MANAGING RESOURCES FOR HIGH PERFORMANCE AND LOW ENERGY IN GENERAL-PURPOSE PROCESSORS

A Dissertation Presented
by
HUAPING WANG

Submitted to the Graduate School of the University of Massachusetts Amherst in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2010

Electrical and Computer Engineering
MANAGING RESOURCES FOR HIGH PERFORMANCE AND LOW ENERGY IN GENERAL-PURPOSE PROCESSORS

A Dissertation Presented

by

HUAPING WANG

Approved as to style and content by:

__________________________
Israel Koren, Chair

__________________________
C. Mani Krishna, Member

__________________________
Sandip Kundu, Member

__________________________
Charles C. Weems, Member

__________________________
Christopher V. Hollot, Department Chair
Electrical and Computer Engineering
ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor Prof. Israel Koren for his invaluable guidance, many helpful discussions and attentive reading of many drafts. Prof. Koren is an outstanding advisor who provides not only insightful guidance on research but also extensive moral support. He kept encouraging me with great patience and kept my morale high when depression seemed the only choice.

I would like to thank Professors Israel Koren, C. Mani Krishna, Sandip Kundu and Charles C. Weems for serving in my committee. I thank them for their thoughtful suggestions and comments through my preparation of the proposal to the final defense and revision of this dissertation.

I also would like to thank Prof. C. Mani Krishna and Dr. Zahava Koren for their help during my research at UMass. The stimulating discussions and constructive suggestions have helped my research work and this dissertation greatly. Special thanks to Prof. Krishna for helping me modifying paper drafts with great patience.

Furthermore, I would like to thank my colleagues and friends in my lab (ARTS lab) for their helpful discussions in research and life. Their suggestions and support really helped me much when my morale was not high during my research.

Finally, I am deeply indebted to my parents and my husband for their patience, encouragement, love and help in all aspects of my personal life.
ABSTRACT

MANAGING RESOURCES FOR HIGH PERFORMANCE AND LOW ENERGY IN GENERAL-PURPOSE PROCESSORS

MAY 2010

HUAPING WANG

B.Sc., ZHEJIANG UNIVERSITY
M.Sc., ZHEJIANG UNIVERSITY
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren

Microarchitectural techniques, such as superscalar instruction issue, Out-Of-Order instruction execution (OOO), Simultaneous Multi-Threading (SMT) and Chip Multi-Processing (CMP), improve processor performance dramatically. However, as processor design becomes more and more complicated, how to manage the abundant processor resources to achieve optimal performance and power consumption of processors becomes increasingly more sophisticated. This dissertation investigates resource usage controlling techniques for general-purpose microprocessors (supporting both single hardware context and multiple hardware contexts) targeting both energy and performance.

We address the power-inefficient resource usage issue in single-context processors and propose a Compiler-based Adaptive Fetch Throttling (CAFT) technique which combines the benefits of a hardware-based runtime throttling technique and a
software-based static throttling technique providing good energy savings with a low performance loss. Our simulation results show that the proposed technique doubles the energy-delay product (EDP) savings compared to the fixed threshold throttling.

We introduce the resource competing problem for SMT processors, which allow multiple threads to simultaneously share processor resources and improve the energy-efficiency indirectly by resource sharing. We present a novel Adaptive Resource Partitioning Algorithm (ARPA) to control the usage and sharing of processor resources in SMT processors. ARPA analyzes the resource usage efficiency of each thread in a time period and assigns more resources to threads which can use them in a more efficient way. Simulation results on a large set of 42 multiprogrammed workloads show that ARPA outperforms the currently best dynamic resource allocation technique, Hill-climbing, by 5.7% with regard to the overall instruction throughput. Considering fairness accorded to each thread, ARPA attains 9.2% improvements over Hill-climbing, using a commonly used fairness metric.

We also propose resource adaptation approaches to adaptively control the number of powered-on ROB entries and partition shared resources among threads for both shared-ROB and divided-ROB structures, targeting both high performance and low energy. Our resource adaptation algorithms approaches consider not only the relative resource usage efficiency of each thread like ARPA, but also take into account the real resource usage of threads to identify cases of inefficient resource usage behavior and save energy. Our experimental results show that for an SMT processor with a shared-ROB structure, our resource adaptation approach achieves 16.7% energy savings over ARPA, while the performance loss is negligible across 42 sample workloads. For an SMT processor with a divided-ROB structure, our resource adaptation approach outperforms ARPA by 4.2% in addition to achieving 12.4% energy savings.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>x</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
</tbody>
</table>

## CHAPTER

1. INTRODUCTION ..................................................... 1
   1.1 The Needs of Resource Management ..................... 1
   1.2 Two Categories of Techniques ......................... 2
       1.2.1 Architecture Adaptation .......................... 3
       1.2.2 Resource Sharing among Multiple Threads .......... 4
   1.3 Related work ............................................. 6
   1.4 Thesis Contributions ................................. 10
   1.5 Organization of the Thesis ........................... 12

2. FETCH THROTTLING IN SINGLE-THREAD PROCESSORS .......... 14
   2.1 Introduction ........................................... 15
   2.2 CAFT: Compiler-based Adaptive Fetch Throttling .... 17
       2.2.1 Compiler-based IPC Estimation .................... 17
       2.2.2 Adaptive Fetch Throttling ........................ 18
           2.2.2.1 The Algorithm ................................ 20
           2.2.2.2 Architecture-level Implementation ............ 21
       2.2.3 Hardware-based Fetch Throttling .............. 22
2.3 Experimental Setup .................................................. 24
2.4 Results and Analysis .................................................. 25
  2.4.1 IPC Distribution .................................................. 25
  2.4.2 Execution Time and Energy .................................... 27
  2.4.3 EDP/ED^2P ....................................................... 30
2.5 Chapter Summary ..................................................... 31

3. DYNAMIC RESOURCE SHARING IN SIMULTANEOUS
   MULTITHREADING (SMT) PROCESSORS ....................... 32
  3.1 Introduction ....................................................... 33
  3.2 ARPA: Adaptive Resource Partitioning Algorithm .......... 35
    3.2.1 Framework ................................................... 35
    3.2.2 Resource Utilization Analysis ............................. 35
      3.2.2.1 Metric of Usage Efficiency .............................. 36
      3.2.2.2 Partitioning Process ................................. 36
    3.2.3 Partitioning Algorithm .................................... 39
    3.2.4 Implementation of ARPA .................................... 40
  3.3 Experimental Setup ................................................ 41
    3.3.1 Configuration ................................................ 41
    3.3.2 Workloads .................................................... 42
    3.3.3 SMT Performance Metrics ................................... 43
  3.4 Results and Analysis ............................................. 45
    3.4.1 Adaptivity of ARPA .......................................... 45
    3.4.2 Performance Results ....................................... 48
    3.4.3 Sensitivity Analysis ....................................... 53
      3.4.3.1 Epoch and Step Size .................................. 53
      3.4.3.2 Queue Entries ......................................... 54
  3.5 Chapter Summary .................................................. 56

4. RESOURCE ADAPTATION FOR HIGH PERFORMANCE
   AND LOW ENERGY IN SMT PROCESSORS ......................... 57
  4.1 Introduction ....................................................... 58
  4.2 Resource Utility Analysis ....................................... 60
  4.3 Resource Adaptation ............................................. 61
4.3.1 Multi-banked Structure .................................................. 61
4.3.2 Resource Adaptation for Shared-ROB Structure .................... 65
4.3.3 Resource Adaptation for Divided-ROB Structure .................... 68
4.3.4 Implementation of Resource Adaptation ............................... 70

4.4 Evaluation Methodology .................................................... 71
4.5 Results and Analysis ........................................................ 72

4.5.1 Shared-ROB Structure ................................................. 72
   4.5.1.1 Performance Evaluation ........................................... 72
   4.5.1.2 Energy Savings ..................................................... 73
   4.5.1.3 Energy-to-Single_WIPC-Ratio Improvements .................... 75
   4.5.1.4 Adaptation Process ............................................... 77
   4.5.1.5 Sensitivity Analysis .............................................. 78

4.5.2 Divided-ROB Structure ................................................. 80
   4.5.2.1 Performance Evaluation ........................................... 81
   4.5.2.2 Energy Savings ..................................................... 81
   4.5.2.3 Energy-to-Single_WIPC-Ratio Improvements .................... 82

4.6 Chapter Summary .......................................................... 83

5. SUMMARY AND FUTURE RESEARCH ................................. 85

5.1 Conclusion ............................................................... 85
5.2 Future Work ............................................................... 85

BIBLIOGRAPHY ................................................................. 88
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Baseline parameters</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>SMT simulator settings</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Twenty two SPEC CPU2000 benchmarks used in this study.</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>Benchmark combinations based on cache behavior of threads.</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>The movements of pointers from the current bank to the Next Bank (NB)</td>
<td>64</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Comparison of issue slot for different architectures</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Hardware implementation for CAFT</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Normalized throttling cycles</td>
<td>26</td>
</tr>
<tr>
<td>2.3 IPC distribution when throttling using CAFT</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Normalized execution time and energy</td>
<td>27</td>
</tr>
<tr>
<td>2.5 Reduction of miss-fetched instructions relative to non-throttling case</td>
<td>28</td>
</tr>
<tr>
<td>2.6 Fetch width distribution (averaged across all benchmarks)</td>
<td>29</td>
</tr>
<tr>
<td>2.7 Energy-delay product and energy-delay² product savings</td>
<td>30</td>
</tr>
<tr>
<td>3.1 A high-level description of the ARPA algorithm</td>
<td>35</td>
</tr>
<tr>
<td>3.2 An example illustrating the CIPRE changes after epoch $n + 1$</td>
<td>37</td>
</tr>
<tr>
<td>3.3 ARPA pseudo-code</td>
<td>39</td>
</tr>
<tr>
<td>3.4 An implementation of ARPA</td>
<td>40</td>
</tr>
<tr>
<td>3.5 An example (twolf-apsi) illustrating the adaptive nature of ARPA over a short time period.</td>
<td>46</td>
</tr>
<tr>
<td>3.6 An example (twolf-apsi) illustrating the adaptive nature of ARPA over a long time period.</td>
<td>46</td>
</tr>
<tr>
<td>3.7 Two additional examples illustrating the adaptive behavior of ARPA</td>
<td>48</td>
</tr>
<tr>
<td>3.8 Single_WIPC of different schemes for 42 workloads.</td>
<td>49</td>
</tr>
</tbody>
</table>
3.9 Avg_IPC and Single_WIPC improvement of different schemes over ICOUNT across 42 workloads ........................................ 50

3.10 Baseline_WIPC of different schemes for 42 workloads. ................. 52

3.11 Avg_IPC improvement of ARPA over ICOUNT as epoch and step sizes change for the MIX.4.4 and ILP.4.5 workloads. ............... 53

3.12 Avg_IPC of different schemes as the number of queue entries changes for 12 out of the 42 workloads in Table 3.3. ................................. 54

3.13 Single_WIPC improvement of different schemes over ICOUNT as the number of queue entries changes for 12 out of the 42 workloads in Table 3.3. .................................................................. 55

4.1 The normalized Weighted_IPC and Energy as a function of ROB size for two workloads ................................................................. 59

4.2 Representative locations of allocated and deallocated banks. .......... 62

4.3 State machine of each bank ............................................................... 64

4.4 Pseudo-code of resource adaptation for an SMT processor with a shared-ROB structure ................................................................. 67

4.5 Pseudo-code of resource adaptation for an SMT processor with a divided-ROB structure ................................................................. 69

4.6 Implementing Resource Adaptation .................................................. 70

4.7 Single_WIPC of different schemes for 42 workloads normalized with respect to ICOUNT for an SMT processor with a shared-ROB structure ........................................................................... 73

4.8 Energy consumption of different schemes for 42 workloads normalized with respect to ICOUNT for an SMT processor with a shared-ROB structure ............................................................ 74

4.9 The percentage of power-off banks of RASR and ARPA-Min_E across 42 workloads ....................................................................... 75

4.10 Energy-to-Single_WIPC-Ratio (E-WIPC-R) of different schemes for 42 workloads normalized with respect to ICOUNT ................. 75
4.11 Resource Partitioning and Adaptation: (a) ARPA (b) RASR
(fma3d-mesa) .................................................................77

4.12 Energy Savings of RASR over ARPA as the number of bank size changes for 12 out of the 42 workloads. ...............................78

4.13 Energy-to-Single_WIPC-Ratio (E-WIPC-R) improvement of RASR over ARPA for different values of Threshold1 and Threshold2 for 12 out of the 42 workloads. ...............................79

4.14 Single_WIPC of different schemes for 42 workloads normalized with respect to ICOUNT for an SMT processor with a divided-ROB structure .................................................................80

4.15 Energy consumption of different schemes for 42 workloads normalized with respect to ICOUNT for an SMT processor with a divided-ROB structure .................................................................82

4.16 Energy-to-Single_WIPC-Ratio (E-WIPC-R) of different schemes for 42 workloads normalized with respect to ICOUNT ..........83
CHAPTER 1

INTRODUCTION

As processor designs become more and more complicated, how to manage the abundant processor resources to achieve the optimal performance and power consumption of processors becomes increasingly more sophisticated. This thesis investigates resource usage controlling techniques for general-purpose microprocessors targeting both performance and energy.

1.1 The Needs of Resource Management

By allowing an increasing number of transistors, advances in integrated circuit technology makes complicated microprocessor designs possible. Using a range of microarchitectural techniques [36], such as out-of-order processing of instructions, register renaming, and deep pipelines supported by sophisticated branch predictors, Instruction-Level Parallelism (ILP) has been well exploited. Simultaneous Multi-Threading (SMT) [53, 78] and Chip Multi-Processing (CMP) [35, 76] techniques improve the processor performance further by exploiting Thread-Level Parallelism (TLP) of applications. Moreover, the use of cache memories has significantly reduced the large latency of main memory accesses. The combination of architectural and cache organizational enhancements has boosted performance dramatically in the last few decades. However, the increased complexity of processor designs makes power consumption and heat dissipation significant challenges in the design of modern high-performance processors. The ever increasing power density of microprocessors intro-
duced many severe problems such as excessive heat and has a negative impact on reliability.

On the other hand, although general-purpose processors are designed to satisfy very complicated instruction executions and provide good performance across a spectrum of applications, applications needs for hardware resources such as caches, instruction queues, and renaming registers within a microprocessor, can vary significantly from one application to the other and even within the different phases of a given application. For example, applications or application phases with very high ILP or TLP prefer an architecture with high pipeline width and more parallel resources to simultaneously execute many instructions in the same cycle. In contrast, for applications or application phases with a large number of data dependencies or outstanding cache access misses, a complicated architecture design with high pipeline width and parallel resources only improves the execution speed by very little (as Amdahl’s law suggests, the speedup of parallel applications is limited by the sequential portion of the execution) but wastes considerable energy in powering many processor resources. Therefore, performance and/or power consumption are often non-optimal for a specific program or workload - or for a specific phase of a program. How to manage the given hardware resources to achieve optimal performance and power consumption becomes significantly sophisticated as processor designs enter into an era where both performance and power consumptions must be considered [7, 30, 38, 49].

1.2 Two Categories of Techniques
One category of techniques to handle the mismatching between the number of hardware resources and the real application needs for power-performance efficiency is called architecture adaptation [88]. These techniques power off processor components or throttle usage of resources when they do not contribute to performance. Another approach is to allow multiple threads to simultaneously share processor resources of a
single-core. The hardware resource usage rate is increased significantly by exploiting TLP and resource sharing, thus improving the overall instruction throughput. Energy is saved indirectly by having fewer idle resources [50, 51].

1.2.1 Architecture Adaptation

Architecture adaptation [88] adaptively activates and deactivates hardware resources in accord with the changes of an application’s behavior [4, 16, 17, 18, 24, 71].

As Zhu et al. [88] mentioned, the two key factors in architecture adaptation are when to trigger the adaptation and what adaptation techniques to apply. The approaches to find out when the adaptation should be triggered are diverse [18, 39, 42, 58, 71, 82, 83], while multi-configuration hardware resources [24] and pipeline throttling [58] are the two mainly used adaptation techniques.

