model a 32nm technology that is an incremental improvement on Intel’s current high-performance 45nm process [9]. ITRS [1] projects that neither threshold nor supply voltage are scaling in the near term, so we use most values directly from [9] in our model, as shown in Table 6.2. The constant of proportionality for leakage in Equation 4.3 is set so that the die expends 25W of leakage power when under uniformly nominal conditions ($V_{dd} = 1V$, $T = 85^\circ C$, and no process variation). Finally, because BlueShift is intended to inject overclockability into designs that otherwise are not amenable to TS, the evaluation assumes a core timing guardband of only 10% using Fine Grain Binning (Section 2.2.4). The guardband is applied during manufacturing test by measuring the critical path delay $1/f_0$ of each core measured at $T_0$. This delay is then multiplied by the guardband to obtain the rated clock period $1/f_r$. 

68
## 6.1.3 BlueShifted Module Implementation

Using Synopsis Design Compiler 2007.03 and Cadence Encounter 6.2, we perform full physical (placed and routed) implementations of sample modules from the OpenSPARC T1 core in a real 130nm commercial standard cell process. We modified the cell library by adding low-$V_t$ gates that have a 10x higher leakage and a 20% lower delay than normal gates [61, 74]. To make the 130nm results more representative of the near-future 32nm technology that we target (Table 6.2), we scale the cell power and delay to match.

For each module, we perform four different implementations — one for each design (Paceline Base, Paceline+OSB, Razor Base, and Razor+PCT). Because Paceline Base requires low power consumption in its non-TS mode, we do not allow it to use low-$V_t$ devices, but otherwise, it uses the most aggressive per-module timing targets that a traditional design flow can meet. Paceline+OSB starts from the Paceline Base design, converting some of its gates to receive FBB. When FBB is enabled on a gate, that gate performs identically to a low-$V_t$ gate. Razor Base targets the most aggressive possible per-module timing target (at a power cost) by allowing the traditional design flow to freely assign low-$V_t$ gates. Razor+PCT also allows free assignment of low-$V_t$, but uses BlueShift PCT to set the timing constraints. Additionally, in the Razor experiments, we add hold-time delay constraints to the paths to accommodate shadow latches. Moreover, shadow latches are inserted wherever worst-case delays exceed the speculative clock period.

Table 6.3 lists the BlueShift parameters. Each profiling phase (Step 2 of Figure 3.3) comprises a parallel run of 200 (or 400 for OSB) benchmark samples, each one running for 25K cycles.

---

1 Admittedly, dealing with two technologies invites ambiguity. When following sections report power or delay numbers, they specify which technology (130nm or 32nm) that measurement assumes.

2 For some modules, the commercial design tools that we use are unable to meet the minimum path delay constraints, but we make a best effort to honor them.
For all implementations, we use the unmodified RTL sources from OpenSPARC, but we simplify the physical design by modeling the register file and the 64-bit adder as black boxes. In a real implementation, these components would be designed in full-custom logic. We use timing information supplied with the OpenSPARC to build a detailed 900MHz black box timing model (in 130nm) for the register file; then, we use CACTI [72] to obtain an area estimate and build a realistic physical footprint. The 64-bit adder is modeled on [82], and has a worst-case delay of 500ps in 130nm.

The sample modules are taken from throughout the pipeline and are shown in Table 6.4. Taken together, these modules provide a representative profile of the various pipeline stages. For each module, the Stage column of Table 6.4 shows where in the pipeline (Fetch/Decode, EXEcute, or MEMory) the module resides. The next two columns show the size in number of standard cells and the shortest worst-case delay $T_r$ achievable under a traditional CAD flow without using any low-$V_t$ cells (the Paceline Base design) in the 130nm process.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Stage</th>
<th>Num. Cells</th>
<th>$T_r$ (ns)</th>
<th>Target $P_E$ (Errors/Cycle)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparc_exu</td>
<td>EXE</td>
<td>21,896</td>
<td>1.50</td>
<td>$10^{-4}$ $10^{-6}$</td>
<td>Integer FUs, control, bypass</td>
</tr>
<tr>
<td>lsu_stb_ctl</td>
<td>MEM</td>
<td>765</td>
<td>1.11</td>
<td>$10^{-9}$ $10^{-7}$</td>
<td>Store buffer control</td>
</tr>
<tr>
<td>lsu_qctl1</td>
<td>MEM</td>
<td>2,336</td>
<td>1.50</td>
<td>$10^{-9}$ $10^{-7}$</td>
<td>Load/Store queue control</td>
</tr>
<tr>
<td>lsu_dctl</td>
<td>MEM</td>
<td>3,682</td>
<td>1.00</td>
<td>$10^{-9}$ $10^{-7}$</td>
<td>L1 D-cache control</td>
</tr>
<tr>
<td>sparc_ifu_dec</td>
<td>F/D</td>
<td>727</td>
<td>0.75</td>
<td>$10^{-8}$ $10^{-7}$</td>
<td>Instruction decoder</td>
</tr>
<tr>
<td>sparc_ifu_fdp</td>
<td>F/D</td>
<td>7,434</td>
<td>0.94</td>
<td>$10^{-9}$ $10^{-7}$</td>
<td>Fetch datapath and PC maintenance</td>
</tr>
<tr>
<td>sparc_ifu_fci</td>
<td>F/D</td>
<td>2,299</td>
<td>0.96</td>
<td>$10^{-9}$ $10^{-7}$</td>
<td>L1 I-cache and PC control</td>
</tr>
</tbody>
</table>

Table 6.4: OpenSPARC modules used to evaluate BlueShift.

The next two columns in Table 6.4 show the per-module error rate targets under PCT and OSB. This is the $P_E$ that BlueShift will try to ensure for each module. We obtain these numbers
by apportioning a “fair share” of the total processor $P_E$ to each module — roughly according to its size. With these $P_E$ targets, when the full pipeline is assembled (including modules not in the sample set), the total processor $P_E$ will be roughly $10^{-3}$ errors/cycle for PCT and $10^{-5}$ for OSB. These were the target total $P_E$ numbers in Table 5.1. They are appropriate for the average recovery overhead of the corresponding architectures: 5 cycles for Razor (Table 5.1) and about 1,000 cycles for Paceline (which include 100 cycles spent in checkpoint restoration as per Table 5.1). Indeed, with these values of $P_E$ and recovery overhead, the total performance lost in recovery is 1% or less.

The largest and most complex module is sparc_exu. It contains the integer register file, the integer arithmetic and logic datapaths along with the address generation, bypass, and control logic. It also performs other control duties including exception detection, save/restore control for the SPARC register windows, and error detection and correction using ECC. This module alone is larger than many lightweight embedded processor cores.

Although we find that BlueShift is widely applicable to logic modules, it is not effective on array structures where all paths are exercised with approximately equal frequency. As a result, we classify caches, register files, branch predictor, TLBs, and other memory blocks in the processor as Non-BlueShiftable. We assume that these modules attain performance scaling without timing errors through some other method (e.g. increased supply voltage) and account for the attendant power overhead.