Pipeline throttling slows down the rate of instructions passing through the pipeline or throttles instruction fetching for a specific number of cycles when the current ILP is very low or branches with low confidence level are executed. Throttling at these times can not only save energy by reducing the number of mis-fetched instructions entering into the pipeline but can also reduce the energy consumption of the ICache and issue queue logic with no performance loss.

Typically, architecture adaptation techniques with pipeline throttling can be classified into hardware-based runtime throttling techniques [8, 9, 16, 42, 58] and software-based static throttling techniques [39, 61, 82, 83]. Hardware-based runtime techniques assume the program behavior to be stable for a short time and they use past behavior to predict future behavior. The energy saving is relatively good. However, because they cannot catch future irregular situations such as abrupt program phase changes, they may cause substantial performance degradation. Software-based static techniques are based on compile-time program analysis or profiling and can estimate future phase changes, but they cannot capture dynamic effects, such as cache misses.
The resulting energy savings are relatively small but the performance loss is lower than that for hardware-based techniques. In Chapter 2, we propose a compiler-based architecture adaptation with fetch throttling technique to reduce the chip-wide energy.

1.2.2 Resource Sharing among Multiple Threads

The architecture adaptation mentioned in Section 1.2.1 is a passive way to save power consumption. The inefficient use of a very large transistor budget and the issue of increasing power consumption have caused computer architects to seek new paradigms for processor designs [31, 41, 69, 76].

SMT is a relatively new microarchitectural paradigm developed to improve the usage rate of current large transistor budgets. It has already found commercial applications [40, 44, 60]. SMT boosts the energy efficiency of modern microprocessor by allowing multiple threads to simultaneously share processor resources. This can increase the resource usage rate and improve the overall instruction throughput of processors, thus decreasing the energy consumption per instruction.
Figure 1.1 compares the SMT architecture with superscalar and fine-grained multithreading architecture. Each row of Figure 1.1 represents a single cycle of execution; the four boxes show four potential instructions that could be issued each cycle. A filled box means the issue slot is occupied by a thread. An empty box shows an unused instruction slot. Superscalar architecture, such as PowerPC 604 or UltraSPARC 1 [44] supports the execution of a single thread. Compared with other two architectures, superscalar has many empty slots. Simple Multithreading architecture like Tera [5] allows multiple threads run on different time cycles. Context switching between these threads can be done in a single cycle. However, because of low ILP there are still many empty horizontal slots compared to SMT [37, 78, 87]. SMT architecture allows instructions from multiple threads to be simultaneously fetched and executed in the same pipeline. Through resource sharing of multiple threads, SMT takes advantage not only of the existing ILP within each thread but also of TLP among threads. Instruction throughput is increased by effectively countering the impact of both long memory latencies and limited available parallelism within a single thread. Li et al. [51] show that SMT can substantially benefit energy efficiency metrics such as ED$^2$. CMP [35, 46] instantiates multiple processor cores on a single die. Usually, the cores are small and simple. They share chip resources such as memory controller, off-chip bandwidth, and L2 cache. CMP enables multiple threads running simultaneously on these different cores to improve the utilization of the transistors, thus increasing the overall throughput of processors. Each core in a CMP can also be extended to support multiple hardware contexts, achieving resources sharing in two levels.

Resource sharing among multiple threads can significantly improve resource utilization, thus improving the overall instruction throughout and decreasing energy consumption per instruction. However, applications running on SMT or CMP processors also compete among themselves for resources. If a single thread monopolizes
most of the resources, it will run almost at its full speed, but the other threads will suffer a slow down due to resource starvation. This will not only deteriorate the overall throughput, but also affect fairness which is a desirable objective for SMT or CMP processors. Equally partitioning resources [68] to threads can resolve the clogging problem, but the resource sharing benefits of SMT or CMP processors will be obliterated. Although the contribution of each thread to the overall processor performance is dependent on the amount of resources available to it, unfortunately, the relationship between a thread’s performance and the amount of resources allocated to it is rarely linear. Grasping the thread behavior and dynamically controlling the resource usage of each thread will be a good way of keeping the resource sharing benefits of SMT or CMP processors and resolving the resource clogging problem.

In Chapter 3, we introduce an Adaptive Resource Partitioning Algorithm (ARPA) in SMT processors to help threads efficiently share processor resources, thus improving performance.

Compared to the single-thread case, resource requirements of multi-threaded workloads running on an SMT processor have a higher variability. In Chapter 4, we analyze the variability of resource requirements of multi-threaded workloads and incorporate architecture adaptation into ARPA to adaptively control the number of powered-on resources and partition resources among threads targeting both performance and energy in SMT processors.

1.3 Related work

We present two categories of related work in this section. The first category is architecture adaptation techniques and the second category is techniques for controlling resource sharing of SMT processors.

Prior fetch throttling related work can be divided into two groups: software-based techniques [39, 61, 82, 83] and hardware-based techniques [8, 9, 16, 42, 58].
Unsal et al. [82] propose a compiler-driven static IPC-estimation-based fetch throttling scheme that is based on dependence testing in the compiler back-end. This scheme throttles fetching at a low estimated IPC value. They fix the throttling IPC-threshold to 2 to lower the effect on performance, but with a low threshold the energy savings are not significant.

Mehta et al. [61] present the fetch halting technique that suspends instruction fetching when the processor is stalled by a critical long latency instruction. In order to characterize critical instructions, they use software-profiling techniques to annotate the critical load instruction. Then, if the hardware predicts L2 cache or main memory misses when executing a critical load instruction, fetch will be halted.

An early hardware-based front-end technique is the pipeline gating work of Manne et al. [58]. The authors inhibit speculative execution when such execution is very likely to fail. They analyze the likelihood of a branch to mispredict and exclude wrong-path instructions from being fetched into the pipeline. Their results show a 38% reduction in wrong-path executions with a 1% performance loss.

Aragon et al. [8] also focus on reducing the power dissipated by mis-specified instructions. They propose selective throttling as an effective way of triggering different power-aware techniques (fetch throttling, decode throttling or disabling the selection logic). For branches with a low confidence prediction, the most aggressive throttling mechanism is used whereas high confidence branch predictions trigger the least aggressive techniques.

An alternative front-end approach is the fetch/decode throttling proposed by Baniasadi et al. [9]. This fine-grained approach utilizes the information passing through each pipeline stage to estimate the ILP. Based on this information, the fetch/decode stage is stalled when insufficient parallelism exists. However, as mentioned by the authors, traffic per pipeline stage is used as an indirect, and approximate, metric of power dissipation.
Karkhanis et al. [42] suggested throttling based on Just In Time Instruction Delivery. This scheme monitors and dynamically adjusts the maximum number of in-flight instructions in the processor. A counter for the number of in-flight instructions is incremented when an instruction is fetched, and decremented when an instruction is committed. Another register, MAXcount, sets the limit on the allowable in-flight instructions. Whenever the in-flight instruction count exceeds MAXcount, instruction fetching is stopped. The algorithm searches for the “optimal” number of in-flight instructions and changes the value of MAXcount at intervals of 100K committed instructions.

Buyuktosunoglu et al. [16] introduced an issue-centric fetch-gating scheme based on issue queue utilization and application parallelism characteristics. The issue queue utilization is obtained by tracking the occupancy of the issue queue and the application parallelism characteristics are obtained by monitoring from how deep in the Reorder Buffer (ROB) are instructions being issued. Fetching is stopped if over half of the instructions that were issued are located in the lower half of the ROB and the issue queue is at least half full.

Multi-configurations of architecture adaptation techniques usually focus on power-hungry hardware components like IQ, LSQ, ROB, register file and caches. Resources are activated and deactivated according to program behavior changes. Energy savings is achieved at the cost of a small performance degradation.

In [18], the authors examine the power saving potential in the design of an adaptive IQ structure. They propose an implementation that divides the IQ into separate chunks allocated dynamically according to the prevailing level of instruction level parallelism. Power savings are achieved by turning off unused chunks. Folegnani et al. [29] use program parallelism statistics to drive the adaptation of IQ size. If the processor rarely issues instructions from the back of the issue queue, the system assumes the queue to be larger than necessary and downsizes it.
Abella et al. [2] propose a power-efficient adaptive resizing scheme for the IQ and register file. The proposed mechanism is based on monitoring how much time instructions spend in both the IQ and ROB and limits their occupancy based on these statistics. Ponomarev et al. [65] present a mechanism to dynamically, simultaneously and independently resize the IQ, ROB and LSQ. Downsizing is driven by directly using sampled estimates of their individual occupancies. Upsizing is more aggressive using the relative rate of blocked dispatches to limit the performance penalty. In [67], Powell et al. propose a resizable I-cache that dynamically resizes and adapts to an application’s required size based on the average miss rate over a time interval.

Prior related work on controlling sharing of resources in SMT processor can be categorized into three groups: fully flexible resource distribution [19, 26, 56, 80], static resource allocation [60, 68] and dynamic resource partitioning [20, 22].

Tullsen et al. [78, 80] exploit several fetch policies which determine how threads are selectively fetched to share a common pool of resources. RR is their simplest policy; it fetches instructions from all threads in round robin order, disregarding the resource usage of each thread. ICOUNT dynamically biases toward threads that will use processor resources most efficiently, thereby improving processor throughput. It performs better than RR and is easy to implement. However, ICOUNT cannot prevent some threads with high L2 miss rate from being allocated an excessive share of pipeline resources.

STALL and FLUSH [79] are two techniques built on top of ICOUNT to ameliorate this problem. STALL prevents a thread with a pending L2 miss from entering the pipeline. FLUSH, an extension of STALL, flushes all instructions from such a thread; this obviously has an energy overhead. FLUSH++ [19] combines FLUSH and STALL.

Data Gating (DG) [26] stalls threads when the number of L1 data misses exceeds a given threshold. Predictive Data Gating (PDG) [26] prevents a thread from fetch-
ing instructions as soon as a cache miss is predicted. Both techniques build upon ICOUNT to prevent resource hogging.

Static resource partitioning [60, 68] evenly splits critical resources among all threads, thus preventing resource monopolization by a single thread. However, this method lacks flexibility and can cause resources to remain idle when one thread has no need for them, while the other threads could benefit from additional resources.

DCRA [20] is a partly dynamic resource sharing algorithm. Each thread is assigned a resource usage bound and these bounds are changed dynamically. The bound is higher for threads with more L1 Data cache misses. However, DCRA does not work well on applications like mcf with high data cache miss rates and extremely low baseline performance. Allocating more resources to such threads improves their performance by very little and comes at the expense of decreased performance of other resources-starved threads.

Hill climbing [22] uses performance feedback to direct the partitioning. This learning-based algorithm starts from equal partitioning, then tries to move an equal amount of resources from all the other threads to a “trial” thread.

Like DCRA [20] and hill climbing [22], our algorithm, ARPA also partitions resources dynamically. However, ARPA’s analysis of program behavior results in a more effective use of resources.

Sharkey et al. assume a logically separated ROB structure and present a mechanism to adaptively determine the number of usable ROBs to each thread [72]. The purpose of ROB downsizing in [72] is to prevent threads from clogging shared data-path resources, thus improving performance.

1.4 Thesis Contributions

This dissertation makes the following primary contributions:
• Conventional superscalar processors attempt to maximize the number of “in-flight” instructions at all times in order to fully exploit ILP. This results in a significant waste of energy when ILP is low. We present a Compiler-based Adaptive Fetch Throttling (CAFT) technique in superscalar processors to save energy. CAFT combines compile-time estimated IPC information with hardware runtime information to dynamically change the throttling threshold according to the changes in program behavior. The energy-delay product (EDP) savings of CAFT is doubled compared to a fixed low-IPC-threshold scheme.

• We present a utility-based dynamic resource partitioning scheme called ARPA to control processor resource sharing. The algorithm identifies the resource usage efficiency of each thread and allocates more resources to threads which can use them in a more efficient way. The efficient usage of processor resources greatly improves the overall instruction throughput.

Our simulation results on a set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy ICOUNT by 55.8% considering overall instruction throughput and achieves a 33.8% improvement over Static Partitioning. It also outperforms the current best dynamic resource allocation technique, Hill-climbing by 5.7%. ARPA is an adaptive process that allows threads to share resources more fairly and efficiently. Considering fairness accorded to each thread, ARPA attains 43.6%, 18.5% and 9.2% improvements over ICOUNT, Static Partitioning and Hill-climbing, respectively.

• Compared to the single-thread case, resource requirements of multi-threaded workloads running on an SMT processor have a higher variability. We propose resource adaptation approaches to adaptively control the number of powered-on ROB entries and partition shared resources among threads for both shared-ROB and divided-ROB structures, targeting both high performance and low energy.
We consider not only the relative resource usage efficiency of each thread, but also take into account the real resource usage of threads to identify cases of inefficient resource usage. Energy savings are achieved by powering off resources that do not improve the performance and only waste power. Our experimental results show that for an SMT processor with a shared-ROB structure, our resource adaptation approach achieves 16.7% energy savings over ARPA, while the performance loss is negligible across 42 sample workloads. For an SMT processor with a divided-ROB structure, our resource adaptation approach outperforms ARPA by 4.2% in addition to achieving 12.4% energy savings.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows.

In chapter 2, our Compiler-based Adaptive Fetch Throttling (CAFT) technique is presented. We briefly describe the compiler-based static IPC estimation approach. Then, we present our compiler-based adaptive fetch throttling technique. For comparison purposes, we also describe the dependence-based hardware-only throttling technique, the just-in-time instruction delivery technique and the DID-based technique. Finally we present our numerical results.

Chapter 3 presents a utility-based Adaptive Resource Partitioning Algorithm (ARPA) in SMT processors. We analyse the thread behavior and dynamically allocate resources to threads according to thread behavior changes. We also present the detailed algorithm and describe its implementation. Then we show our evaluation methodology followed by numerical results.

In chapter 4, we propose resource adaptation approaches targeting both high performance and low energy for SMT processors. We first provide an overview of the resource utility analysis which our resource adaptation is based on. Then, we present the detailed resource adaptation algorithms for SMT processors with a shared-ROB
and a divided-ROB structure. We also compare the performance and energy results of our resource adaptation scheme with other resource partitioning schemes.

Finally, chapter 5 includes a summary of the primary results in this dissertation and a number of directions for future work in this area.
CHAPTER 2
FETCH THROTTLING IN SINGLE-THREAD PROCESSORS

Front-end instruction delivery accounts for a significant fraction of energy consumption in dynamically scheduled superscalar processors. Different front-end throttling techniques have been introduced to reduce the chip-wide energy consumption caused by redundant fetching. Hardware-based techniques, such as flow-based throttling, could reduce the energy consumption considerably, but with a high performance loss. On the other hand, compiler-based IPC-estimation-driven software fetch throttling (CFT) techniques result in relatively low performance degradation, which is desirable for high-performance processors. However, their energy savings are limited by the fact that they typically use a predefined fixed low IPC-threshold to control throttling.

In this chapter, we propose a Compiler-based Adaptive Fetch Throttling (CAFT) technique that allows changing the throttling threshold dynamically at runtime. Instead of using a fixed threshold, our technique uses the Decode/Issue Difference (DID) to assist the fetch throttling decision based on the statically estimated IPC. Changing the threshold dynamically makes it possible to throttle at a higher estimated IPC, thus increasing the throttling opportunities and resulting in larger energy savings. We demonstrate that CAFT could increase the energy savings significantly compared to CFT, while preserving its benefit of low performance loss. Our simulation results show that the proposed technique doubles the EDP savings compared to the fixed threshold throttling and achieves a 6.7% overall EDP saving.
2.1 Introduction

Power consumption has emerged as a significant factor in computer architecture. Out-of-order processing of instructions, speculative execution, and register renaming techniques improve performance significantly compared to in-order execution. However, they also introduce significant energy overhead to keep track of all the instructions and their dependencies.

A large fraction of power in modern high-performance processors is dissipated by the front-end of the pipeline, including the fetch and decode units. Conventional superscalar processors attempt to maximize the number of “in-flight” instructions at all times in order to achieve high performance. Following a branch misprediction, they begin fetching at full speed and continue until the next branch misprediction flushes the pipeline or until the issue queue (or re-order buffer) is full. No matter how low the ILP may be, instructions are still fetched, decoded and then put into the issue queue. This not only increases the energy consumption of the issue queue logic with additional wake-ups and selections, but also adds more pressure to the register file.

A number of front-end throttling techniques have been proposed for improving the energy efficiency during the fetch process in superscalar pipelines. These techniques can be categorized into hardware-based runtime [8, 9, 16, 42, 58] and software-based static [39, 61, 82, 83] techniques.

Hardware-based techniques assume that the program state is stable and use the recent history information to predict future behavior. These can catch dynamic behavior such as cache misses but cannot catch irregular situations such as abrupt phase changes. For example, a flow-based fetch throttling technique [9] uses the instruction Decode/Commit rate (DCR) in the previous cycle to decide whether to stall instruction fetch in the next cycle. Throttling is triggered when high DCR values occur as a result of branch misprediction. Because this approach cannot catch the bursty
behavior of programs, it may cause substantial performance degradation (more than 8% for some benchmarks).

Software-based throttling techniques can estimate the ILP based on compile-time program analysis and provide indications of sharp changes in ILP (or ILP bursts). Static techniques may, however, produce inaccurate predictions due to their inability to capture dynamic effects such as branch mispredictions and cache misses. Previous research [82] employed compiler techniques to estimate the IPC and used the estimated IPC to drive its fine-grained fetch-throttling energy-saving heuristic. A fetch will be stalled in the following cycle if the estimated IPC is lower than a predefined threshold, which in [82] has been set to 2 for an 8-way issue processor. Throttling using such a low threshold will have only a small negative effect on performance, but will also yield relatively small energy savings.

There are two potential problems using a fixed low value of the IPC-threshold to drive fetch throttling. The first one is that it limits the throttling opportunities at high IPC values. If there are many instructions left unexecuted in the previous cycle, we could throttle at a higher IPC-threshold with probably no performance loss. The second problem is that the fixed IPC-threshold technique may throttle at an inappropriate time, resulting in a performance loss. Assume, for example, that the estimated IPC in the following cycle is 2, but there are no instructions left in the issue queue from the previous cycle; a throttling at this time is inappropriate and will result in a performance loss. Therefore, by using adaptive rather than fixed IPC-thresholds for fetch throttling, we could overcome both problems and obtain better results.

We present a new approach called Compiler-based Adaptive Fetch Throttling (CAFT), which allows changing the throttling IPC-threshold adaptively at runtime. Our technique is based on compile-time static IPC estimation, but we use the Decode/Issue Difference (DID) to assist the fetch throttling decision that is based on
the statically estimated IPC. DID is the difference between the numbers of decoded and issued instructions in the previous cycle, which can be considered as the recent history information. The IPC-threshold is changed dynamically according to the DID value, making it possible to throttle at a higher estimated IPC if appropriate. This increases the throttling opportunities and thereby results in larger energy savings.

2.2 CAFT: Compiler-based Adaptive Fetch Throttling

We start with a brief introduction to the compiler-level IPC-estimation scheme.

2.2.1 Compiler-based IPC Estimation

Our implementation considers only true data dependencies (Read-After-Write or RAW) to check if instructions depend on each other or can be executed in parallel. As mentioned in [59], a major limitation of increasing ILP is the presence of true data dependencies. Tune et al.[81] also remark that the bottleneck for many workloads on current processors is true dependencies in the code. Although the impact of true dependencies can be mitigated through the use of value speculation, the energy overhead of value speculation hardware has been found to be prohibitively high [10]. In our experiments, we consider a standard, non-value speculating out-of-order architecture. However, the compiler-driven framework is equally applicable to an architecture with value speculation, only the compiler-level passes need to be modified.

We statically determine true data dependencies using data dependency analysis at the assembly-code level. Our post-register allocation scheme uses monotone data flow analysis, similar to [6]. We identify data dependencies at both registers and memory accesses. Register analysis is straightforward: the read and written registers in an instruction can be identified easily, since registers do not have aliases. The determination of reaching uses is achieved using the algorithm in [3].
However, for memory accesses, there are three implementation choices: no alias analysis, complete alias analysis, or alias analysis by instruction inspection [62]. We perform an approximate and speculative alias analysis by instruction inspection that provides ease of implementation and sufficient accuracy. In this scheme, we distinguish between different classes of memory accesses such as static or global memory, stack and heap. We also consider indexed accesses by analyzing the base register and offset values to determine if different memory accesses are referenced. If this is the case, we do not consider this pair of read-after-write memory accesses as a true dependency.

We use SUIF [86]/MachSUIF [74] as our compiler framework. SUIF does high-level passes while MachSUIF performs machine-specific optimizations. The final MachSUIF pass produces Alpha assembly code. We have added new passes to both SUIF and MachSUIF to annotate and propagate the static IPC-estimation. Our IPC-estimation is at the basic block or loop level: loop beginnings and endings serve as natural boundaries for the estimation. The high-level loop annotation pass works with expression trees and traverses the structured control flow graph (CFG) of each routine. The other added pass, the IPC-prediction pass, is a lower-level MachSUIF pass that runs just prior to assembler code generation.

2.2.2 Adaptive Fetch Throttling

As mentioned before, the previous compile-time static IPC-estimation based fetch throttling framework fixes the throttling IPC-threshold at 2 and assumes that throttling at such a low IPC will have little impact on performance. This approach limits the energy savings because it ignores many throttling opportunities which exist at a higher IPC. However, if we fix the throttling IPC-threshold at a higher value, the performance will rapidly decrease due to too frequent throttling.
An adaptive throttling IPC-threshold, which would allow us to throttle at a higher estimated IPC, could be beneficial if it can still keep the performance loss low. If we can change the IPC-threshold adaptively and throttle at a higher IPC only when appropriate, we can reduce the number of cache accesses considerably. As a result, instructions are fetched just in time to exploit the available parallelism. Also, since fetching will proceed at a slower pace, we will reduce the number of incorrectly speculated instructions that enter the pipeline.

After instructions are decoded, they are put into the issue queue and executed on individual functional units if all the operands are ready and enough functional units are available. In a perfect machine with no constraints such as true data dependencies, the number of instructions through the decode and issue stages should be identical during a period of time. When the number of decoded instructions surpasses the number of issued instructions, this means that sufficient parallelism does not exist and instruction decode has to be suspended. If we continue to fetch and decode instructions at this time we only add to the overall energy consumption, especially in the I-cache, while not improving performance. A large Decode/Issue Difference (DID) value means that many instructions were left unexecuted, and the performance will not be affected if we throttle for one cycle.

The difference between decoded and issued instructions can be considered as the recent history information that is used to change the throttling IPC-threshold dynamically. With both recent history information (runtime DID value) and future estimation (compiler-time IPC estimation), we can capture the properties of the pipeline behavior more accurately than by using the hardware-based dynamic throttling or the compiler-based static throttling individually.
2.2.2.1 The Algorithm

Instead of using a fixed IPC-threshold such as 2, CAFT dynamically changes the IPC-threshold between 2 to 5. Even when the estimated IPC is 2, we still selectively throttle based on the runtime DID value, instead of throttling whenever the IPC is below 2 as in [82]. The reason is that although the estimated IPC may be low in some cycles, we should not throttle if execution units are idle and waiting for incoming decoded instructions. If we do, throttling will impact the performance. On the other hand, we limit our highest IPC-threshold to 5. Our experiments show that throttling above an estimated IPC of 5 is very rare.

As mentioned above, we use the instruction Decode/Issue Difference (DID) to assist the IPC-estimation throttling technique to throttle at changeable thresholds. If the instruction decoding rate matches the instruction issuing rate (i.e., the DID value is zero), no fetch throttling is needed. If the DID value in the last cycle is greater than zero, which means that redundant instructions were decoded, there exist opportunities to throttle the fetch in the next cycle. Additional fetching will introduce the possibility of miss-fetching and increase the number of Icache accesses, resulting in a waste of energy. For example, if the DID in the previous cycle is 3 and the IPC estimate in the next cycle is less than 3, we can safely throttle for one cycle during instruction fetching. If the instructions left unused in the previous cycle can provide the needs of the next cycle, stopping fetching for one cycle will not hurt the performance. The algorithm can be summarized as follows:

\[
\text{IF Estimated\_IPC} \leq \text{DID} \\
\text{THEN throttle for one cycle}
\]

For different DID values, we throttle for all the estimated IPCs up to the DID value. The DID value captures dynamic effects such as cache misses and branch
mispredictions, which are not captured by the fixed IPC-threshold compiler-based fetch throttling methods.

2.2.2.2 Architecture-level Implementation

The structure of our architecture-level design is similar to [82], which uses a fixed IPC-threshold. Estimated IPC values are inserted into the binary code and forwarded to the pipeline during decoding. It requires 2-3 bits to encode the estimated IPC values. If enough flexibility exists in the ISA of the target processor, this information can be encoded directly into the instructions, eliminating the need for a special instruction. We have also conducted an analysis for worst-case code size increase, in case the ISA cannot accommodate the extension. The results indicate an average worst-case code size increase of 5.2%.

We show the architectural implementation of CAFT in Figure 2.1. The compiler-supplied estimated IPC value is identified and latched at the decode stage. We also add two counters to monitor the number of instructions decoded and issued in the previous cycle. The values of these counters are subtracted to calculate the DID, which is then compared with the estimated-IPC latched at the decode stage. If the

Figure 2.1. Hardware implementation for CAFT
estimated-IPC is smaller than the DID, a fetch throttling signal is generated and transmitted to a clock-gater to stall fetching for one cycle [82].

2.2.3 Hardware-based Fetch Throttling

Dependence-Based (DEP) For purposes of comparison, we also implemented a hardware Dependence-based (DEP) scheme [9]. DEP inspects the instructions currently being decoded and counts the dependencies among them. Whenever the number of dependencies exceeds a pre-specified threshold, a throttling signal is triggered. The justification for this scheme is that a large number of dependencies is an indication of a long and probably critical computation path. Consequently, it is unlikely that prefetching additional instructions will significantly improve performance.

Instead of throttling for both fetch and decode stages as in [9], we only throttle the fetch stage, as in CAFT. Also, we compare the number of dependencies among the decoded instructions to the number of decoded instructions every cycle instead of to the decode-width as done in [9]. Comparing to the decode-width does not consider the fact that the number of decoded instructions in one cycle affects the number of dependencies in that cycle. Our experiments also prove that this approach will have higher energy savings than [9]. We obtain the best results when, once the number of dependent instructions among the decoded instructions is greater than half the number of decoded instructions, a throttle is triggered at the following cycle. We show the comparison of this approach with our CAFT approach in Section 2.4.

Just-In-Time Instruction Delivery (JIT) The Just-In-Time (JIT) instruction delivery scheme [42] is another hardware-based fetch throttling technique similar to our technique. It uses information about in-flight instructions to control the front-end instruction fetching. When the number of in-flight instructions exceeds the MAXcount, instruction fetching is inhibited. The MAXcount value can be dy-
namically adjusted to the least value such that performance is not reduced by some threshold amount, e.g., 2%.

Because our processor configuration is different from that in [42], the tuning parameters in adjusting the MAXcount value are also different. After extensive experiments, we conclude that when the initial value of MAXcount is 32 and the MAXcount increment is 16 in every 100K instructions interval during the tuning process, the energy savings are maximal. With these parameters, our measured performance reduction is similar to that in [42] (i.e., 3%). Another difference is that when MAXcount is an “optimal” value, we only restart tuning if the performance (IPC) changes by more than some “noise” margin. We do not consider the changes of the number of branches as a reason for retuning, because in some benchmarks, considering branches as “noise” will cause many unnecessary tunings, resulting in a substantial performance reduction.

**Decode/Issue Difference (DID)** In order to verify that the DID-directed CAFT is more efficient than any of the individual schemes alone, we also test the hardware-only Decode/Issue Difference (DID) technique. This technique assumes that insufficient parallelism exists when the number of instructions decoded exceeds the number of instructions issued and continuing fetching will make the instructions stay longer in the issue queue wasting the wake-up and selection energy of the issue logic. In such a case, we can throttle the fetch.

We tested different ratios between the numbers of decoded and issued instructions and obtained the best results when the number of decoded instructions is twice the number of issued instructions. If we set the ratio to a lower value, the performance will decrease rapidly. This is different from our CAFT scheme, which can throttle at a very low Decode/Issued Difference value if the IPC estimate in the next cycle is low. We will show the results in Section 2.4.
### Table 2.1. Baseline parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Speed</td>
<td>1.5GHz</td>
</tr>
<tr>
<td>Process Parameters</td>
<td>0.18 $\mu$m, 2V</td>
</tr>
<tr>
<td>Issue</td>
<td>Out-Of-Order</td>
</tr>
<tr>
<td>IF,ID,IS,IC Width</td>
<td>8-way</td>
</tr>
<tr>
<td>Fetch Queue Size</td>
<td>32</td>
</tr>
<tr>
<td>Load/Store Queue Size</td>
<td>64</td>
</tr>
<tr>
<td>Instruction Queue Size</td>
<td>128</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>2K entry bimodal</td>
</tr>
<tr>
<td>Int. Functional Units</td>
<td>4 ALUs, 1 Mult./Div.</td>
</tr>
<tr>
<td>FP Functional Units</td>
<td>4 ALUs, 1 Mult./Div.</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>128K, 4-way, writeback</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>128K, 4-way, writeback</td>
</tr>
<tr>
<td>Combined L2 cache</td>
<td>1M, 4-way associative</td>
</tr>
<tr>
<td>L2 Cache hit time</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Main memory hit time</td>
<td>100 cycles</td>
</tr>
</tbody>
</table>

#### 2.3 Experimental Setup

The baseline architecture is described in Table 2.1. Our baseline processor configuration has 128 entries in its instruction queue; therefore we use a 128 element Register Update Unit (RUU). The RUU includes the instruction queue as well as the physical register files and the reorder buffer. We use a size of 64 for the Load-Store Queue.

We used the SimpleScalar [13]/Wattch [12] framework to run the binaries and collect the energy results. We ran our baseline application without any annotations and without any throttling, and all other throttling versions were compared against this baseline. SimpleScalar has been modified to recognize the compiler-generated IPC flags and IPC values. In Wattch, we used the activity-sensitive power model with aggressive conditional clocking. The rationale for this choice was to compare our fetch-throttling framework to an unthrottled baseline that is already power-efficient. Wattch can be retuned for the state-of-the-art technology scaling parameters; we use a 0.18 $\mu$m, 1.5GHz, 2V process. We extended the power dissipation model in Wattch.
so that it accounts for the extra power overhead due to the 2-bit field decoding in the dispatch stage and the comparison hardware logic.

We selected a mix of computation-bound and multimedia applications from the SPEC2000 [1] and Mediabench benchmark suites. We randomly chose four applications from each suite: pegwit, gsm, jpeg, mesa from Mediabench; gap, parser, vpr, equake from SPEC2000. We ran all Mediabench applications to completion. For the SPEC CPU2000 benchmarks we skipped past the initialization stage and simulated the next 500 million instructions using the reference input set. To skip the initialization phase, we fast-forwarded by the number of instructions as prescribed by Sair et al. [70] in their SPEC CPU2000 initialization segment analysis.

2.4 Results and Analysis

In this section, we first show the throttling cycles and the IPC distribution when throttling using CAFT in the different benchmarks. Then, we present the execution time and energy results for different throttling schemes in different benchmarks. When explaining how fetch throttling can save energy, we also show the reduction in miss-speculated instructions and the distribution of the fetch width for each fetch operation. Finally, we show the Energy-Delay Product (EDP) and ED$^2$P for the different throttling techniques.

2.4.1 IPC Distribution

In order to show how many throttle cycles are added after changing the threshold adaptively, we counted the number of throttle cycles in different benchmarks for both CFT and CAFT, normalizing the number of throttle cycles of CAFT to that of CFT. The results appear in Figure 2.2. From this figure we can see that CAFT has more throttling cycles than CFT, especially for pegwit, where the value is up by 3.8x. More
throttling cycles result in higher energy savings. On average, the total number of throttling cycles for CAFT is almost doubled compared to CFT.