### 6.1.4 Module-Level $P_E$ and Power

For each benchmark, we use Simics \[44\] to fast-forward execution over 1B cycles, then checkpoint the state and transfer the checkpoint to the gate-level simulator. To perform the transfer, we use the CMU Transplant tool \[66\]. This enables us to execute many small, randomly-selected benchmark samples in gate-level detail. Further, only the modules from Table 6.4 need to be simulated at the gate level. Functional, RTL-only simulation suffices for the remaining modules of the processor.
Table 6.5: Static power consumption ($P_{sta}$) and switching energy per cycle ($E_{dyn}$) for each module implementation in 130nm.

The experiments use SPECint2006 applications as the Training set in the BlueShift flow (Steps 1–5 of Figure 3.3). After BlueShift terminates, we measure the error rate for each module using SPECint2000 applications as the Evaluation set. From the latter measurements, we construct a $P_E$ versus $f$ curve for each SPECint2000 application on each module. All $P_E$ measurements are recorded in terms of the fraction of cycles on which at least one flip-flop or register receives the wrong value. This is an accurate strategy for the Razor-based evaluation, but because it ignores architectural and microarchitectural masking across stages, it is highly pessimistic for Paceline.

Circuit-level power estimation for the sample modules is done using Cadence Encounter. We perform detailed capacitance extraction and then use the tool’s default leakage and switching analysis. Table 6.5 shows the estimated switching energy per clock ($E_{dyn}$) and static power consumption ($P_{sta}$) for each module under each implementation in the 130nm ASIC technology.

### 6.1.5 Microarchitecture-Level $P_E$ and Power

We compute the performance and power consumption of the Paceline- and Razor-based microarchitectures using the SESC [2] simulator augmented with Watch [13], HotLeakage [79], and HotSpot [63] power and temperature models. For evaluation, we use the SPECint2000 applications, which were also used to evaluate the per-module $P_E$ in the preceding subsection. The sim-
ulator needs only a few key parameters derived from the low-level circuit analysis to accurately capture the $P_E$ and power impact of BlueShift.

To estimate the $P_E$ for the entire pipeline, we first sum up the $P_E$ from all of the sampled modules of Table 6.4. Then, we take the resulting $P_E$ and scale it so that it also includes the estimated contribution of all the other BlueShiftable components in the pipeline. We assume that the $P_E$ of each of these modules is roughly proportional to the size of the module. Note that by adding up the contributions of all the modules, we are assuming that the pipeline is a series-failure system with independent failures, and that there is no error masking across modules. The result is a whole-pipeline $P_E$ versus frequency curve for each application. We use this curve to initiate error recoveries at the appropriate rate in the microarchitectural simulator.

For power estimation, we start with the dynamic power estimations from Wattch for the simulated pipeline. We then scale up these Raw power numbers to take into account the higher power consumption induced by the BlueShift optimization. The scale factor is different for the BlueShiftable and the Non-BlueShiftable components of the pipeline. Specifically, we first measure the dynamic power consumed in all of the sampled OpenSPARC modules as given by Cadence Encounter (Table 6.5). The ratio of the power after BlueShift to the power before BlueShift is the factor that we use to scale up the Raw power numbers in the BlueShiftable components. For the Non-BlueShiftable components, we first compute the increase in supply voltage that is necessary for them to keep up with the frequency of the rest of the pipeline, and then scale their Raw power numbers accordingly.

For the static power, we use a similar approach based on HotLeakage and Cadence Encounter. However, we modify the HotLeakage model to account for the differing numbers of low-$V_t$ gates in each environment of our experiments. As a thermal environment, microarchitectural power simulations assume the 16-core floorplan of Figure 2.2, where each core is based on the default EV6 floorplan from HotSpot [63]. We optimistically assume that only one thread is running on the processor, yielding thermal headroom that allows per-thread performance to reach the highest possible levels. Heatsink thermal resistance $R_{th}$ is 0.33 $K/W$, and maximum temperature con-
strains are enforced.

6.2 Results

6.2.1 Error Curve Transformations

We begin by examining BlueShift’s effect on the whole-pipeline $P_E(f)$ curve. Figures 6.1(a)-(d) show the curves for each of the four implementations. These curves do not include the effect of the non-BlueShiftable modules. In each plot, the $x$ axis shows the frequency relative to the Rated Frequency ($f_r$) of Paceline Base. Each plot has one curve for each SPECint2000 application. In addition, there is a horizontal dashed line that marks the whole-pipeline target $P_E$, namely $10^{-5}$ errors per cycle for the Paceline-based environments and $10^{-3}$ for the Razor-based ones.

Figure 6.1: Whole-pipeline $P_E(f)$ curves for the four implementations. The frequencies are given relative to the Rated Frequency ($f_r$) of Paceline Base.

Figure 6.1(a) shows the curve for Paceline Base. Here, we assume a 10% guardband, so the maximum error-free frequency $f_0$ is 1.1 in the plot. As we increase $f$, $P_E$ takes non-zero values past $f_0$ (although it is invisible in the figure) and quickly reaches high values. This is due to the critical path wall of conventional designs. Consequently, using TS on a non-BlueShift OpenSPARC design can only manage frequencies barely above $f_0$ before $P_E$ becomes prohibitive.

Figure 6.1(b) shows the curve for Paceline+OSB. This plot follows the Targeted Acceleration shape of Figure 3.1. Specifically, $P_E$ starts taking non-zero values past $f_0$ like in Paceline Base. A static timing analysis shows that the worst-case delays are unchanged from Paceline Base.
However, the rise in $P_E$ is delayed until higher frequencies. Indeed, $P_E$ remains negligible until a relative frequency of 1.27, compared to about 1.11 in Paceline Base. This shows that the application of BlueShift with OSB enables an increase in processor frequency of 14%.

Figure 6.1(b) also shows a dashed vertical line. This was OSB’s frequency target, namely a 20% increase over Paceline Base (Section 6.1.3) — or $1.2 \times 1.1 = 1.32$. However, we see that OSB did not meet its target. This is because the Training application set (which was used in the optimization algorithm of Figure 3.3) failed to capture some key behavior of the Evaluation set (which was used to generate the $P_E$ curves). Consequently, Paceline+OSB, like other TS microarchitectures, will rely on its control mechanism to operate at the frequency that maximizes performance (1.27 in this case) rather than at its target. While higher frequencies may be possible with more comprehensive training, the obtained 14% frequency increase is substantial.

Figure 6.1(c) shows the curve for Razor Base. Since this is not a BlueShifted design, it exhibits a rapid $P_E$ increase as in Paceline Base. The difference here is that, because it targets a high-performance (and power) design point, it attains a higher frequency than Paceline Base.