The total number of throttling cycles increases substantially because CAFT can throttle fetching at higher estimated IPC values. In order to identify how many times CAFT throttles with an estimated IPC above 2, we analyzed the estimated IPC distribution when throttling (see Figure 2.3). From this figure, we can see that more than half of all the throttling cycles have an IPC which is above the threshold of 2, in all the benchmarks except mesa. In many of the throttles the IPC is 3 or 4 but very few have an IPC of 5, since throttling with such a high IPC value may result in a significant performance penalty.
2.4.2 Execution Time and Energy

Figure 2.4 shows the execution time and energy consumption of five different fetch throttling schemes, normalized to the baseline without fetch throttling (No-Throttle). The schemes are: hardware dependence-based (DEP) [9], just-in-time instruction delivery (JIT) [42], compiler-based fixed threshold (CFT) [82] and our compiler-based adaptive scheme (CAFT). In order to verify that the DID-directed CAFT is more efficient than any of the individual schemes alone, we also compared CAFT to the hardware only Decode/Issue Difference (DID) technique.

First, we observe that in most cases, the hardware-based fetch throttling schemes (DEP, JIT and DID) have a longer execution time than the static IPC-estimation based fetch throttling technique. On average, DEP increases execution time by 4%, JIT increases execution time by 3.3%, while CAFT increases it by only 1.5%. For the equake benchmark the execution time is increased by more than 8% when using DEP, which is undesirable for high performance processors.

Hardware-based schemes cause a large performance loss because such techniques can only capture the history information and use the past behavior to drive fetch.
throttling. They assume that the program behavior in the past and the near future is stable, yet many programs exhibit irregular or bursty behavior which cannot be detected solely based on the past behavior. Although JIT can dynamically adjust the future MAXcount as a function of past program behavior, the tuning process itself may cause significant performance degradation. Also, such changes can be detected only after a large interval (e.g., 10K instructions).

Static IPC, on the other hand, is a compile-time estimation of the actual IPC based on program analysis, and thus can provide an indication of a sharp change in ILP. CFT uses a fixed low IPC-estimation as the throttling threshold and has a small performance loss. For CAFT, although it can throttle at higher estimated IPCs and for more cycles, the performance loss is still low. The combination of DID and future IPC estimation can capture the program behavior more accurately than software- or hardware-only fetch throttling techniques. With the help of recent history information, dynamic effects like cache misses and branch misprediction will have a smaller effect on static IPC-estimation-based fetch throttling. Thus, when DEP cannot catch such bursty program phase changes, it causes a significant performance decrease. This is especially evident in Figure 2.7 which shows that the ED$^2$P for DEP is worse than for the No-Throttle baseline.

When comparing the normalized energy in Figure 2.4, CFT is the worst scheme, with only 4.6% total energy savings compared to the baseline. The reason is that
CFT loses many throttling opportunities when the estimated IPC is high, due to its low IPC-threshold of 2. CAFT, which changes the throttling threshold adaptively based on the application characteristics, can throttle at a high estimated IPC and thus throttles more cycles than CFT. The use of DID information ensures that CAFT does not increase the execution time compared to CFT. CAFT achieves greater energy reduction than CFT and the total average energy reduction is almost 8%. CAFT also has higher energy savings when compared to the hardware-based fetch throttling schemes DEP, JIT and DID.

From Figure 2.4, we also observe that DID, when applied alone, cannot match the energy/performance benefits of CAFT. Like DEP and JIT, DID can not catch bursty program phase changes. As a result, it has a larger performance decrease than the software-based schemes CFT and CAFT.

The increase in the number of throttling cycles is the main cause for the reduction in energy consumption in CAFT. A higher number of throttling cycles means a greater reduction in the number of fetched and executed miss-speculated instructions and a reduction in Icache accesses. With fewer instructions fetched, it not only saves Icache accesses energy, but also reduces the actions of forwarding instructions through pipeline stages, resulting in whole chip energy savings. As shown in Figure 2.5, the average reduction in miss-fetched instructions in CAFT is near 45% relative to the non-throttle scheme, while JIT achieves only a 36% reduction in miss-fetched

**Figure 2.6.** Fetch width distribution (averaged across all benchmarks)
instructions. Fetch throttling can greatly reduce the unnecessary miss-fetched flushes and save energy in different pipeline stages. Also, with a higher number of throttling cycles compared to CFT, CAFT can greatly reduce the number of Icache accesses and cause the number of instructions in each Icache access to be either 0 or 8 most of the time. The reason is that the number of available entries of the fetch buffer determines the number of instructions that can be fetched. Fetch throttling can cause the fetch buffer to be drained, allowing it to accommodate a full fetch (i.e., 8 instructions) fairly often. With the same number of instructions to run, the increase in the number of full fetches will decrease the number of total Icache accesses. From Figure 2.6, we can see that for CAFT and JIT the number of fetches is most of the time either 0 or 8 while in the non-throttle case the number of instructions fetched is distributed between 1 to 7 half of the time.

2.4.3 EDP/ED$^2$P

In order to highlight the impact on performance, we show the EDP and ED$^2$P savings in Figure 2.7. Although CAFT has no significant energy savings relative
to DEP and JIT, the EDP reduction is significant due to the smaller performance decrease. Similarly, although CAFT has the same low performance loss as CFT, the EDP reduction for CAFT is larger because it saves more energy. CAFT is thus more beneficial than software- or hardware-only fetch throttling techniques when EDP is used as the metric.

As shown in Figure 2.7, DEP achieves 3.5% EDP reduction and CFT achieves 3% reduction on average. The EDP savings of CAFT are more than doubled for several benchmarks compared to either DEP (e.g., mesa and jpeg) or CFT (e.g., pegwit and gsm), averaging 6.7% reduction on all benchmarks. When considering ED^2P, DEP has no benefit at all because of the high performance penalty. The other three schemes (JIT, DID and CFT) have only an average reduction of less than 2%, while the average ED^2P improvement for CAFT is 5.3%.

### 2.5 Chapter Summary

Throttling at a fixed low IPC limits the capability to reduce energy for static IPC-estimation-based fetch throttling techniques. In this chapter, we presented the Compiler-based Adaptive Fetch Throttling (CAFT) technique, which attempts to change the throttling IPC-threshold adaptively and still maintain a good performance. Compared to the previous fixed threshold approach (CFT) [82], CAFT achieves a 3.7% additional EDP saving and 6.7% overall EDP reduction. In comparison with previous hardware dependence-based fetch throttling schemes (DEP and JIT), CAFT has a lower performance degradation and a higher EDP reduction.
CHAPTER 3

DYNAMIC RESOURCE SHARING IN SIMULTANEOUS MULTITHREADING (SMT) PROCESSORS

A superscalar processor dispatches multiple instructions within a single application to redundant functional units in order to fully exploit ILP. However, the performance improvement is usually limited by both long memory latencies and limited available parallelism within a single application. SMT [14, 25, 37, 43, 53, 57, 77, 78, 87] allows instructions from multiple threads to execute at the same time, taking advantage not only of the existing ILP within each thread but also the TLP among them. Overall instruction throughput is significantly improved by effectively countering the impact of both long memory latencies and limited available parallelism within a single thread. In an SMT processor, all the resources can be shared among threads, except for some resources related to the architectural state which are separated to maintain the correct state of each logical processor.

However, multiple threads not only share the microprocessor resources, but also compete with each other on the resources. With inefficient sharing of microprocessor resources, threads that flow slowly through the pipeline may clog the resources, preventing other threads from entering into the pipeline. Different techniques [21, 28, 32, 54, 55, 73] have been proposed to further improve the performance and fairness of SMT processors. In this chapter, we propose a new Adaptive Resource Partitioning Algorithm (ARPA) to handle these sharing and competing issue in current SMT processors.
3.1 Introduction

Traditionally, a fetch policy [80] decides which threads enter the pipeline to share available resources in an SMT processor. Threads compete for resource access and there are no individual restrictions on the resource usage of each thread. Unfortunately, some threads may occupy a disproportionately large share of system resources, and slow down others. For example, threads with outstanding L2 data cache misses often run slowly as they must wait for these misses to be served, thereby clogging system resources [79].

Statically partitioning resources among threads has been suggested as a way to prevent a single thread from clogging resources [27, 52, 68]. However, such techniques are limited by the fact that different threads have differing requirements, and that these can vary with time. Recently, techniques have been proposed to dynamically partition resources [20, 22]. Such techniques can lead to significant improvements in performance.

Current resource partitioning approaches [20, 22, 68] mainly focus on some critical resources which significantly impact performance if clogged by some threads. Commonly, they apply the same partitioning principles to all the resources to be partitioned. [68] studies the effect of partitioning the instruction queue or the reorder buffer (ROB). DCRA [20] separately partitions queue and register entries using the same sharing model. Threads exceeding their specified bound are prevented from entering instructions into the pipeline. Hill-climbing [22] partitions integer rename registers among the threads, assuming that the integer issue queue (IQ) and ROB will be proportionately partitioned. It does not directly control the floating point IQ and the corresponding renaming registers.

In this chapter, we present a new Utility-based Adaptive Resource Partitioning Algorithm (ARPA) which concentrates on partitioning the following shared queue structures: instruction fetch queue (IFQ), IQ and ROB. We do not partition the
renaming registers since partitioning ROB can effectively control the sharing of registers. Doing so, however, would be quite easy. The physical implementation of ROB in an SMT processor can either be a single buffer with multiple head and tail pointers [20, 22, 68], one for each thread, or a separate buffer per thread [72, 80]. We assume a shared ROB structure for consistency with previous resource partitioning schemes in [20, 22, 68]. Moreover, the shared ROB implementation allows a more flexible ROB usage by the threads, resulting in greater performance benefits. However, our partitioning algorithm can be applied to the divided ROB structure as well. If the ROB is physically divided, we will constrain the ROB usage by each thread. The difference is that for a private ROB structure, the ROB usage of each thread cannot exceed its private ROB capacity, while for a shared ROB structure, each thread can use more than its equally partitioned share. A detailed experimental results of ARPA for a divided ROB structure will be presented in the next chapter. We do not constrain the usage of individual queues. Instead, we impose an upper bound on the sum of IFQ and ROB assigned to each thread. The total number of instructions, in any thread, occupying these queues should not exceed this bound. The IQ is partitioned proportionately. Since the thread’s usage of different hardware resources are dependent on each other, partitioning one type of resources will indirectly control the usage of the other resources.

The goal of ARPA is to prevent resource underutilization and make each resource unit be used efficiently, thus improving overall instruction throughput. ARPA analyzes the resource usage efficiency of each thread and assigns more resources to threads which can use them in a more efficient way. Our simulation results on a set of 42 multiprogramming workloads show that ARPA outperforms the traditional fetch policy, ICOUNT and the current best dynamic resource allocation technique, Hill-climbing.
3.2 ARPA: Adaptive Resource Partitioning Algorithm

Existing dynamic resource partitioning techniques consider only one clogging source (e.g., L1 data cache misses [20]) or are based on periodic “trials” to select the best partitioning [22]. Instead of using specific clogging hints [20] or exhaustive “trials” [22], ARPA allocates resources based on the resource usage efficiency of each thread. By assigning more resources to threads which can use them in a more efficient way, ARPA can not only improve the overall resource usage efficiency, but also ameliorate different kinds of clogging, thereby improving the overall instruction throughput. In the next sections, we show the framework of ARPA and describe how we analyze the resource usage efficiency and use the analysis to drive the partitioning.

3.2.1 Framework

Figure 3.1 shows a high-level flow chart of ARPA. We divide the whole program execution into fixed-sized epochs (measured in processor cycles) and start with equally partitioned resources among the threads. After each epoch, we analyze the current resource usage to identify whether the threads have used their allocated resources efficiently in this epoch. Our resource partitioning decision is driven by these analyses. The analysis and partitioning processes are repeated every epoch until the end of the program.

3.2.2 Resource Utilization Analysis

Our resource utilization analysis is carried out at the end of each epoch and is based on the following metric.
3.2.2.1 Metric of Usage Efficiency

We use Committed Instructions Per Resource Entry (CIPRE) to represent the usage efficiency of processor resources in each epoch. The CIPRE metric can express two different characteristics: (a) the usage efficiency of all processor resources and (b) the usage efficiency of only the resources that are allocated to a specific thread. Note that a thread with a higher CIPRE does not necessarily have a higher IPC. For example, threads A and B run simultaneously with 50 and 20 queue entries, respectively and suppose that thread A commits 2000 instructions and thread B commits 1000 instructions during an epoch of length 1000 cycles. Therefore, the IPC of thread A is 2 and that of thread B is 1. The CIPRE of thread A is \( \frac{2000}{50} = 40 \) while the CIPRE of thread B is \( \frac{1000}{20} = 50 \). Because the CIPRE of B is greater than that of A, we say that thread B is more productive in this epoch. Resources allocated to B contribute more per unit to system performance than resources allocated to A although the IPC of A is greater than that of B. Therefore, giving more resources to the higher-CIPRE thread does not necessarily mean allocating the high-IPC thread more resources.

3.2.2.2 Partitioning Process

ARPA follows an adaptive resource partitioning strategy. It adjusts the number of resources allocated to threads at each epoch based on their CIPRE metric. In every epoch a thread with a greater CIPRE value will take some resources from a thread with a lower CIPRE value until the CIPREs of the two threads are close to each other. Through adaptive resource tuning, all threads will use their allocated resources with approximately equal efficiency, thus improving the usage efficiency of all processor resources. However, there is a possibility that threads with lower CIPRE values will lose most of their resources but their CIPREs will still be lower than that of the
most efficient thread, resulting in resource starvation. ARPA avoids this situation by assigning each thread a minimum number of resources no matter what its CIPRE is.

In order to explain ARPA better, we use a two-thread example to illustrate the adaptive process. Figure 3.2 shows the change in the CIPRE value when a program completes epoch $n+1$. $X_1$ and $X_2$ are the number of resource entries allocated to threads 1 and 2, respectively, during epoch $n$. $Y_1$ and $Y_2$ are the numbers of committed instructions of threads 1 and 2 during epoch $n$, respectively. \( \Delta \) is the number of resource entries that a thread can transfer to another thread in any one epoch. The CIPREs of threads 1 and 2, and the CIPRE of all processor resources at the end of epoch $n$ are shown below in 3.1, 3.2 and 3.3, respectively.

\[
CIPRE_1 = \frac{Y_1}{X_1} \quad (3.1)
\]

\[
CIPRE_2 = \frac{Y_2}{X_2} \quad (3.2)
\]

\[
CIPRE_{overall}^{(n)} = \frac{Y_1 + Y_2}{X_1 + X_2} \quad (3.3)
\]
Assuming that \( CIPRE_1 > CIPRE_2 \) in this example, it is easy to show that

\[
CIPRE_2 < CIPRE^{(n)}_{\text{overall}} < CIPRE_1
\]  

(3.4)

Since thread 1 achieves a more efficient usage of the allocated resources in epoch \( n \), ARPA will transfer to it \( \Delta \) resources from thread 2 in the next epoch. That is to say, thread 1 will be assigned \( X_1 + \Delta \) resource entries and thread 2 will be restricted to \( X_2 - \Delta \) resource entries in epoch \( n + 1 \). If both threads still use their allocated resources with the same efficiency as in epoch \( n \), the CIPRE of the total resources in epoch \( n + 1 \) will be:

\[
CIPRE^{(n+1)}_{\text{overall}} = \frac{Y_1 + Y_2 + \Delta \cdot \left( \frac{Y_1}{X_1} - \frac{Y_2}{X_2} \right)}{X_1 + X_2}
\]  

(3.5)

Compared with the no-adjustment case which has the same CIPRE as in (3.3), after epoch \( n + 1 \), the CIPRE of all resources is increased by

\[
CIPRE^{(n+1)}_{\text{overall}} - \frac{Y_1 + Y_2}{X_1 + X_2} = \frac{\Delta \cdot \left( \frac{Y_1}{X_1} - \frac{Y_2}{X_2} \right)}{X_1 + X_2}
\]  

(3.6)

Whenever \( CIPRE_1 \) is greater than \( CIPRE_2 \), resources continue to be transferred from thread 2 to thread 1 in subsequent epochs as long as each thread has at least its specified minimum allocation. The overall CIPRE keeps increasing and getting ever closer to \( CIPRE_1 \) (but will not exceed \( CIPRE_1 \)). That is to say, the line for \textit{Aggregate with adjustment} is getting closer to the line for \textit{Thread 1} in Figure 3.2.

As thread 1 obtains more resources, its resource usage efficiency, i.e., \( CIPRE_1 \), will tend to decrease. At the same time, \( CIPRE_2 \) will increase gradually (as the number of resources allocated to a thread reduces, the usage efficiency of the remaining resources will increase). In one situation, \( CIPRE_1 \), \( CIPRE_2 \) and \( CIPRE \) will be getting closer and closer (the lines for \textit{Thread 1}, \textit{Thread 2} and \textit{Aggregate with adjustment} in Figure 3.2 will nearly overlap). Both threads can use the allocated resources at the same efficiency and the CIPRE of all resources reaches its optimal value. Another
situation is that the CIPRE value of thread 1 is still greater than that of thread 2 even when thread 1 already took all the resources it could from thread 2. As mentioned previously, ARPA assures each thread a certain minimum number of resources to avoid resource starvation in such situations.

Although the objective of ARPA is to improve the efficiency of resource utilization, thereby improving overall instruction throughput, our experimental results show that ARPA also provides a good balance between throughput and fairness. A detailed analysis is presented in Section 3.4.

### 3.2.3 Partitioning Algorithm

Figure 3.3 presents the pseudocode of ARPA. At the end of an epoch, the `ComInsts` function computes the CIPRE value of each thread: this is the number of committed
instructions divided by the total number of IFQ and ROB entries allocated to the thread in the current epoch. We then compare the CIPRE of each thread and select the thread with the greatest CIPRE as the reference thread (if the CIPREs of the two threads are equal, we randomly select one of them as the reference thread). The reference thread can take step (i.e., $\Delta$) entries of IFQ and ROB from every other thread. IQ entries are also proportionately moved. This partitioning process is repeated every epoch until the end of the program.

3.2.4 Implementation of ARPA

Figure 3.4 shows the required processor modifications for implementing ARPA. The top layer in Figure 3.4 is the baseline SMT processor structure used in our study. We do not modify this part.

The middle layer lists the counters and comparators we add to the processor for each thread, which will be used for resource partitioning using ARPA. We need one In-flight_Instructions_Counter per thread to monitor the current usage of queue entries by each thread. The counter will be incremented as instructions are fetched and decremented as instructions are committed. The Committed_Instructions_Counters are used to count the committed instructions for each thread in the current epoch. A Committed_Instructions_Counter will be reset to zero at the start of each epoch.
while an $\text{In-flight	extunderscore Instructions	extunderscore Counter}$ will not be reset during the execution of a thread. We use one comparator per thread to determine if the current resource usage of the thread has already exceeded its specified bound; if so, a throttling signal will be generated to throttle further fetching for this thread.

The bottom layer is the implementation of the algorithm. At the end of each epoch, we execute the resource allocation algorithm. The resources upper bound assigned to each thread is saved in its Partition Register. At the start of the program, this assignment is set to be equal for every thread. In every epoch, the Partition Registers will be read and the CIPRE computed for each thread. Based on this value, a new partition will be generated and the Partition Registers will be updated. As was done in [22], we suggest to implement this in software. At the end of each epoch, an interrupt signal can be sent to one of the application threads, using its hardware context to execute the partitioning algorithm. The overhead of running the algorithm is considered in the same way as was done in [22].

3.3 Experimental Setup
3.3.1 Configuration

Our simulator is based on Simplescalar [13] for the Alpha AXP instruction set with Wattch [12] power extensions. We modified SimpleScalar to support simultaneous multithreaded processors. Moreover, we have decoupled the centralized Register Update Unit (RUU) structure adopted by SimpleScalar and have separate issue queue, reorder buffer and physical registers. Our baseline processor configuration is shown in Table 3.1. Other detailed features are based on the SMT architecture of Tullsen et al. [80].

Our simulator adds support for dynamic partitioning of the fetch queue and the reorder buffer. We keep counters for the number of In-flight instructions (which are the instructions in the fetch queue and reorder buffer) per thread, allowing a thread
Table 3.1. SMT simulator settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF,ID,IS Width</td>
<td>8-way</td>
</tr>
<tr>
<td>Pipeline Depth</td>
<td>8 stages</td>
</tr>
<tr>
<td>Queue size</td>
<td>32 IFQ, 80 IQ, 64 LSQ</td>
</tr>
<tr>
<td>Functional Units</td>
<td>6 Int, 4 FP, 4 ld/st</td>
</tr>
<tr>
<td></td>
<td>2 Int Mul/Div, 2 FP Mul/Div</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>256 Int, 256 FP</td>
</tr>
<tr>
<td>Reorder Buffer size</td>
<td>256 entries</td>
</tr>
<tr>
<td>BTB</td>
<td>2048 entries, 4-way associative</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>4K entries gshare, 10-bit global history</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>128KB, 4-way, writeback</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>128KB, 4-way, writeback</td>
</tr>
<tr>
<td>Combined L2 cache</td>
<td>1MB, 4-way associative</td>
</tr>
<tr>
<td>L2 Cache hit time</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Main memory hit time</td>
<td>300 cycles</td>
</tr>
</tbody>
</table>

Table 3.2 lists the benchmarks used in our simulations. All benchmarks are taken from the SPEC2000 suite and use the reference data sets. We use the pre-compiled alpha binaries from C. Weaver source: (www.simplescalar.com); these binaries are built with the highest level of compiler optimization. From these 22 benchmarks, we created multiprogrammed workloads following the methodology proposed in [20, 22, 42].

3.3.2 Workloads

We use the ICOUNT fetch policy to fetch instructions. Other parameters are set as shown in Table 3.1.
<table>
<thead>
<tr>
<th>App</th>
<th># skipped (in millions)</th>
<th>Type</th>
<th>App</th>
<th># skipped (in millions)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>4000</td>
<td>MEM</td>
<td>gcc</td>
<td>1000</td>
<td>ILP</td>
</tr>
<tr>
<td>lucas</td>
<td>2000</td>
<td>MEM</td>
<td>wupwise</td>
<td>2500</td>
<td>ILP</td>
</tr>
<tr>
<td>applu</td>
<td>500</td>
<td>MEM</td>
<td>vortex</td>
<td>0.5</td>
<td>ILP</td>
</tr>
<tr>
<td>equake</td>
<td>3400</td>
<td>MEM</td>
<td>gap</td>
<td>65</td>
<td>ILP</td>
</tr>
<tr>
<td>twolf</td>
<td>400</td>
<td>MEM</td>
<td>mesa</td>
<td>250</td>
<td>ILP</td>
</tr>
<tr>
<td>vpr</td>
<td>1150</td>
<td>MEM</td>
<td>perlmk</td>
<td>500</td>
<td>ILP</td>
</tr>
<tr>
<td>art</td>
<td>2900</td>
<td>MEM</td>
<td>gzip</td>
<td>40</td>
<td>ILP</td>
</tr>
<tr>
<td>swim</td>
<td>250</td>
<td>MEM</td>
<td>crafty</td>
<td>10</td>
<td>ILP</td>
</tr>
<tr>
<td>parser</td>
<td>250</td>
<td>MEM</td>
<td>bzip2</td>
<td>200</td>
<td>ILP</td>
</tr>
<tr>
<td>ammp</td>
<td>2600</td>
<td>MEM</td>
<td>con</td>
<td>3</td>
<td>ILP</td>
</tr>
<tr>
<td>apsi</td>
<td>30</td>
<td>ILP</td>
<td>fma3d</td>
<td>3000</td>
<td>ILP</td>
</tr>
</tbody>
</table>

Table 3.2. Twenty two SPEC CPU2000 benchmarks used in this study.

SPEC benchmarks are first categorized into memory-bound and computation-bound programs (represented by MEM and ILP, respectively, in Table 3.2). Based on the MEM or ILP character of different benchmarks, we created our multiprogrammed workloads with 2-thread and 4-thread combinations as shown in Table 3.3. All the workloads are labeled to indicate the character and number of threads, as well as an index to distinguish one workload from another. MIX workloads select half of their threads from ILP and the other half from MEM. We selected simulation regions of different benchmarks based on [70] as shown in Table 3.2. We stopped simulations after running 400 million cycles.

3.3.3 SMT Performance Metrics

Measuring the performance of a single thread is simple, but for multithreaded workloads things become more complicated. We need to consider not only the overall instruction throughput of the processor but also the fairness accorded to each thread running on the processor. Several metrics [56, 75, 79] have been proposed to measure SMT performance, however, no single measure has emerged as a standard. We therefore show the throughput and fairness results quantified by each of the following three metrics to give a comprehensive comparison of the different algorithms.
<table>
<thead>
<tr>
<th>Name</th>
<th>Combinations</th>
<th>Name</th>
<th>Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM.2.1</td>
<td>applu, anmp</td>
<td>MEM.4.1</td>
<td>ammp, applu, art, mcf</td>
</tr>
<tr>
<td>MEM.2.2</td>
<td>art, mcf</td>
<td>MEM.4.2</td>
<td>art, mcf, swim, twolf</td>
</tr>
<tr>
<td>MEM.2.3</td>
<td>swim, twolf</td>
<td>MEM.4.3</td>
<td>ammp, applu, swim, twolf</td>
</tr>
<tr>
<td>MEM.2.4</td>
<td>mcf, twolf</td>
<td>MEM.4.4</td>
<td>mcf, twolf, vpr, parser</td>
</tr>
<tr>
<td>MEM.2.5</td>
<td>art, vpr</td>
<td>MEM.4.5</td>
<td>art, twolf, equake, mcf</td>
</tr>
<tr>
<td>MEM.2.6</td>
<td>art, twolf</td>
<td>MEM.4.6</td>
<td>equake, parser, mcf, lucas</td>
</tr>
<tr>
<td>MEM.2.7</td>
<td>swim, mcf</td>
<td>MEM.4.7</td>
<td>art, mcf, vpr, swim</td>
</tr>
<tr>
<td>MIX.2.1</td>
<td>applu, vortex</td>
<td>MIX.4.1</td>
<td>ammp, applu, apsi, eon</td>
</tr>
<tr>
<td>MIX.2.2</td>
<td>art, gzip</td>
<td>MIX.4.2</td>
<td>art, mcf, fma3d, gcc</td>
</tr>
<tr>
<td>MIX.2.3</td>
<td>wupwise, twolf</td>
<td>MIX.4.3</td>
<td>swim, twolf, gzip, vortex</td>
</tr>
<tr>
<td>MIX.2.4</td>
<td>lucas, crafty</td>
<td>MIX.4.4</td>
<td>gzip, twolf, bzip2, mcf</td>
</tr>
<tr>
<td>MIX.2.5</td>
<td>mcf, eon</td>
<td>MIX.4.5</td>
<td>mcf, mesa, lucas, gzip</td>
</tr>
<tr>
<td>MIX.2.6</td>
<td>twolf, apsi</td>
<td>MIX.4.6</td>
<td>art, gap, twolf, crafty</td>
</tr>
<tr>
<td>MIX.2.7</td>
<td>equake, bzip2</td>
<td>MIX.4.7</td>
<td>swim, fma3d, vpr, bzip2</td>
</tr>
<tr>
<td>ILP.2.1</td>
<td>apsi, eon</td>
<td>ILP.4.1</td>
<td>apsi, eon, fma3d, gcc</td>
</tr>
<tr>
<td>ILP.2.2</td>
<td>fma3d, gcc</td>
<td>ILP.4.2</td>
<td>apsi, eon, gzip, vortex</td>
</tr>
<tr>
<td>ILP.2.3</td>
<td>gzip, vortex</td>
<td>ILP.4.3</td>
<td>fma3d, gcc, gzip, vortex</td>
</tr>
<tr>
<td>ILP.2.4</td>
<td>gzip, bzip2</td>
<td>ILP.4.4</td>
<td>gzip, bzip2, eon, gcc</td>
</tr>
<tr>
<td>ILP.2.5</td>
<td>wupwise, gcc</td>
<td>ILP.4.5</td>
<td>mesa, gzip, fma3d, bzip2</td>
</tr>
<tr>
<td>ILP.2.6</td>
<td>fma3d, mesa</td>
<td>ILP.4.6</td>
<td>crafty, fma3d, apsi, vortex</td>
</tr>
<tr>
<td>ILP.2.7</td>
<td>apsi, gcc</td>
<td>ILP.4.7</td>
<td>apsi, gap, wupwise, perlbmk</td>
</tr>
</tbody>
</table>

Table 3.3. Benchmark combinations based on cache behavior of threads.

\[ \text{Avg}\_IPC = \frac{\sum \text{IPC}_i}{T} \quad (3.7) \]

The Avg\_IPC metric only quantifies the overall throughput and doesn’t take fairness into consideration. Therefore, using this metric may boost the overall IPC by starving some threads.

\[ \text{Single\_WIPC} = \frac{\sum \frac{\text{IPC}_i}{\text{Single\_IPC}_i}}{T} \quad (3.8) \]

The Single\_WIPC metric weighs the IPC of each thread when running on an SMT with respect to its IPC if run alone, and reflects the fairness accorded to each thread. The drawback of this metric is that it does not assign any importance to the overall throughput and may bias against a thread with very low IPC. For example, consider thread A with single thread IPC of 3.0 and thread B with single thread IPC of 0.1 running simultaneously, with thread A achieving IPC=1.5 and thread B
IPCs=0.09 with the ICOUNT fetch policy, while Static Partitioning achieves IPC=2.1 and IPC=0.06, respectively. The Single_WIPC of Static Partitioning is lower by 7.1% than that of ICOUNT although the overall throughput of Static Partitioning is much better.

\[
Baseline_{WIPC} = \frac{\sum IPC_{new,i}}{IPC_{baseline,i}}\quad (3.9)
\]

The Baseline_WIPC weighs the IPC of each thread with respect to its IPC in the baseline or reference scheme. It reflects the change in IPC of each thread for the optimized scheme compared to the baseline scheme. Regardless of how each thread would run in a single thread mode, Baseline_WIPC benefits from any thread running faster.

3.4 Results and Analysis

We first illustrate the adaptive nature of ARPA through an example. Then, we compare ARPA with other schemes using the above three metrics across the 42 workloads. Finally, we provide a sensitivity analysis of ARPA to its main parameters, namely, step, epoch and the number of queue entries.

3.4.1 Adaptivity of ARPA

Figure 3.5 illustrates the adaptive nature of the resource partitioning by ARPA. Threads twolf, a memory-bound program and apsi, a computation-bound program, are running simultaneously. Figure 3.5(a) displays the CIPRE changes of these two threads for the epochs 1 to 100 when using ARPA, while Figure 3.5(b) shows the resulting partitioning of queue entries between the two threads for epochs 1 to 100.

In the first epoch, we equally partition resources to twolf and apsi, as indicated in Figure 3.5(b). The CIPRE of twolf is higher than that of apsi in this epoch, and consequently in epoch 2, twolf takes \( \Delta = 2 \) queue entries from apsi. We can
Figure 3.5. An example (twolf-apsi) illustrating the adaptive nature of ARPA over a short time period.

Figure 3.6. An example (twolf-apsi) illustrating the adaptive nature of ARPA over a long time period.
see that the CIPRE of \textit{twolf} is larger than that of \textit{apsi} until epoch 49. Therefore, \textit{twolf} takes 2 queue entries from \textit{apsi} at each epoch until epoch 49. The allocated number of queue entries of \textit{twolf} increases linearly while the allocated number of queue entries of \textit{apsi} decreases linearly during this time period. Now the CIPREs of the two threads have become close to each other in epoch 49. In other words, the two threads are using their allocated resources with similar efficiency. Between epoch 49 and 63, a small number of queue entries move back and forth between the two threads. At epoch 64, the number of queue entries allocated to \textit{apsi} reaches its minimum of 36. Although the CIPREs of \textit{twolf} are higher than those of \textit{apsi} most of the time after epoch 64, \textit{twolf} cannot take additional resources from \textit{apsi} in order to prevent resource under-utilization. The number of queue entries becomes stable for each thread and the resources allocated to each thread will remain in this setting if no program phase changes occur.

Figure 3.5 illustrates the tuning process over a short time period (100 epochs). In order to understand the resource adaptation process during a long program execution time, we show in Figures 3.6(a) and 3.6(b), respectively, the CIPRE changes and the corresponding resource allocations of these two threads for the entire program execution that lasts 12206 epochs.