Finally, Figure 6.1(d) shows the curve for Razor+PCT. The plot follows the Delay Trading shape of Figure 3.1. Specifically, $P_E$ starts taking non-zero values at lower frequencies than even Paceline Base, but the curve rises more gradually than in any of the other designs. The dashed vertical line shows the target frequency, which was 30% higher than Paceline Base (Section 6.1.3) — or $1.3 \times 1.1 = 1.43$. We can see that most applications reach this frequency at the whole-pipeline target $P_E$. Compared to the frequency of 1.28 attained by Razor Base, this means that BlueShift with PCT enables an increase in processor frequency of 12%. The two exceptions are the twolf and vpr applications, which fail to meet the target $P_E$ due to discrepancies between the Training and Evaluation application sets. For these applications, the Razor+PCT architecture will adapt to run at a lower frequency, so as to maximize performance.
### 6.2.2 Paceline+OSB Performance and Power

We compare three Paceline-based architectures. First, *Unpaired* uses the *Paceline Base* module implementation and one core runs at the Rated Frequency while the other is idle. Secondly, *Paceline Base* uses the *Paceline Base* module implementation and the cores run paired under Paceline. Finally, *Paceline+OSB* uses the *Paceline+OSB* module implementation and the cores run paired under Paceline. For each application, *Paceline Base* runs at the frequency that maximizes performance. For the same application, *Paceline+OSB* runs at the frequency that maximizes performance considering only the $P_E$ curves of the BlueShiftable components; then, we apply traditional voltage scaling to the non-BlueShiftable components so that they can catch up — always subject to temperature constraints.

Figure 6.2(a) shows the speedup of the *Paceline Base* and *Paceline+OSB* architectures over *Unpaired* for the different applications. We see that *Paceline+OSB* delivers a performance that is, on average, 8% higher than that of *Paceline Base*. Therefore, the impact of BlueShift with OSB is significant. The figure also shows that, on average, *Paceline+OSB* improves the performance by 21% over *Unpaired*. Finally, given that all applications cycle at approximately the same frequency for the same architecture (Figures 6.1(a) and 6.1(b)), the difference in performance across applications is largely a function of how well individual applications work under Paceline. For example, applications with highly-predictable branches such as *vortex* cause the checker to be a bottleneck and, therefore, the speedups in Figure 6.2(a) are small.

Figure 6.2(b) shows the power consumed by the processor and L1 caches in *Paceline Base*, *Paceline+OSB*, and two instances of *Unpaired*. The power is broken down into power consumed by the checker core (which is never BlueShifted), non-BlueShiftable modules in the leader, BlueShiftable modules in the leader, and extra Paceline structures (checkpointing, VQ, and BQ). On average, the power consumed by *Paceline+OSB* is 14% higher than that of *Paceline Base*. Consequently, BlueShift with OSB does add to the power consumption, but delivers a significant performance gain.
Figure 6.2: Performance (a) and power consumption (b) of Paceline before (Paceline Base) and after (Paceline+OSB) BlueShift. BS and NonBS refer to BlueShiftable and non-BlueShiftable modules, respectively.
6.2.3 *Razor+PCT* Performance and Power

We now compare two Razor-based architectures. *Razor Base* uses the *Razor Base* module implementation, while *Razor+PCT* uses the *Razor+PCT* one obtained by applying BlueShift with PCT. As before, *Razor+PCT* runs at the frequency given by the $P_E$ curves of the BlueShiftable components; then, we apply traditional voltage scaling to the non-BlueShiftable components so that they can catch up.

Figure 6.3(a) shows the speedup of the *Razor Base* and *Razor+PCT* architectures over the *Unpaired* one used as a baseline in Figure 6.2(a). Since these Razor-based designs target high performance, they deliver higher speedups. We see that, on average, *Razor+PCT*’s performance is 6% higher than that of *Razor Base*. This is the impact of BlueShift with PCT in this design — which is not negligible considering that *Razor Base* was already designed for high performance. We also see that *vpr* and, to a lesser extent, *twolf* do not perform as well as the other applications under *Razor+PCT*. This is the result of the unfavorable $P_E$ curve for these applications in Figure 6.1(d).

Figure 6.3(b) shows the power consumed by the two designs. Recall that all other cores on die are assumed idle, providing a favorable thermal environment. The power is broken down into the contributions of the non-BlueShiftable and the BlueShiftable modules. On average, *Razor+PCT* consumes 28% more power than *Razor Base*. This is because it runs at a higher frequency, uses a higher supply voltage for the non-BlueShiftable modules, and needs more shadow latches and hold-time buffers. In general, the *Razor* designs consume high power because of their high frequencies and widespread use of low-$V_t$ devices, which consume considerable leakage power at high temperatures. Consequently, as chip loading increases, all of the *Razor* designs quickly become temperature-limited and the performance gains from BlueShift evaporate.

Given *Razor+PCT*’s delivered speedup and power cost, we see that BlueShift with PCT is not compelling from an $E \times D^2$ perspective. Instead, we see it as a technique to further speed-up a high-performance design (at a power cost) when conventional techniques such as voltage scaling
Figure 6.3: Performance (a) and power consumption (b) of Razor before (Razor Base) and after (Paceline+OSB) BlueShift. BS and NonBS refer to BlueShiftable and non-BlueShiftable modules, respectively.
or body biasing do not provide further performance. Specifically, for logic (i.e., BlueShiftable) modules, BlueShift with PCT provides an orthogonal means of improving performance when further voltage scaling or body biasing becomes infeasible. In this case however, for the pipeline as a whole, non-BlueShiftable stages remain a bottleneck that must be addressed using some other technique.

6.2.4 Computational Overhead

Although most modules of Table 6.4 were fully optimized with BlueShift in one day on our 100-core cluster, the optimization of sparc_exu took about one week. Such long turnaround times during the frantic timing closure process would be unacceptable in industry. Fortunately, the current implementation is only a prototype, and drastic improvements in runtime are possible. Specifically, referring to Figure 3.3, a roughly equal amount of wall time is spent in physical implementation (Step 1) and profiling (Step 3). Luckily, the profiling phase is embarrassingly parallel, so simply adding more processors can speed it up. However, the CAD tools in Step 1 are mostly sequential. To reduce the overall runtime, the number of BlueShift iterations in Figure 3.3 must be reduced. This can be done, for example, by adding more constraints at each iteration. Our experiments added few constraints per iteration to avoid overloading the commercial CAD tools, which have a tendency to crash if given too many constraints.
Chapter 7

LeadOut Evaluation

This chapter explores the benefits of extreme configurability from combining Paceline\(^1\) with VBoost. It first examines the impact of each of the techniques when the goal is to accelerate a single thread on the processor in the presence of other throughput-oriented threads. In the process, it identifies the most performance-limiting constraints (e.g., maximum temperature $T_{\text{max}}$, per-core maximum power $P_{\text{max}}$, and maximum supply voltage $V_{\text{ddmax}}$) under different loading conditions. Next, it characterizes the power-performance tradeoffs available with the different techniques. It also provides a sensitivity study (Section 7.2.3). Finally, it charts the performance of the configurable multicore on a workload requiring multiple high-speed threads and multiple throughput threads.

7.1 Experimental Setup

Except where noted, all technology and microarchitecture parameters are identical to those in the preceding BlueShift evaluation. The main difference is that this evaluation adds the ability to dynamically vary voltage and frequency on a per-core basis. In other words, each core has its own voltage and frequency domain (although the L2 caches still run at nominal, fixed voltage and frequency). The maximum $V_{\text{dd}}$ allowed by the process is an especially critical parameter that limits performance scaling of VBoost. In Intel’s current 45nm process, the maximum $V_{\text{dd}}$ ranges from 1.26V – 1.38V depending on the processor’s current draw \(^{[33]}\) (lower values at higher currents). These values are likely to decrease slightly in future technologies in order to meet reliability re-

\(^{1}\)It excludes BlueShift for simplicity and clarity of evaluation
quirements. Therefore, for simplicity, we assume a maximum $V_{dd}$ of 1.3V. To maintain signal integrity and SRAM stability, the minimum $V_{dd}$ is set at 800mV. Likewise, to ensure power integrity, we limit the maximum per-core power consumption $P_{\text{max}}$ to 12W, which is roughly twice a core’s average power consumption.

The Paceline experiments in this chapter use the pre-BlueShift $P_E(f)$ curves measured in the preceding chapter except that they assume a 15% process guardband (instead of 10%) by default. As before, the guardband is applied during manufacturing test assuming Fine Grain Binning (Section 2.2.4), where the critical path delay of each core is measured at $T_{\text{max}}$. This delay is then multiplied by the guardband to obtain the rated clock period. Because the exact guardbands for current processes are not publicly disclosed and may vary significantly as technology scales, section 7.2.3 of this chapter undertakes a guardband sensitivity study.

Finally, the preceding Paceline and BlueShift evaluations ignored process variation for simplicity, but these variations have a profound impact on delay and power consumption. Consequently, this section adds the variation model from VARIUS [57] to the device parameters of Table 6.2. VARIUS models within-die variation comprising systematic (spatially correlated) and random (spatially uncorrelated) deviations in $V_{t0}$ and $L_{\text{eff}}$, both of which impact the effectiveness of the proposed techniques — e.g., by increasing leakage power for some cores and causing them to quickly reach their power or temperature limits as voltage increases. For the magnitude of variation, we assume the default parameters from VARIUS [57] as shown in Table 7.1, which are representative of near-future technologies; the magnitude of the random $V_{t0}$ variation assumed agrees with recent measurements from a 65nm technology [3]. VARIUS is also used to adjust the measured per-core error rates from Section 6.2.1 for the current supply voltage and temperature conditions.

Given this model, we perform three types of experiments. During the initial characterization of performance and power for each technique, we incorporate the effects of variation by running

\[ 2 \text{The experiments in this section use } n = 1.5 \text{ for historical reasons, but this is not expected to have a tangible effect on the outcomes.} \]
Table 7.1: Variation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Random $\sigma/\mu$ (%)</th>
<th>Systematic $\sigma/\mu$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{t0}$</td>
<td>6.4%</td>
<td></td>
</tr>
<tr>
<td>$L_{eff}$</td>
<td>3.2%</td>
<td></td>
</tr>
<tr>
<td>Corr. range ($\phi$)</td>
<td>1 cm</td>
<td></td>
</tr>
</tbody>
</table>

50 Monte Carlo simulations, each using a different sample die variation map, while running all the SPECint2000 codes. Additionally, we construct a pseudo-oracle against which to compare the results of our proposed dynamic controller. The pseudo-oracle uses Nelder-Mead optimization [50] to generate static (i.e., not time-varying) values for the controller outputs from Section 4.3.1. Finally, to keep the computation reasonable, when we conduct sensitivity studies in Section 7.2.3, we use only one typical die selected from among the 50 samples created before.

### 7.2 Results

#### 7.2.1 Performance

The performance of both Paceline and VBoost is determined by the application’s responsiveness to frequency increase (i.e., the degree to which the application is CPU-bound) and the amount of frequency increase that is possible within the $P_{max}$, $T_{max}$, $V_{ddmax}$, and $P_{Emax}$ constraints. We begin with the goal of speeding up a single target thread. This speedup will be highly dependent on the chip’s loading, which we measure in terms of the number of idle cores that are available to run the target thread: an un-loaded chip would have all 16 cores available, while a maximally-loaded chip would have only one core available and the other cores would be active running other load. Note that Paceline, VBo+Mig, and VBo+Pl require a core pair to run, so are only applicable when there are at least two cores available.

Figure 7.1 shows the performance improvements attained by the SPECint applications after applying each of the techniques. The improvements are over the application running on a plain core — an environment we call Unoptimized. For each application, the individual bars cor-
respond to the different techniques. In a given bar, we stack the improvements attained under
different amounts of load in the multicore — measured in number of available cores. The top
segment represents the lightest possible load, while the bottom segment represents the heaviest.
The rightmost group of bars (\textit{p-oracle}) shows the performance of the pseudo-oracular optimizer
for comparison. Note that because its outputs are static, the pseudo-oracle does not provide a
strict upper bound on performance. Nevertheless, the real controller compares favorably with the
pseudo-oracle.

Figure 7.1: Performance improvements relative to \textit{Unoptimized} at various multicore loading
levels.

In situations with just one core available, the only applicable technique is \textit{VBoost} (because
the others require a free core pair), and it improves performance by 12\% on average. If two cores
are available, \textit{Paceline} comes online to provide a performance gain of 12\%, which is compara-
tible to \textit{VBoost}'s 14\%. Note that even at such high loading, the system is at least in the \textit{Synergistic}
regime. This is seen by combining \textit{Paceline} and \textit{VBoost}, which deliver a 26\% average perfor-
mance improvement. In fact, the \textit{Individual} regime occurs only once, namely for \textit{vortex}, when
just two cores are free. In this case, \textit{VBo+Mig} outperforms \textit{VBo+Pl} because of tight thermal con-
straints.
For all loads, $VBo+Mig$ provides a consistent and compelling improvement of 20%. In some lightly-loaded systems, it is outperformed by $VBoost$. This is due to an artifact of our thread controller implementation: for simplicity, we set the $f$ of both cores in the $VBo+Mig$ pair to be that of the slower core in the pair. This makes it slightly slower than $VBoost$ under best-case conditions. Finally, $VBo+Pl$ performs much better than either technique alone — especially as the number of available cores increases. When all 16 cores are available, $VBo+Pl$’s average performance improvement reaches 38%.

To explain these performance trends, Figure 7.2 shows the limiting constraints for each technique and loading level. The size of each bar segment represents the fraction of samples (one for each of the 50 dies and 11 benchmarks, for a total of 550 samples) from the pseudo-oracular experiments in which the particular constraint was the limiting one. The constraints can be $P_{E\text{max}}$, $P_{\text{max}}$, $T_{\text{max}}$, or $V_{dd\text{max}}$. The number above each bar is the average power consumption of an $S$ (performance-enhanced) thread under that configuration.