The transitions from a stable phase to another one only last a few tens of epochs and there are different stable resource allocation phases during the execution of the workload combination of \textit{twolf} and \textit{apsi} as indicated in Figure 3.6. The first stable phase comes after the resource tuning process shown in Figure 3.5 and is short. The second stable phase is also short compared to the following five phases; during this phase, the numbers of resources allocated to each thread are close to each other. In the third tuning process phase, the CIPREs of \textit{twolf} are higher than those of \textit{apsi} in most epochs, allowing \textit{twolf} to own more resources to improve resource usage efficiency. Similar tuning processes happen at the start of the next three stable
Figure 3.7. Two additional examples illustrating the adaptive behavior of ARPA.

phases. In the final stable phase, although the CIPRE of twolf is larger than that of apsi all the time, resource allocations are fixed since the number of queue entries of apsi has reached the low-bound limit. Clearly, a static resource partitioning can not satisfy these varied program phases. ARPA retunes the resource allocation whenever program phase changes occur.

The tuning phases of different workloads usually vary significantly. They can be relatively stable with few changes, as shown in Figure 3.6; they can also exhibit frequent changes as shown in Figure 3.7 (a) or aperiodic behavior as shown in Figure 3.7 (b). ARPA can dynamically change the number of resources allocated to threads according to their real resource requirements, thus improving performance significantly. We will discuss the performance results of ARPA in the following section.

3.4.2 Performance Results

Figure 3.8 compares the Single_WIPC of ARPA with those of other schemes across the 42 workloads listed in Table 3.3. I_COUNT [80] is a traditional fetch policy and is therefore suitable as a baseline scheme for comparison. Static Partitioning [68]
partitions resources equally to each thread for the entire program execution. We compare ARPA with it to see the benefits of an adaptive scheme over a non-adaptive one. Hill-climbing [22] has been so far the best resource partitioning scheme achieving significant improvements over previous schemes like STALL/FLUSH++ [19, 79]. To avoid a cluttered figure, we only choose Hill-climbing for comparison. The epoch size we used in these experiments is 32K cycles and the step size is 2 queue entries. We allow each thread to keep at least a quarter of the equally partitioned queue entries to avoid resource starvation.

From Figure 3.8 we see that ARPA outperforms ICOUNT and Static Partitioning significantly in MEM and MIX workloads. For some workloads like MIX.2.5 and MIX.4.2, the improvement of ARPA over ICOUNT and Static Partitioning is more than 50%. The ICOUNT policy gives priority to threads which move faster through the pipeline, i.e., threads which have an efficient resource usage. However, ICOUNT cannot constrain threads from clogging resources, resulting in poor performance when this happens. Because the memory-bound threads in MEM and MIX workloads more readily clog resources than do computation-bound threads in ILP workloads, we can see from Figure 3.8 that the improvement in MEM and MIX workloads is much greater than that in ILP workloads for both 2-thread and 4-thread workloads. Static Partitioning can prevent resource monopolization by a single thread. This charac-
Figure 3.9. Avg_IPC and Single_WIPC improvement of different schemes over ICOUNT across 42 workloads.

...characteristic benefits especially resource tight situations since the possibility of resource monopolization increases when the number of resources reduces. From Figure 3.8 we can see that Static Partitioning achieves higher improvement over ICOUNT in 4-thread workloads than in 2-thread workloads. However, Static Partitioning does not consider program phase changes and the needs of individual threads. As a result, the performance improvement of ARPA over Static Partitioning is considerable.

Hill-climbing [22] is the best-performing published partitioning algorithm. From Figure 3.8, we can see that Hill-climbing outperforms ICOUNT and Static Partitioning significantly in MIX workloads. Using the Single_WIPC metric, Hill-climbing achieves an 8.1% improvement over ICOUNT, close to the results published in [22], increasing our confidence in the precision of our implementation of this algorithm. However, since Hill-climbing does not analyze the behavior of individual threads and makes its decisions based only on periodic trials, there is scope for further improvements. Figure 3.8 shows that ARPA outperforms Hill-climbing in all but 8 of the 42 workloads.

Figure 3.9 shows the Avg_IPC and Single_WIPC improvements of different schemes over ICOUNT across 42 workloads. The figure shows the averages for the MEM, MIX, ILP workloads separately in both 2-thread and 4-thread workloads.

The Avg_IPC improvement of Static Partitioning, Hill-climbing and ARPA over ICOUNT is much better than the Single_WIPC improvement for the MEM and MIX groups. For example, Hill-climbing achieves a 100.3% improvement in Avg_IPC over
ICOUNT, but using the *Single_WIPC* metric, the improvement is only 11.7% for MIX.2.Avg. However, for ILP workloads, the improvements in terms of the two metrics are similar. This is caused by the metric characteristics as explained in Section 3.3.3. Static Partitioning, Hill-Climbing and ARPA control the resource utilization by potentially clogging threads to improve the overall throughput. Since a clogging thread which uses clogged resources for a longer period of time usually has a low single-thread IPC, a small absolute IPC reduction will result in a greater weighted IPC reduction. That is to say, the weighted IPC improvement of one thread can not make up for the weighted IPC loss of another thread for the *Single_WIPC* metric although the overall throughput increases greatly.

The *Avg_IPC* improvements of the 2-thread workloads are less than those of the 4-thread workloads for all three schemes. It is obvious that with the same number of resources, 4-thread workloads result in greater resource clogging than 2-thread workloads. The improvement of MIX workloads is much greater than that of MEM and ILP workloads. By preventing the resource clogging of memory-bound threads in MIX workloads, a resource partitioning scheme can improve the performance of the computation-bound threads over ICOUNT, thus improving the overall throughput. The improvement over ICOUNT is especially significant for aggressive clogging and resource-tight situations.

ARPA performs much better than Static Partitioning and Hill-climbing in MEM and MIX workloads but shows no significant advantages for ILP workloads compared to these two schemes. The reason is that computation-bound threads require fewer resources to exploit ILP, and consequently, Static Partitioning and Hill-climbing achieve nearly optimal solutions.

Figure 3.9 shows that with the *Avg_IPC* metric, Static Partitioning, Hill-Climbing and ARPA achieve 13.5%, 48.5% and 55.8% improvements over ICOUNT, respectively. With the *Single_WIPC* metric, they achieve 6.8%, 8.1% and 14.0% improve-
Figure 3.10. Baseline_WIPC of different schemes for 42 workloads.

...ments over ICOUNT. ARPA achieves a 5.7% improvement over the current best-performing algorithm, Hill-climbing, using both the \textit{Avg}\_IPC and \textit{Single}\_WIPC metrics.

As mentioned in Section 3.3.3, the drawback of the \textit{Single}\_WIPC metric is that it ignores the overall throughput and may bias against threads with very low IPC. However, regardless of how each thread would run in the single thread mode, \textit{Baseline}\_WIPC reflects the improvement of a resource partitioning scheme over the baseline scheme. Figure 3.10 shows the \textit{Baseline}\_WIPC speedup of Static Partitioning, Hill-climbing and ARPA over the ICOUNT baseline. ARPA achieves a better \textit{Baseline}\_WIPC speedup than Static Partitioning and Hill-climbing for almost all workloads in the MEM and MIX groups but the improvement in the ILP group is not as significant. Static Partitioning, Hill-climbing and ARPA achieve 7.8%, 32.6% and 39.1% improvements in the 2-thread workloads, and 18.2%, 39.4% and 48.0% improvements in the 4-thread workloads, respectively. Compared to Static Partitioning, ARPA achieves a 18.5% improvement in \textit{Baseline}\_WIPC while compared to Hill-climbing, it achieves a 9.2% improvement in \textit{Baseline}\_WIPC.
3.4.3 Sensitivity Analysis

In this section, we study the impact of the epoch and step sizes for ARPA. Then, we compare ARPA with other schemes when the amount of resources changes.

3.4.3.1 Epoch and Step Size

Figures 3.11(a) and 3.11(b) show the average IPC improvement of ARPA over ICOUNT as the step size changes from 2 to 54 with epoch size fixed at 256K, and as the epoch size changes from 32K to 65536K with step size fixed at 2. We focus on two representative 4-thread workloads from the 42 workloads.

From Figure 3.11(a) we can see that as the step size increases from 2 to 18, the performance improvement of ARPA over ICOUNT is roughly the same. It drops as we continue to increase the step size beyond 18. For a small step size, ARPA may take a slightly long tuning time (if epoch size is not too big) to reach the optimal
partitioning, with only a small impact on performance. However, if the step size is big, partitioning may miss an optimal level, causing performance loss.

From Figure 3.11(b) we can see that as the epoch size is increased from 32K to 65536K cycles, MIX.4.4 and ILP.4.5 exhibit different IPC improvement trends: the performance improvement of MIX.4.4 reduces significantly as epoch size is increased beyond 4096K, while for ILP.4.5, the performance improvement is roughly constant for all tested epoch sizes. The reason is that ILP.4.5 performs well when resources are equally partitioned; since ARPA starts with equal resource partitioning, there is no significant performance decrease even with large epoch sizes for ILP.4.5. However, equally partitioned resources are not ideal for MIX.4.4. ARPA cannot adjust to thread behavior changes accurately when using a big epoch size, causing a performance loss.

### 3.4.3.2 Queue Entries

We now examine the impact of the size of LSQ, IQ and ROB on the performance of different schemes.

Figure 3.12 shows the average IPC of ARPA versus ICOUNT, Static Partitioning and Hill-climbing when LSQ, IQ and ROB increase from (32, 40, 128) to (96, 120, 384). The results in this figure and in Figure 3.13 are for 12 out of the 42 workloads in Table 3.3.
From Figure 3.12 we can see that the average IPC improves when the number of queue entries increases from (32, 40, 128) to (96, 120, 384) for each scheme. However, the degree of improvement declines when the number of queue entries is increased from (64, 80, 256) to (96, 120, 384) due to the ILP limitation of the workloads. As the number of queue entries is increased to (96, 120, 384), the average IPCs of Static Partitioning and ICOUNT draw close to each other since an increased number of resources reduces the resource clogging of ICOUNT.

ARPA and Hill-climbing perform much better than ICOUNT and Static Partitioning in all resource sizes since a flexible dynamic resource partitioning can combine the resource sharing benefit of ICOUNT and the resource partitioning benefit of Static Partitioning, thus improving performance significantly.

The average IPCs of ARPA and Hill-climbing at (32, 40, 128) are close to each other, while as the number of queue entries increases, the performance improvement of ARPA is bigger than that of Hill-climbing. The reason is that Hill-climbing is a learning-based algorithm and the increased number of resources will increase the search space, thereby increasing the learning time and the chances of being trapped in a local maxima.

Figure 3.13 compares the Single_WIPC improvement of different schemes over ICOUNT as the number of queue entries changes for 12 out of the 42 workloads in Table 3.3.
From Figure 3.13 we can see that as the number of resources increases, the improvements over ICOUNT decrease for almost all the schemes since the increased number of resources mitigates the resource clogging problem. The performance improvement of ARPA over ICOUNT is the biggest among all the schemes in any number of queue sizes, achieving 21.7%, 16.9% and 17.1%, respectively, while Hill-climbing only achieves 13.4%, 7.8% and 6.8% improvements.

3.5 Chapter Summary

In this chapter, we presented a utility-based Adaptive Resource Partitioning Algorithm (ARPA) for SMT processors. The algorithm identifies the resource usage efficiency of each thread using the CIPRE metric and allocates more resources to threads which can use them in a more efficient way. Our experimental results show that ARPA improves $\text{Avg}_{\text{IPC}}$ by 55.8% over ICOUNT, while Static Partitioning only achieves a 13.5% improvement over ICOUNT. Compared with the currently best-performing algorithm Hill-climbing, ARPA achieves a 5.7% $\text{Avg}_{\text{IPC}}$ improvement.

Allocating more resources to threads which can use them more efficiently does not always mean giving more resources to threads with a higher IPC. In fact, ARPA is an adaptive process that allows threads to share resources more fairly and efficiently. With respect to the $\text{Single}_{\text{WIPC}}$ metric, ARPA achieves a 14.0% improvement over ICOUNT and attains 6.8% and 5.7% improvements over Static Partitioning and Hill-climbing, respectively. With the $\text{Baseline}_{\text{WIPC}}$ metric, ARPA attains 43.6%, 18.5% and 9.2% improvements over ICOUNT, Static Partitioning and Hill-climbing, respectively.
CHAPTER 4

RESOURCE ADAPTATION FOR HIGH PERFORMANCE AND LOW ENERGY IN SMT PROCESSORS

As presented in chapter 3, a side-effect of resource sharing by multiple threads in SMT processors is that the overall performance may deteriorate due to resource monopolization. Resource partitioning approaches [20, 22, 27, 52, 68, 85] have been developed to reduce the resource conflicts among threads, thus improving performance. However, these are usually not energy-efficient. No matter how varied resource requirements of multi-threaded workloads may be, all available resources are allocated, resulting in a significant waste of energy.

In this chapter, we present a resource adaptation approach to adaptively control the number of powered-on ROB entries and partition shared resources among threads, targeting both performance and energy. Our approach considers not only the relative resource usage efficiency of each thread (assigning more resources to threads which can use them more efficiently), but also takes into account the real resource usage of threads to identify cases of inefficient resource usage. Energy savings are achieved by powering off resources that do not improve the performance and only waste power. Our experimental results show that for an SMT processor with a shared-ROB structure, our resource adaptation approach achieves 16.7% energy savings over ARPA, while the performance loss is negligible across 42 sample workloads. For an SMT processor with a divided-ROB structure, our resource adaptation approach outperforms ARPA by 4.2% in addition to achieving 12.4% energy savings.
4.1 Introduction

Modern microarchitectural techniques [36], such as out-of-order processing of instructions, register renaming, and deep pipelines, have boosted single-thread performance dramatically. However, a side-effect has been a significant increase in power consumption [11, 33, 34] as the amount of hardware required to implement these techniques has increased considerably. Furthermore, the need for hardware resources such as caches, instruction queues, and renaming registers can vary significantly from one program to another and even within the different execution phases of the same program. Thus, resources that can enhance the performance of a program during one phase, may become idle during other phases and only waste energy.

Simultaneous Multithreading is an indirect approach for improving the energy efficiency of modern microprocessors. By allowing instructions from multiple threads to share processor resources by executing them in the same pipeline, SMT processors can significantly increase the resource usage efficiency and improve the overall instruction throughput, thus providing substantial energy benefits [47, 50, 51].

Compared to the single-thread case, resource requirements of multi-threaded workloads running on an SMT processor have a higher variability. In Figure 4.1, we present an example showing the significant variations in resource requirements of both memory- and computation-bound workloads. art.mcf consists of the two memory-bound benchmarks art and mcf; fma3d.mesa consists of the two computation-bound benchmarks fma3d and mesa. When the ROB size increases from 96 to 256 entries, the Weighted-IPC of art.mcf improves by almost 30%, while there is no big performance improvement for fma3d.mesa. However, the energy consumption increases linearly with the ROB size for fma3d.mesa. Considering both energy and performance, a 96-entry ROB is ideal for fma3d.mesa, while a 256 entry configuration is better for art.mcf. Therefore, architecture adaptation [88] which has been proposed for single-thread processors [4, 18, 65] to save energy may also be beneficial for SMT processors.
Architecture adaptation adjusts the activation of hardware resources in accordance with the changes in program behavior. Energy savings are achieved by powering off resources that do not contribute to the performance and only waste power.

In this chapter, we present a resource adaptation scheme for SMT processors, which targets both high performance and low energy consumption. Our approach not only partitions resources among threads based on their relative resource usage efficiency, but also adaptively controls the number of powered-on resources based on the real resource requirements in order to save energy.

We focus on the dynamic instruction scheduling logic (IQ, LSQ, ROB and renaming registers) to perform architecture adaptation, since these are the most power-hungry components, consuming between them about 55% of the total power [15, 65]. In some designs like Pentium III, the ROB is integrated with the renaming registers to support register renaming. The ROB can consume more than 27% of the total power [29, 45]. Adaptation techniques can be applied either to only one resource such as IQ [18] or ROB [66], or to multiple resources simultaneously [2, 64, 65] to reduce energy consumption. Our algorithm attempts to control the number of powered-on ROB entries which act as the renaming registers as well. By adaptively tuning the number of ROB entries, instructions can be fetched as needed. This not only reduces resource
competition among threads but also achieves significant dynamic power savings in units like IQ logic by avoiding unnecessary wake-up and selection operations.