The three regimes of Table 4.1 appear in the figure as follows. When $Paceline$ is limited by $P_{\text{max}}$ or $T_{\text{max}}$, the system is in the Individual regime. There is insufficient headroom to apply two techniques simultaneously. This regime is seen rarely in Figure 7.2 when only two cores are available. When $Paceline$ is limited by $P_{E\text{max}}$ and $VBo+Pl$ is mostly limited by $P_{\text{max}}$ or $T_{\text{max}}$, the system is in the Synergistic regime. This is the dominant regime for 2-8 available cores. Finally, when $VBo+Pl$ is often limited by $P_{E\text{max}}$ or $V_{dd\text{max}}$, it is Unfulfilled. This is commonly seen for 16 available cores.

More specifically, for the fully-loaded system (one core available), $T_{\text{max}}$ is the key constraint. The high total power consumption of all cores on chip raises the heatsink temperature, reducing thermal headroom. Consequently, $VBoost$ can only increase voltage slightly before hitting the temperature limit. The result is that $VBoost$ is not able to provide large performance improvements on a fully-loaded system (Figure 7.1).

When the system has two available cores, $VBoost$ remains temperature limited, able to dissipate only 9W on average. $VBo+Mig$ addresses the thermal problem effectively, allowing the
Figure 7.2: Relative incidence of limiting constraints across all applications and die samples. The numbers above the bars show the average power consumption of the enhanced thread on each system.

Thread to dissipate 12W on average and reach the supply voltage limit in many cases. Referring to Figure 7.1, this provides substantial marginal performance gains over VBoost. Meanwhile, Paceline opens a new avenue for performance enhancement: Paceline is limited by $P_{E_{\text{max}}}$ and has ample room to grow in power and temperature. In other words, Paceline provides an orthogonal means of increasing performance. When VBoost and Paceline are used together in VBo+Pl, they work in the Synergistic regime. They attain substantial improvements (Figure 7.1) and are bounded mainly by temperature.

The cases of four to eight available cores show that Paceline is usually limited by $P_{E_{\text{max}}}$ and VBo+Mig is often limited by the maximum supply voltage, but by applying VBo+Pl, we usually arrive at the power or temperature limits. This is still the Synergistic regime.

In lightly loaded systems (16 available cores), VBoost and VBo+Mig are practically supply voltage limited, while Paceline is $P_{E_{\text{max}}}$-limited. After combining them in VBo+Pl, the system
is still often limited by supply voltage. Therefore, the system works in *Unfulfilled* regime. It has left-over power and temperature headroom to accommodate other techniques that increase per-core power.

Overall, the key benefit of combining *Paceline* with *VBoost* is that they provide orthogonal means of improving performance because they have different limiting constraints. This leads to a *Synergistic* regime (2-8 available cores) and an *Unfulfilled* one (16 available cores).

### 7.2.2 Power–Performance Tradeoffs

To fully assess the different techniques, Figure 7.3 shows the performance and power consumption of an *S* thread under the different techniques and a range of loading conditions. The performance is normalized to the *Unoptimized* environment. In the figure, each curve corresponds to one technique. The diamond point on each curve represents an unloaded system with 16 available cores, while the circle (or the triangle in the case of the *Paceline*, *VBo+Mig*, and *VBo+Pl*), represents a heavily loaded system. Other markers on each curve identify the remaining loading points from Figures 7.1 and 7.2.

This plot clearly shows the *composability* of the *Paceline* and *VBoost* (or *VBo+Mig*) techniques, and the great *capability* of the *VBo+Pl* technique. Indeed, the figure shows that, roughly speaking, *Paceline*, *VBoost*, or *VBo+Mig* deliver at best a 20% improvement in single-thread performance at the cost of doubling the power of the thread. However, *VBo+Pl* delivers up to 38% improvement in single-thread performance at the cost of trebling the thread power. This is a remarkably high performance improvement. It is unreachable with either base technique alone, and shows that the two base techniques are orthogonal. For the arguably more likely case when only 8 cores are available, *VBo+Pl* improves single-thread performance by 34% while consuming 220% more power than *Unoptimized*.

The figure also shows that *VBoost* (and *VBo+Mig*) are more power-efficient than *Paceline*. At approximately the same power, *VBoost* and *VBo+Mig* deliver higher performance than *Paceline*.

Finally, we note that power consumption does not increase monotonically with performance.
Figure 7.3: Per-thread power consumption vs performance for each technique under different load conditions.

For most techniques, power decreases slightly for the highest performance point — namely, the one with the lowest load. This is due to the decrease in leakage power at the lower die temperatures that prevail under light load. When most of the cores on the die are idle, the total chip power dissipation is low, so the heatsink is cold. As a result, all points on the die surface decrease in temperature, and leakage goes down.

### 7.2.3 Sensitivity Analysis

We now characterize the sensitivity of the performance improvements of VBoost and Paceline. In many cases, $T_{\text{max}}$ is a limiting factor, so we examine different thermal design points. Additionally, the Paceline systems achieve their performance improvements primarily by consuming
guardband, so we evaluate several different guardband sizes. Finally, we consider different per-core power limits that correspond to more robust and more frail per-core power grids. Figure 7.4 shows the three sensitivity studies varying the parameters that have been assumed for the preceding experiments (labeled as “default”). All numbers are normalized to the performance of Un-optimized with the same parameters under the same conditions. Note that the default-parameter bars are not exactly like those in Figure 7.1 because here, we only simulate a single (typical) die, rather than 50 samples. In the following, we consider each plot in turn.

Figure 7.4: Sensitivity of performance improvements to the thermal environment (a), process guardband (b), and power grid design (c).

Power/Thermal Envelope

We consider three different power/thermal environments, namely laptop, desktop, and server, as shown in Table 7.2. Configurations vary the total number of cores on the chip to fit into their respective thermal envelopes — 4 cores in laptop, 16 cores in desktop, and 24 cores in server. Additionally, server assumes a powerful high-airflow cooling system typical of datacenter installations, while laptop is constrained by a relatively weak heatsink due to its physical confines.

Figure 7.4(a) shows that the performance trends are largely identical in all three thermal environments. On an unloaded system, the performance gains for each technique differ by less than 3%.

Note: Laptop, desktop, and server heatsinks are modeled on Aavid Thermalloy types 3680, 1002/D, and 037704, respectively.
<table>
<thead>
<tr>
<th></th>
<th>Laptop</th>
<th>Desktop</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cores</td>
<td>4</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>TDP</td>
<td>24W</td>
<td>96W</td>
<td>144W</td>
</tr>
<tr>
<td>Heatsink $R_{th}$</td>
<td>1.3 K/W</td>
<td>0.33 K/W</td>
<td>0.30 K/W</td>
</tr>
<tr>
<td>Ambient $T$</td>
<td>313K</td>
<td>313K</td>
<td>305K</td>
</tr>
</tbody>
</table>

Table 7.2: Power/thermal environments for the sensitivity study.