The detailed resource adaptation implementations for SMT processors with divided-ROB structure and with shared-ROB structure are different. We present resource adaptation algorithms for both structures. We try to use as few resources as possible to achieve significant performance improvement and reduce the energy consumption. Our experimental results show that for an SMT processor with a shared-ROB structure, our resource adaptation approach achieves 16.7% energy savings over ARPA, while the performance loss is negligible across 42 sample workloads. For an SMT processor with a divided-ROB structure, our resource adaptation approach outperforms ARPA by 4.2% in addition to achieving 12.4% energy savings.

4.2 Resource Utility Analysis

The framework of our resource adaptation is similar to ARPA. We divide the entire program execution into fixed-sized epochs and start with resources being equally partitioned among the threads. After each epoch, we analyze the current resource usage to identify whether the threads have used their allocated resources efficiently in this epoch, and adjust the number of powered-on resources and do resource assignment appropriately. The detailed allocation mechanisms will be different for shared and divided ROB structures.

According to ARPA, whenever the CIPRE value of one thread is greater than the CIPRE value of another thread, resources continue to be transferred in subsequent epochs as long as each thread has at least its specified minimum allocation. Although this approach can substantially improve the overall instruction throughput, it is not necessarily energy-efficient. The CIPRE metric is a relative concept (relative to other threads sharing the resources). It is possible that the actual resource usage is very low. In such cases, we do not need to assign all resources to threads. Our resource
adaptation approach allocates resources considering both performance and energy-efficiency.

4.3 Resource Adaptation

In this section, we first introduce a multi-banked ROB structure which facilitates resource adaptation. Then, we present the detailed resource adaptation mechanisms for shared-ROB SMT processors and divided-ROB SMT processors, individually.

4.3.1 Multi-banked Structure

The physical implementation of the ROB in an SMT processor can either be a single buffer with multiple head and tail pointers [20, 22, 68], one for each thread, or a separate buffer per thread [72, 80]. In a shared-ROB implementation, instructions for each thread are dispatched to the free ROB entries after that thread’s tail pointer in program order and are committed starting from its head pointer, thus maintaining logical correctness. Each thread can use more than its equally partitioned share. This approach is helpful when some threads require more than their equally partitioned number of ROB entries, while others can do quite well with a smaller allocation. However, the physical implementation of shared ROB is more complicated than that of the divided ROB. For the divided ROB structure, each thread has its private ROB as in a single-thread processor. For such a structure, the ROB usage is obviously not as flexible as for the shared ROB, thus potentially impacting performance. We propose resource adaptation schemes for both shared and divided ROB structures.

In order to dynamically control the number of powered-on ROB entries to save energy, large ROBs (for either divided ROB or shared ROB structures) are partitioned into independent banks. Each bank has its own precharger, sense amplifiers and input/output drivers and can be powered on or off to save energy according to changes in program behavior. To power off a bank, the bypass switch of the bank is turned
on and the power supply to the bank is disabled; to power on a bank, the bypass switch is turned off and the power supply to the bank is enabled. Note that a bank can be powered off only after all the instructions residing in it have been committed. We define the power-off latency of a bank as the time duration from receiving the deallocation signal to the time the power supply is disabled. Once a bank has been deallocated, instructions cannot be dispatched to it anymore. A detailed hardware implementation is described in [65].

Banks can be selected for deallocation in two ways. [65, 72] deallocate banks from the highest index to the lowest index sequentially. Such an in-order deallocation policy may have high power-off latencies. We can also deallocate banks according to their expected power-off latencies. It is obvious that an empty bank has the shortest power-off latency while a bank with tail pointers has the longest latency since it must wait until the tail moves out of the bank and all the instructions in the bank are committed.

Allocation/deallocation of banks in a multi-banked ROB structure must be done so that the FIFO characteristic of the ROB is preserved. In Figure 4.2 we use a single
thread example to illustrate the different situations that must be dealt with. Figure 4.2 lists several representative locations of to-be-allocated and to-be-deallocated banks in a three-bank ROB. The dispositions of ROB head and tail are also shown in the figure. The blank banks with the □ mark at the top corner in Figure 4.2(a) are currently powered-off and will be allocated and powered-on in the next cycle. In cases (a1) and (a4), the Head and Tail can move to the allocated banks to dispatch instructions as normal. However, in (a2) and (a3), the Head needs to jump over the allocated banks in order to commit instructions correctly. The banks with the ■ mark at the top corner in Figure 4.2(b) are deallocated and will be powered-off if all the instructions in the banks are committed. In (b1) and (b4), the blank banks can be immediately powered off once the deallocation decision was made. However, in (b2) and (b4), they can be powered off only when all the instructions have been committed. Moreover, no Tail can be moved to these banks to dispatch instructions during this time. Therefore, in order to maintain the correct FIFO order of ROB, care must be taken when the pointers move to the newly allocated/deallocated banks.

In order to control the movements of the ROB pointers, we add one bit per bank which represents the allocation/deallocation state of the bank, and one bit per thread per bank which represents whether the thread has instructions residing in the bank. We name them Allocated and Vacant, respectively. If we decide to allocate a bank, the Allocated bit of the bank will be set to 1. Similarly, if we decide to deallocate a bank, the bit of this bank will be set to 0 immediately. The purpose of the Vacant bit is to indicate whether the thread has no instructions in the bank that await to be committed. If the thread has no instructions residing in the bank, the Vacant bit of this thread in the bank will be 0. Only when the Allocated bit and the Vacant bits of all threads of a bank are zero, can the bank be powered off.

Figure 4.3 shows the transformation of different states for each bank. There are 4 states in the state machine: (1,0), (1,1), (0,0) and (0,1). The first index represents
Table 4.1. The movements of pointers from the current bank to the Next Bank (NB)

<table>
<thead>
<tr>
<th>NB State</th>
<th>Tail Movement</th>
<th>Head Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>do not move</td>
<td>move to NB</td>
</tr>
<tr>
<td>(0,0)</td>
<td>jump over the NB</td>
<td>jump over the NB</td>
</tr>
<tr>
<td>(1,0)</td>
<td>move to the NB</td>
<td>jump over the NB</td>
</tr>
<tr>
<td>(1,1)</td>
<td>move to the NB</td>
<td>move to the NB</td>
</tr>
</tbody>
</table>

the Allocated bit and the second is the Vacant bit of one thread. If there are multiple threads, the number of Vacant bits is equal to the number of threads. (1,0) represents the case where the bank is powered-on and has no instruction in it. (1,1) represents a powered-on bank with at least one instruction of the thread in the bank. (0,0) means the bank is in powered-off state. (0,1) means the bank is deallocated and waiting to be powered off.

At the beginning, all the banks are in the (1,0) state. After the first instruction is dispatched in a bank, its state will change to (1,1). If all the instructions are committed, the state will change back to (1,0) again. The state of the bank will keep oscillating between (1,0) and (1,1) if no allocation/deallocation signal is received. Deallocation can happen from both (1,0) and (1,1) states. At the (1,0) state, if a deallocation signal is received, we can power off the bank immediately since this is an empty bank. At the (1,1) state, if a deallocation signal is received, the state will be...
changed to (0,1) immediately. However, only when all the instructions in this bank have been committed, will the system state be changed to (0,0) and the bank be powered off. Therefore, the state transitions for a full deallocation of a used bank will be (1,1)→(0,1)→(0,0). Allocation to power on a bank and dispatching instructions to it, can only be applied to a powered-off bank, i.e., a bank in state (0,0). The state will change from (0,0) to (1,0). Once an instruction has been placed there, the state changes to (1,1).

In order to maintain the correct execution order, the movements of ROB pointers from one bank to the next bank must take into consideration the state of the next bank. Table 4.1 lists all the possible movements of pointers. There are three different actions to choose from. The first moves to the next bank and continues to dispatch instructions in this bank as normal; the second keeps the pointer at its current position and waits until all the instructions in the next bank are committed; The third jumps over the next bank and finds other available entries in the following banks. For example, when the state of the Next Bank (NB) is (0,1) (this means that the next bank is in the deallocating state), the Tail should not move to this bank anymore, while the Head should move to the bank to commit the instructions.

Our experimental results have shown that our allocation/deallocation policy has on average 66% shorter power-off latencies than that of the in-order policy for shared-ROB structure. However, in practice, the power-off latencies for both polices have a negligible energy impact.

4.3.2 Resource Adaptation for Shared-ROB Structure

Resource adaptation in an SMT processor with a shared-ROB structure consists of upsizing or downsizing the available large ROB space and partitioning the shared resources among threads. Both relative resource usage efficiency of each thread based
on the CIPRE metric and the real dynamic usage of ROB entries are evaluated to create a new partitioning for each epoch.

At each epoch, we count the number of full ROB occupancy cycles to determine whether we need to upsize or not. If the ROB is fully occupied in most cycles, it means that threads may face a shortage of ROB entries and we turn on a powered-off bank and allocates additional entries to the thread with the highest CIPRE value since this is the thread that is likely to use the additional resources most efficiently.

When the number of full ROB occupancy cycles is low in a given epoch, it does not necessarily mean that resources are redundant and should be downsized. Some threads may still need the available resources to achieve high performance. We coordinate with other hints to decide whether to downsize or only repartition resources.

As mentioned before, ARPA assigns more resources to the thread with the highest CIPRE value in order to improve the overall resource usage efficiency. However, it is possible that the thread with the highest CIPRE value already holds enough resources, and additional resources will not be used as efficiently and can instead be powered off to save energy.

To determine this we check whether the in-flight instructions (the instructions in the fetch queue and reorder buffer) of this thread rarely use up all the allocated resource entries. If this is the case, we will not allocate additional resources to this thread even if its CIPRE value is the highest among all the threads. These resources will instead be powered off to save energy.

Figure 4.4 presents the pseudocode of resource adaptation for an SMT processor with a shared-ROB structure. The code in the shaded box is responsible for either down- or up-sizing ROB space. Our resource adaptation follows the similar general partitioning approach as ARPA, assigning more resources to threads which can use resources more efficiently. This ensures performance improvement. Moreover, our
#define Num Number of running threads
#define ComInsts(x) Compute CIPRE[x]
#define max(A,n) Get index of the max value in array A[0:n]
#define STEP Number of queue entries moved at each epoch
#define NumOffBank Number of powered-off banks
#define Bank_size Number of ROB entries per bank

For every epoch cycles{
    for(tid = 0; tid < Num; tid ++)
    {
        /* compute CIPRE of each thread */
        CIPRE[tid] = ComInsts(tid);
        Ref_tid = max(CIPRE, Num); /* select reference thread */
        if(full_ROB > Threshold1)
        {
            /* power on a ROB bank */
            NumOffBank --;
            Partition[Ref_tid] += Bank_size;
        }
        else
        {
            /* power off a ROB bank */
            if(full_in_flight[Ref_tid] < Threshold2)
            {
                for(tid = 0; tid < num; tid ++)
                    if(tid != Ref_tid)
                        Partition[tid] -= Bank_size/(num-1);
                NumOffBank ++;
            }
            else /* resource reallocation */
            {
                for(tid = 0; tid < num; tid ++)
                    Partition[Ref_tid] += STEP;
                Partition[Ref_tid] = STEP;
            }
        }
    }
}

Figure 4.4. Pseudo-code of resource adaptation for an SMT processor with a shared-ROB structure

approach tunes the allocated ROB space based on the real dynamic resource usage to save energy.

At the end of each epoch, the ComInsts function computes the CIPRE value of each thread: this is the number of committed instructions divided by the total number of IFQ and ROB entries allocated to the thread in the current epoch. Then, we compare the CIPREs of all threads and select the thread with the highest value as the reference thread (if two CIPREs are equal, we randomly select one of them as the reference thread).

Whether we change the size of the ROB space or only repartition it is determined by two metrics: full_ROB, which is the number of cycles when the ROB is fully occupied during an epoch, and full_in_flight for the reference thread, which is the number of cycles when the in-flight instructions of the reference thread fully occupy all the allocated entries during an epoch. If full_ROB is larger than Threshold1,
upsizing is performed. The new ROB entries will be allocated to the reference thread since this is the one that can use them most efficiently. If full_ROB is less than Threshold1 and full_in_flight of the reference thread is less than Threshold2, we turn off the ROB entries which would be allocated to the reference thread (we set the total number of entries taken from other threads to be equal to the bank size). In other cases, the normal resource partitioning will be performed: the reference thread can take STEP (same as Δ in Equation 3.5) entries of IFQ and ROB from every other thread. IQ entries are also proportionally reassigned.

4.3.3 Resource Adaptation for Divided-ROB Structure

In this section, we propose a modified resource adaptation scheme for an SMT processor with a divided-ROB structure. It follows an adaptation principle similar to that for the shared-ROB structure. We control the number of powered-on ROB banks for each thread by evaluating its resource usage efficiency. If a thread has a higher CIPRE metric, it means that this thread has a relatively higher usage efficiency than other threads and has the potential to improve performance if we add more resources to it. In contrast, if a thread has a lower CIPRE metric, it means that this thread has a relatively lower usage efficiency than other threads and powering off part of its resources may not decrease the performance. The final resource adaptation decision is made by this relative resource usage efficiency analysis together with the real resource usage efficiency of that thread. We use full_ROB, which is the number of cycles when a thread fully occupies its allocated ROB entries during an epoch to measure whether we need to power on a ROB bank for it, if it is a thread with the highest CIPRE value or power off a ROB bank for it, if it is a thread with the lowest CIPRE value. Other shared resources will be partitioned proportionally to their ROB allocations.

Figure 4.5 shows the pseudo-code of resource adaptation for an SMT processor with a divided-ROB structure. At the end of each epoch, the ComInsts function com-
# define Num  Number of running threads
# define ComInsts(x)  Compute CIPRE[x]
# define max(A,n)  Get index of the max value in array A[0:n]
# define min(A,n)  Get index of the min value in array A[0:n]
# define NumOffBank  Number of powered-off banks
# define Bank_size  Number of ROB entries per bank

For every epoch cycles{
    for(tid = 0; tid < Num; tid++)  /* compute CIPRE of each thread */
        CIPRE[tid] = ComInsts[tid];
    max_tid = max(CIPRE, Num);  /* select thread with max CIPRE */
    if(full_ROB[max_tid] > Threshold 1)/* upsizing */
        Allocated_ROB[max_tid] += BANK_SIZE;
        NumOffBank[max_tid]--;
    }
    min_tid = min(CIPRE, Num);  /* select thread with min CIPRE */
    if(full_ROB[min_tid] < Threshold 2)/* downsizing */
        Allocated_ROB[min_tid] -= BANK_SIZE;
        NumOffBank[min_tid]--;
}

Figure 4.5. Pseudo-code of resource adaptation for an SMT processor with a divided-
ROB structure

Computes the CIPRE value of each thread: this is the number of committed instructions
divided by the total number of IFQ and powered-on ROB entries of the thread in the
current epoch. Then, we compare the CIPREs of all threads and select the thread
with the highest CIPRE value and also the thread with the lowest CIPRE value. If
full_ROB of the thread with the highest CIPRE is above Threshold1, we will power
on an ROB bank and add them to the thread. If full_ROB of the thread with the
lowest CIPRE is below Threshold2, we will power off a ROB bank. It is possible to
select more than two threads (one with the highest CIPRE value and one with the
lowest CIPRE value) for tuning the ROBs if there are more than two threads running
simultaneously. We have done experiments choosing all threads whose CIPREs are
within the range of 10% of the highest CIPRE or 10% of the lowest CIPRE. The
obtained results were similar to those for only two threads.

Note that using only full_ROB to drive resource adaptation is not enough in SMT
processors. A memory-bound thread can have heavy ROB occupancy and adding
it more ROB entries may result in clogging of other shared resources. Therefore,
it is important to combine the relative resource usage efficiency (CIPRE) with real resource usage efficiency (full_ROB) to perform resource adaptation.