2% across all environments. Moreover, for all environments, performance falls off at roughly the same rate as system load increases.

**Guardband**

Figure 7.4(b) shows that, as the process guardband increases from 10% to 20%, the effectiveness of Paceline also grows. This is because Paceline gets much of its speedup from removing the guardband (refer to Figure 4.1(b)). As the guardband increases from 10% to 15%, the returns from Paceline grow by roughly 0.04. However, if the guardband increases above 15%, the performance benefits from Paceline rise at a slower rate — tempered by the application’s responsiveness to $f$ increases and the limited bandwidth of the checker core.

**Power Grid Capacity**

Figure 7.4(c) shows how increasing the per-core power constraint $P_{max}$ impacts performance. From a design perspective, supporting a higher $P_{max}$ requires a more robust per-core supply grid with lower resistance, higher decoupling capacitance, and lower inductance, which can be difficult to achieve. The default design assumed a grid capable of supplying a worst-case power equal to twice the average power consumption of the SPECint applications, which we believe represents a typical design. Decreasing that budget by one third to arrive at $P_{max} = 8$W degrades all techniques except Paceline, but they still provide compelling speedups. Increasing the budget by one third to 18W benefits only the most power-hungry cases and even then, only slightly. We conclude that none of the techniques requires more than a standard power distribution system.
7.2.4 Configurability

The previous evaluations focused on the goal of optimizing a single thread, but when executing a multiprogrammed workload or an application with some, limited parallelism, it will often be necessary to speed up a larger number $S$ of optimized, performance-critical threads (Section 4.2.3). Here, the key question is how much performance can be delivered to the $S$ optimized threads while the chip concurrently runs a specified number $R$ of unoptimized threads. Note that Paceline, VBo+Mig, and VBo+Pl all use two cores to optimize a single $S$ thread. Therefore, they can execute at most $S = 8$ optimized threads when the system is otherwise unloaded. Only VBoost is able to optimize $S > 8$ threads.

Figure 7.5 shows the percentages of performance improvement experienced by an $S$ thread for each technique when the application demands different numbers of $R$ and $S$ threads. Plots (a)-(d) correspond to one technique each, while plot (e) corresponds to the algorithm used by the global controller (Section 4.3.3), namely choose VBo+Pl if there are twice as many idle cores as $S$ threads, or VBoost otherwise. Each plot shows iso-performance contours for the $S$ threads in increments of 2%. Performance is normalized to that of an Unoptimized chip running the same number of threads. For techniques that use two cores per $S$ thread (plots (b), (c), and (d)), the region where $R + 2S > 16$ is infeasible; for VBoost (plot (a)), the region where $R + S > 16$ is infeasible. Finally, plot (e) is an “overlay” of plot (d) when VBo+Pl is feasible and plot (a) otherwise.

For their feasible configurations, both VBo+Mig and Paceline offer consistent per-thread performance improvement, varying by less than 2% over the entire feasible range of $S$ and $R$. When available, VBo+Pl always offers superior performance improvements to either technique alone. However, that performance is more sensitive to the particular value of $S$ and $R$ demanded. Likewise, the performance of VBoost falls off as the number of $S$ threads approaches the limit.

Plot (e) demonstrates that a configurable multicore benefits greatly from having both VBoost and VBo+Pl available. A chip with only VBoost realizes the full performance gains when the sys-
Figure 7.5: Percentages of performance improvements experienced by an $S$ thread when the application demands different numbers of $R$ and $S$ threads. For each technique, there is an area of infeasible configurations. Iso-performance contours are shown in the feasible regions.

A system must run many $S$ and $R$ threads at the same time (darker region), but it is not able to provide optimal improvements when the number of $S$ threads is low or the load is light (light region).

Likewise, a system with only Paceline would provide less performance increase in the light region and would be helpless in the dark region.
Chapter 8

Related Work

8.1 Timing Speculation

Timing speculation is a relatively recent development, originating with TIMERRTOL [75] less than a decade ago. Since then, many approaches and refinements have been proposed with the goal of either saving power or enabling higher clock frequencies, but Paceline and BlueShift each fill important gaps in this prior work: Paceline is the first proposal to be Configurable in the sense of Section 1.3 and BlueShift provides much-needed CAD support, which has only been addressed in a couple of previous proposals and never in a way that applies to such a wide range of microarchitectures. The following subsections review the previous proposals from a microarchitecture and CAD perspective, contrasting them with Paceline and BlueShift.

8.1.1 TS Microarchitectures

Stage-Level Checking

The original TS proposal, TIMERRTOL [75] was less of a microarchitecture than a general-purpose pipeline latching scheme. It proposed three ways of using dual-phase clocking to detect and correct timing faults at the register level. The most efficient of the schemes works by doubling the pipeline depth, inserting an additional pipeline register in the middle of each combinational logic block. Adjacent registers are then clocked 180 degrees out of phase and at higher frequency than the original circuit. Errors are detected by comparing each register’s input to its output just before the clock edge of the preceding register. In the event of a miscomparison, indi-
cating a timing fault, a global recovery signal is asserted and the contents of the preceding register are used to restore the pipeline state.

TIMERRTOL’s main advantage is that it does not require any change in the design of the combinational logic except aside from inserting the extra pipeline registers. For example, the designer does not need to ensure longer than normal hold times, unlike in Razor [20]. However, TIMERRTOL does add one register delay to the critical path of each pipeline stage, which reduces clock frequency substantially. Nevertheless, Uht et al. successfully demonstrated the technique on a 32-bit ripple-carry adder, achieving a 70% throughput increase over a baseline, non-speculative design.

Constructive Timing Violation (CTV) [59] and Circuit Level Speculation (CLS) [42] are independently-developed proposals similar to TIMERRTOL that evaluate the performance and power impacts of TS on an out-of-order processor pipeline. At the circuit level, their technique is identical to one of TIMERRTOL’s proposed schemes, which requires three identical copies of each speculative logic block. One copy runs at the full pipeline frequency and produces speculative results. Two other “checker” copies run at half the pipeline frequency. CLS showed that using this technique on the adder unit — along with other optimizations — could decrease processor cycle time. Moreover, it simplified the error recovery procedure by using the pipeline’s built-in speculation support, simply re-issuing errant instructions. CTV focused on power reduction by reducing supply voltage. It argued that although replicating functional units may seem wasteful, this type of TS can actually improve power efficiency if the error rate is low enough to allow a significant reduction in supply voltage.

Razor [17, 20] is another scheme that performs checking at each pipeline register, but unlike its predecessors, it relies on wave-pipelining [15] to detect and correct errors. It augments each pipeline register with a “shadow” register. Both the shadow and main register accept the same data input, but the shadow is clocked 180 degrees out of phase with the main one. A minimum hold time of one half cycle is imposed on all logic paths in the design to prevent values from “racing through” to the shadow latch. Thus, the value latched by the shadow should be identical
to the value in the main latch but has had a half cycle longer to propagate through the logic and is therefore nonspeculative. A comparator monitors the contents of the shadow and main registers, and if they disagree, the shadow contents are used to reinitialize the pipeline.

Perhaps Razor’s most important contribution is its rigorous evaluation, which includes a real silicon implementation of a full Alpha processor [17]. The goal of the evaluation was to demonstrate power savings from reduced supply voltage, and this was successful. Razor can also increase frequency at nominal voltage, and this dissertation has focused on that use.

Note that none of these microarchitectures provides configurability. It is true that any TS support can be “turned off” simply by powering down the checker logic and registers, but substantial overhead remains relative to a non-TS design: First, the area overhead of the additional registers and checking logic reduces core count on a CMP. In the case of CLS [42] and CTV [59], the replicated functional units increase power consumption by lengthening bypass paths. The most advanced TIMERRTOL scheme adds to the critical path delay, which increases delay when running at a nonspeculative frequency. Finally, Razor imposes a power overhead by requiring extra buffers to guarantee the half-cycle hold time.

**At-Retirement Checking**

DIVA [7] was developed to simplify processor design and provide tolerance of permanent and transient errors. Although TS was not one of its design goals, timing faults are similar to the soft errors the microarchitecture was designed to correct. DIVA works by augmenting the main pipeline with a checker unit that re-executes every stage (fetch, decode, execute, and writeback) of each instruction before retirement. The checker keeps up with the main pipeline by performing each stage of its checking in parallel after the instruction completes in the main pipeline. DIVA thus isolates the checking logic into at-retirement module that is off the main pipeline’s critical path. Additionally, DIVA’s ability to correct permanent faults allows the main pipeline to relax functional correctness.

Optimistic Tandem [47] is a proposal that builds on DIVA by embracing functional errors in
the main pipeline for the purpose of maximizing performance under TS. Specifically, it designs the main pipeline so that the common cases are fast and uncommon cases are incorrect. It then integrates a checker core that — like DIVA — is much smaller and less powerful than the main core. The way in which the checker is integrated with the main core, however, is closer to Paceline. As in Paceline, the leader passes branch results to the checker and prefetches data into a shared L2 cache. Meanwhile, the two cores periodically compare execution signatures to detect faults. Unlike Paceline, however, the leader and checker are microarchitecturally different. Consequently, Optimistic Tandem does not perform activity migration. The method of branch passing also differs slightly from Paceline, as Optimistic Tandem [47] allows the leader core to train the checker’s branch predictor directly. This provides an interesting and possibly lower-overhead alternative to Paceline’s BQ.

Unfortunately, these at-retirement checkers also lack configurability. TS can never be disabled in Optimistic Tandem because the main pipeline is not functionally correct. DIVA provides more opportunity for configurability, as the checker could be disconnected and used independently as a low-performance processor when TS is disabled. However, this introduces heterogeneity into the design, which may not be desirable.

8.1.2 TS Design Methodologies

Several proposals have analyzed or improved the performance of specific functional blocks — usually adders — under TS [25] [8]. Others have also pointed out an unfulfilled need for general-purpose design flows that optimize performance under TS [8]. Of existing work, Optimistic Tandem [47] (which inspired BlueShift) is the most closely related. Like BlueShift, it uses a profile-based approach, running training benchmarks on an RTL simulation of the system and recording which source RTL statements are only rarely used. However, unlike BlueShift, it achieves its speedup by sacrificing functional correctness, pruning the infrequently-used statements from the design of the main pipeline. The fact that it operates on the RTL (rather than the gate-level design that BlueShift requires) is an advantage of Optimistic Tandem, offering much faster pro-
filing speed. However, because it is based on functional pruning, the Optimistic Tandem design methodology can not be used to build configurable systems.

BTWMap \[37\] is a general design tool that optimizes common-case performance without sacrificing functional correctness by implementing a novel gate-mapping algorithm (part of the logic synthesis flow) that uses a switching activity profile to minimize the common-case delay. BTWMap requires modification of the CAD algorithms, but it is applicable to any synthesis-based design flow. The main disadvantage is that it does not account for physical (layout, routing, and clock skew) effects, which can significantly affect circuit timing. Because BTWMap operates at an early stage of the design flow, many of its gains may be lost in the physical design backend if those tools re-factor the design using more traditional objectives. Integrating BTWMap with the rest of the design flow is therefore not trivial. In contrast, BlueShift sacrifices some ability to do aggressive optimization by not intervening in the synthesis stage but controls the physical design (place and route) to ensure that the final design is suitable for TS.

8.2 Configurable Microarchitectures

8.2.1 Leader-Checker Execution

PIPE \[24\] was the first configurable architecture to apply two cores to the execution of a single thread. In PIPE, two cores — called the Access Processor (AP) and Execution Processor (EP) — execute different instruction streams generated by a specialized compiler. One stream performs all of the memory operations (and their associated slices) and executes on the AP. The other stream executes the non-memory instructions on the EP. The effect is that the AP can execute ahead, buffering its load values in a special load queue for later consumption by the EP, thereby reducing stalls and increasing execution bandwidth.

Slipstream \[53, 70\] was another seminal leader–checker proposal, and Paceline is heavily based on its microarchitecture\[1\] Like Paceline, Slipstream uses two cores in coupled leader–

\[1\]Even the names are similar, both referring to the same aerodynamic effect.
checker mode to boost per-thread performance. The leader generates prefetches and branch results for the checker just as in Paceline. The key difference is the way in which the leader achieves its speedup. Slipstream is actually more clever than Paceline in this respect, speculatively eliding dynamically dead code, highly predictable branches, and synchronization operations from the leader’s instruction stream. In fact, Slipstream could perform TS on top of this optimization to realize even higher performance than Paceline. Slipstream also argued for configurability, showing that even for the highly-parallel SPLASH-2 benchmarks, a 32-processor system running 16 threads in coupled leader–checker mode usually outperforms the same system running twice as many threads in uncoupled mode [30].

Master/Slave Speculative Parallelization (MSSP) [81] took instruction elision a step further by creating an aggressively reduced “distilled” program binary to run on a Master (leader) processor while employing multiple checker processors. A special MSSP compiler partitions the program into a sequence of tasks and generates both full and distilled implementations of each. As the distilled tasks complete on the Master, they distribute predicted live-in values to Slave processors, which then execute the full implementation of the successor tasks starting from those live-ins. Slaves keep up with the Master by checking in parallel, with each Slave core executing one task at a time. A Slave commits its results to architectural state in-order after verifying the live-ins from the preceding task.

More conservatively, other proposals such as Dual-Core Execution [80] and Flea-Flicker [10] pipelining have focused exclusively on prefetching without using compiler support to alter the leader’s program binary. In these proposals, the leader core fetches and executes instructions without stalling for long-latency cache misses. Instead, it executes what it can and passes the complete stream of dynamic instructions to the other core, which executes the full program and integrates the missing loads. Future Execution [21] is similar except that instead of skipping cache-missing loads, it attempts to predict the values that instructions will produce in future loop iterations, thereby allowing it to run several iterations ahead of the checker.

Aside from performance enhancements, the other major application of leader–checker archi-
tectures has been in reliability and especially soft-error tolerance. For example, CRTR [23] redundantly executes one thread on two cores with the leader providing branch outcomes, load values and register results to the checker. The checker re-executes all instructions using the values in the queues for comparison. Madan and Balasubramonian [43] refine this approach by running the checker core at a lower frequency and voltage to provide energy savings. Reunion [64] is yet another high reliability leader–checker microarchitecture. Unlike the others, but like Paceline, it uses checkpoint signatures or fingerprints [65] to compare the leader and checker executions.

### 8.2.2 Voltage-Frequency Boosting

Many previous works have used dynamic voltage and frequency scaling (DVFS [14]) to save power by reducing voltage and frequency below nominal. However, the idea of increasing voltage and frequency above nominal (when the power and thermal budget allows) is relatively recent. Intel’s currently shipping Turbo Boost [34, 5] technology and its predecessor, Foxton [46], are the best examples. They both include on-die controllers that continuously monitor the power consumption and temperature of the cores, increasing voltage and frequency up to an aggressive maximum value when constraints allow. LeadOut (Chapter 4) uses a similar control algorithm.

The combination of voltage-frequency boosting and TS in LeadOut is also not novel. EVAL [58] proposed trading off voltage, error rate, and power to maximize core performance under variation. As part of this tradeoff, it includes the ability to increase voltage above nominal. Unlike the Turbo and Foxton controllers, EVAL uses a fuzzy control system to co-optimize supply voltages (and body-bias settings) for every functional unit in the processor, with each unit potentially receiving a different voltage. Unlike LeadOut, EVAL’s checker architecture is based on DIVA [7] and is therefore not configurable.
8.2.3 Other Proposals

Thread-Level Speculation (TLS) \[67, 39, 54\] provides another configurable way for multiple cores to work together on one thread. In TLS, a sequential program is partitioned into tasks that speculatively execute concurrently on multiple cores. As it executes, each task watches for data dependences from preceding tasks in program order. Whenever the system detects that a data dependence was not honored in the parallel execution, it squashes any tasks that might have been affected. In the absence of such dependence violations, tasks first buffer their updates and then commit them atomically when all earlier tasks have completed. TLS offers the advantage that, like MSSP, it can apply an arbitrary number of cores to the execution of a single thread. The main disadvantage is the modification of the caches needed to detect memory dependences between tasks and to hold speculative data until a task commits.

Core Fusion \[35\] takes a much finer-grain approach to configurability. It lays out multiple dual-issue out-of-order pipelines side-by-side on the die and allows adjacent pipelines to fuse together into a single wider-issue processor. This approach allows an arbitrary number of cores to work together, with the benefit of adding more cores limited only by the application’s ILP. However, it requires the core pipeline to be redesigned from the ground up to incorporate the extra multiplexers and control logic that orchestrate fetch, bypassing, and commit in the fused mode. The hardware overhead is estimated at 8% of the core area.

TFlex \[36\] is another finer-grained approach to configurability. Like its predecessor, TRIPS \[56\], it is an explicit dataflow architecture that distributes operators across a fabric of small execution tiles. These tiles exchange operands (dataflow tokens) under the explicit control of the compiler. Since the tiles are only loosely coupled in the microarchitecture, they can easily be spatially partitioned to allow different threads to run on different segments of the fabric. When per-thread performance is required, the dataflow operators spread out across all of the tiles to maximize ILP, and when TLP is demanded, more operators are placed on a single tile to make room for more threads.
Chapter 9

Conclusion

9.1 Summary of Contributions

The future of performance scaling lies in massively parallel workloads, but less-parallel applications will remain important. Unfortunately, future process technologies and core microarchitectures no longer promise major per-thread performance improvements, so microarchitects must find other ways to address a growing per-thread performance deficit. Moreover, they must do so without sacrificing parallel throughput. To meet these apparently conflicting demands, this dissertation joined with previous works in advocating a configurable multicore. It contributed a novel configurable enhancement based on timing speculation, which can compose with other configurable enhancements to provide large per-thread performance gains. Our example LeadOut multicore achieved mean speedups of 34% on SPECint by combining timing speculation with voltage-frequency boosting.

This dissertation made three interlocking contributions to the design of configurable multicores at the microarchitecture and circuit levels. First, Chapter 2 introduced Paceline, the first TS microarchitecture designed specifically for configurable multicores. Next, it enhanced Paceline with BlueShift, a circuit design method for TS architectures that aims to improve the circuit’s common-case delay rather than focusing on worst-case delay as traditional design flows do. Finally, it introduced LeadOut, a multicore design that combines Paceline with the ability to increase core supply voltage above nominal. It showed major gains from applying the two techniques together when feasible and argued that, in many cases, future multicores can benefit from composing still more configurable enhancements.
9.2 Looking Forward

This dissertation’s contributions were aimed at homogeneous multicores, assuming a scaling model in which the number of cores grows exponentially as technology advances. However, technology experts are now expressing growing concern that power scaling may not support this model. Just as Chapter 1 explained that power density limits future frequency increases, it is possible that even if clock frequency does not scale at all, the power problem could become so acute that it will limit the number of cores that can be simultaneously powered on. This leads to a case where most of the cores on a die must be idle at any given time in order to meet the power and thermal budget. The likely architectural consequence is extreme heterogeneity, where the die includes many customized cores and application-specific accelerators, only a few of which — the best for the phase at hand — are active at any given time.

Such a design is less appealing than a homogeneous one from a design effort and time-to-market perspective, but it may be the only way forward. It is important to note that all of the techniques proposed in this dissertation still apply in a heterogeneous environment; configurability is still valuable within a given class of cores — especially the highest-performance cores where Paceline, BlueShift, and LeadOut can be used to boost per-thread performance to even higher levels than would otherwise be achievable even using arbitrarily complex cores.
References


[34] Intel Corporation. Intel Turbo Boost technology in Intel Core microarchitecture (Nehalem) based processors, November 2008.


Author’s Biography

Brian was born in 1980 and entered directly into the engineering profession. He received his first tools (a screwdriver and hammer) at the age of two and used them on anything within reach. In third grade, he met Wayne Weise, a Bell Labs engineer who introduced him to electronics design and nurtured the hobby by supplying hand-written lessons and parts kits. Through junior high and highschool, Brian tinkered with electronics and created many one-off gadgets.

He obtained a B.S. in Computer Engineering from Clemson University, where he also received an introduction to research under Prof. Ron Sass. Inspired by this experience, he joined the University of Illinois and completed an M.S. in Electrical Engineering and a Ph.D in Computer Science. His research has covered FPGA-based computing, modeling and mitigating process variation, simplifying processor design and verification, and improving per-thread performance through timing speculation.

After receiving his Ph.D, he joined D. E. Shaw Research to assist in high-speed molecular dynamics simulation for biochemistry. Although research papers have been his main product for the last six years, he still has an oscilloscope on his desk and knows how to use it.