### 4.3.4 Implementation of Resource Adaptation

Figure 4.6 shows the processor modifications required for implementing the resource adaptation scheme. The top layer in Figure 4.6 is the baseline SMT processor structure used in our study. The middle layer lists the counters and comparators we add to the processor for each thread. The counters marked with dashed-line boxes are the same for both shared-ROB and divided-ROB SMT processors. We need one In-flight Instructions Counter per thread to monitor the current usage of queue entries by each thread. The counter will be incremented as instructions are fetched and decremented as instructions are committed. The Committed Instructions Counters are used to count the committed instructions for each thread in the current epoch. The Committed Instructions Counter will be reset to zero at the start of each epoch while the In-flight Instructions Counter will not be reset during the execution of a thread. We use one comparator per thread to determine if the current resource usage of the thread has already exceeded its specified bound; if so, a throttling signal will be generated to throttle fetching for this thread. The counters in solid-line boxes are dif-
different for a SMT processor with different ROB structures. The Full-in-flight_counters are only added for the shared-ROB structure to count the number of cycles when the in-flight instructions of the reference thread fully occupy all the allocated entries during an epoch. If the comparator shows that the number of in-flight instructions of the thread exceeds its bound, the Full-in-flight_counter of this thread is incremented by one. We only need a Full-ROB_Counter to monitor the full ROB usage rate in an epoch for a shared-ROB SMT processor while we need one Full-ROB_Counter per thread for a divided-ROB SMT processor. This counter will be incremented by one whenever the dispatch stage notices that the ROB is full in that cycle.

The bottom layer represents the implementation of resource adaptation. At the end of each epoch, we run the architecture adaptation or resource partitioning algorithm. The Partition Registers save the resources’ upper bound assigned to each thread. At the start of the program, they are set to be equal. In every epoch the Partition Registers will be read and the CIPRE computed for each thread. Based on this value, a new partition will be generated and these new partition values will be updated in the Partition Registers.

### 4.4 Evaluation Methodology

The SMT simulator we used in this chapter is as same as that in chapter 3 except that we made some additional changes in order to support architecture adaptation. We maintain a counter for the number of cycles for which the ROB is full and another for the number of cycles when in-flight instructions fully occupy the allocated resources per thread. We modified Wattch to calculate the energy consumption of this adaptive ROB structure. The processor configuration and benchmarks we used are similar to those described in chapter 3. We use the ICOUNT fetch policy to fetch instructions. Other parameters are set as shown in Table 3.1.
The multiprogrammed workloads we used in the experiments are shown in Table 3.3. We select simulation regions of different benchmarks based on [70] and stop the simulations after running 400 million cycles for any workload.

### 4.5 Results and Analysis

In this section, we compare performance and energy results of our resource adaptation approach with those of other schemes for the two different ROB organizations across the 42 sample workloads as shown in Table 3.2.

#### 4.5.1 Shared-ROB Structure

We compare our approach, Resource Adaptation for Shared-ROB (RASR), against 4 different schemes: ICOUNT [80], Hill-climbing [22], ARPA [85] and ARPA-Min_E. ICOUNT is a traditional fetch policy and we select this as our baseline. ARPA-Min_E is an ideal case of ARPA. We tested ARPA using between 96 and 256 ROB entries: ARPA-Min_E is the one with the lowest energy consumption for the given workload, and is used for comparison purposes only.

We use an epoch size of 8K cycles, a STEP size of 2 queue entries for both ARPA and RASR. The bank size we used for RASR is 8. There are in all 32 banks since the total number of ROB entries we used in our experiments is 256. We set both Threshold1 and Threshold2 to 3K.

#### 4.5.1.1 Performance Evaluation

Because RASR follows a resource partitioning principle similar to that used by ARPA, the Single_WIPC improvement of RASR over ICOUNT is very close to that of ARPA for most workloads. RASR achieves 9.8% and 14.2% improvement over ICOUNT in 2-thread and 4-thread workloads, respectively. Compared to ARPA, there is only an average of 1.8% performance loss across 42 workloads. However,
the energy savings of RASR compared with ARPA is very significant. A detailed comparison is provided in the next section.

From Figure 4.7 we can also see that the Single_WIPC of ARPA-Min_E is much worse than that of RASR for most workloads. The average Single_WIPC of ARPA-Min_E is even slightly worse than that of ICOUNT for 2-thread workloads. ARPA cannot achieve a good Single_WIPC in an ROB setting which is good for energy savings.

### 4.5.1.2 Energy Savings

Figure 4.8 compares the energy consumption of different schemes across the 42 workloads listed in Table 3.3. We normalized the energy consumption of each workload with respect to ICOUNT for all the schemes. As we can see, the energy consumption of ICOUNT is much higher than that of Hill-climbing and ARPA for MEM and MIX workloads, while it is close to the others for ILP workloads. The reason is that the performance of ICOUNT is much lower than that of Hill-climbing and ARPA because of resource clogging for MEM and MIX workloads, and close to the others.
for ILP workloads as discussed above. The average energy consumption of ARPA is close to that of Hill-climbing across 42 workloads.

By controlling the number of powered-on ROB entries, RASR fetches instructions just as needed. This not only saves the static and dynamic power associated with active ROB entries but also the dynamic power of units like IQ logic. RASR achieves significant energy savings over ICOUNT, Hill-climbing and ARPA. Compared to ICOUNT, it achieves 24.7% and 31.2% energy savings for 2-thread and 4-thread workloads, respectively. Compared to ARPA, RASR achieves 14.5% and 18.9% energy savings for 2-thread and 4-thread workloads, respectively, while keeping the performance loss very low. By analyzing the thread behavior and adaptively adjusting the number of ROB entries to the minimum requirements of the threads, the energy consumption of RASR is very close to the ideal scheme ARPA-Min_E.

Figure 4.9 shows the average percentage of powered-off banks for RASR and ARPA-Min_E for the MEM, MIX and ILP workloads separately for the 2-thread and 4-thread workloads. The percentage of powered-off banks is highest for the ILP groups and lowest for the MEM groups for both 2-thread and 4-thread workloads.
The reason is that memory-bound workloads need more ROB entries to exploit instruction level parallelism than computation-bound workloads. The percentage of powered-off banks in 4-thread workloads is higher than that in 2-thread workloads since 4-thread workloads have higher thread level parallelism. RASR manages to power off, on average, 45% ROB banks across 42 workloads. Although ARPA-Min_E can power-off more banks than RASR, its performance loss is significant.

4.5.1.3 Energy-to-Single_WIPC-Ratio Improvements

SMT design should care not only about the overall instruction throughput but also about the fairness with which thread is treated. The energy consumption results

Figure 4.9. The percentage of power-off banks of RASR and ARPA-Min_E across 42 workloads

Figure 4.10. Energy-to-Single_WIPC-Ratio (E-WIPC-R) of different schemes for 42 workloads normalized with respect to ICOUNT
in Section 4.5.1.2 reflect only the instruction throughput improvements of different schemes over ICOUNT. In order to take both throughput and fairness into consideration, we now compare RASR to different schemes using an Energy-to-Single_WIPC-Ratio (E-WIPC-R) metric.

Figure 4.10 shows the E-WIPC-R results of the four schemes for 2-thread and 4-thread workloads. All the results are normalized with respect to ICOUNT. As we can see, the E-WIPC-R improvements of all schemes over ICOUNT are highest for MIX workloads and lowest for ILP workloads. The reason is that resource partitioning techniques prevent memory-bound threads from clogging the resources, resulting in significant performance improvement of computation-bound threads for the MIX workloads. This can significantly improve both overall instruction throughput and Single_WIPC.

Without architecture adaptation like RASR, ARPA also achieves higher E-WIPC-R improvement over ICOUNT than Hill-climbing because the Single_WIPC of ARPA is higher than that of Hill-climbing in most workloads.

With both architecture adaptation and resource partitioning according to program behavior changes, RASR achieves significant E-WIPC-R improvements over ICOUNT, Hill-climbing and ARPA. Compared to ICOUNT, it achieves 29.2% and 38.3% E-WIPC-R improvement for 2-thread and 4-thread workloads, respectively. Compared to ARPA, RASR achieves 11.9% and 17.3% E-WIPC-R improvement for 2-thread and 4-thread workloads, respectively. Although ARPA-Min_E has the highest energy savings among all the schemes, the E-WIPC-R improvement over ICOUNT is lower than that of RASR since the Single_WIPC of ARPA-Min_E is much worse than that of RASR.
In this section, we use an example to illustrate why RASR can achieve significant energy savings with negligible performance loss over ARPA as shown above. Figures 4.11 (a) and 4.11 (b) display the partitioning of queue entries of ARPA and RASR between the two simultaneous running threads, \texttt{fma3d} and \texttt{mesa}. The whole program execution lasts 48824 epochs.

As indicated in Figure 4.11, RASR has the same resource tuning phases (seven stable phases) as ARPA. This helps RASR to achieve performance improvement similar to ARPA since the interactive nature of the two threads has not been changed. In contrast with ARPA, RASR tunes resources according to the real usage efficiency as well. As shown in Figure 4.11(a), ARPA partitions all the resources to threads no matter whether or not they are used efficiently by the threads, while RASR powers off a large number of queue entries to save energy in each phase as shown in Figure 4.11(b). During phase 1, both threads can give up a large percentage of resources and retain an efficient resource usage. During phases 2, 4 and 6, \texttt{fma3d} keeps more resources than needed. Therefore, some resources which would be allocated to it by
Figure 4.12. Energy Savings of RASR over ARPA as the number of bank size changes for 12 out of the 42 workloads.

ARPA are powered off to save energy. During phases 3, 5 and 7, similarly, some resources for *mesa* can be powered off to save energy.

4.5.1.5 Sensitivity Analysis

In this section, we study the impact of the three main parameters of our algorithm, namely, bank size, Threshold1 and Threshold2. We concentrate on 12 out of the 42 workloads in Table 3.3.

Figure 4.12 shows the energy savings of RASR over ARPA as the bank size changes from 8 to 64 entries. As we increase the bank size from 8 to 32 entries, the energy savings of RASR over ARPA decreases for most benchmarks. It is obvious that we can do a more fine-grained adaptation with a small bank size, thereby following program behavior more closely. However, the energy consumption of RASR increases significantly when the bank size is 64 entries. It is even higher than that of ARPA for MIX.2.5 and MIX.4.7, since a big bank size will cause greater performance loss, thus increasing the energy consumption. As the bank size decreases to a very low value, the area used to implement the banked ROB components and the hardware implementation complexity of multi-banked structure increase significantly. Since banks of size 8, 16 and 32 do not differ greatly in energy consumption, it would be better to select a bank size of 32.
Figure 4.13. Energy-to-Single_WIPC-Ratio (E-WIPC-R) improvement of RASR over ARPA for different values of Threshold1 and Threshold2 for 12 out of the 42 workloads.

As we have seen in Section 4.3.2, RASR uses the number of full ROB occupancy cycles during an epoch (compared to Threshold1) and the number of cycles when in-flight instructions of the reference thread fully occupy the allocated queue entries in an epoch (compared to Threshold2) to decide whether an architecture adaptation or resource partitioning is performed. We now examine how these two thresholds impact the Single_WIPC and energy consumption of RASR. Combining Single_WIPC and energy consumption, we use the E-WIPC-R as a metric.

The top portion of Figure 4.13 shows the E-WIPC-R improvement of RASR over ARPA as both Threshold1 (T1) and Threshold2 (T2) increase from 0K to 6K, with a step size of 3K. As both thresholds increase, the architecture adaptation becomes more aggressive. As we can see, when T1 increases from 0K to 3K, the E-WIPC-R improvements of RASR increase significantly for any T2 value. However, as T1 increases from 3K to 6K, T2 plays a significant role and for most benchmarks (see the top portion of Figure 4.13), the E-WIPC-R improvement decreases significantly. Therefore, setting T1 to 6K is too aggressive. Among all the tested settings, RASR achieves the best E-WIPC-R improvements for T1=T2=3K.

We also tested RASR for both T1 and T2 between 2K to 4K, with a step size of 1K. The bottom part of Figure 4.13 shows that the difference is small for the tested
Figure 4.14. Single_WIPC of different schemes for 42 workloads normalized with respect to ICOUNT for an SMT processor with a divided-ROB structure

T1 and T2 values. This means that a small deviation of T1 and T2 from their optimal values has little effect on the behavior of RASR. This is a desirable property for our algorithm.

4.5.2 Divided-ROB Structure

In this section, we compare the performance and energy results of our approach, Resource Adaptation for Divided-ROB (RADR), to 3 different schemes: ICOUNT [80], Hill-climbing [22] and ARPA, which are applied to an SMT processor with a divided-ROB structure too. In an SMT with a divided-ROB structure, the ROB usage of each thread can not exceed its private ROB capacity while with a shared-ROB structure, each thread can use more than its equally partitioned share.

We use an epoch size of 8K cycles, a STEP size of 2 queue entries (for both ARPA and RADR), and a bank size of 8. There are in all 32 banks since the total number of ROB entries we used in our experiments is 256. We set Threshold1 to 1K and Threshold2 to 3K.
4.5.2.1 Performance Evaluation

Figure 4.14 compares the Single_WIPC of RADR with those of the other schemes across the 42 workloads.

The Single_WIPC improvement over ICount is highest for the RADR and lowest for Hill-climbing. RADR achieves 14.8% and 19.4% Single_WIPC improvement over ICount in 2-thread workloads and 4-thread workloads, respectively, while Hill-climbing only achieves 7.9% and 7.3% Single_WIPC improvement over ICount in 2-thread workloads and 4-thread workloads, respectively. RADR achieves 6.5% and 11.3% Single_WIPC improvement over Hill-climbing in 2-thread workloads and 4-thread workloads, respectively. ARPA achieves an average 4.5% Single_WIPC improvement over Hill-climbing across 42 workloads; The performance improvement of RADR over ICount is even higher than that of ARPA. RADR achieves 17.1% average performance improvement over ICount, while ARPA only achieves 12.4% improvement over ICount across 42 workloads. By considering both relative usage efficiency and real usage efficiency of queue entries, RADR can assess the resource requirements of each thread more accurately than ARPA. RADR outperforms ARPA even when energy-efficiency is not considered.

4.5.2.2 Energy Savings

Figure 4.15 compares the energy consumption of different schemes across the 42 workloads listed in Table 3.3. We normalized the energy consumption of each workload with respect to ICount for all the schemes. The average energy consumption of Hill-climbing is close to that of ICount across 42 workloads. Because of the significant performance improvement over ICount, ARPA achieves 11.9% and 11.7% energy savings over ICount for 2-thread and 4-thread workloads, respectively. Although both ARPA and Hill-climbing do not activate/deactivate processor resources for energy-saving purpose, ARPA has better overall instruction throughput
than Hill-climbing, resulting significant energy savings over Hill-climbing. By controlling the number of powered-on ROB entries, RADR achieves higher energy savings over ICOUNT than that of ARPA. RADR achieves 24.0% and 21.4% energy savings over ICOUNT for 2-thread and 4-thread workloads, respectively. It also achieves 12.4% energy savings over ARPA.

4.5.2.3 Energy-to-Single_WIPC-Ratio Improvements

In this section, we use the metric, Energy-to-Single_WIPC-Ratio (E-WIPC-R) to compare four different schemes. Figure 4.16 shows the E-WIPC-R results of the three schemes for 2-thread and 4-thread workloads. All the results are normalized with respect to ICOUNT. As we can see, the E-WIPC-R improvements of all schemes over ICOUNT are highest for MIX workloads and lowest for ILP workloads.

ARPA achieves higher E-WIPC-R improvement over ICOUNT than Hill-climbing because the energy consumption of ARPA is lower than that of Hill-climbing and the Single_WIPC of ARPA is higher than that of Hill-climbing in most workloads. Compared with Shared-ROB structure as shown in Figure 4.10, both ARPA and Hill-
climbing achieve less $E$-$WIPC-R$ improvements. The reason is that for divided-ROB structure, ICOUNT has less resource clogging than that for divided-ROB structure, thus the overall improvements over ICOUNT is decreased by ameliorating resource clogging.

RASR achieves significant $E$-$WIPC-R$ improvements over ICOUNT, Hill-climbing and ARPA. Compared to ICOUNT, it achieves 32.2% and 33.2% $E$-$WIPC-R$ improvement for 2-thread and 4-thread workloads, respectively. Compared to ARPA, RASR achieves 15.6% and 16.8% $E$-$WIPC-R$ improvement for 2-thread and 4-thread workloads, respectively.

4.6 Chapter Summary

In this chapter, we presented Resource Adaptation schemes targeting both high performance and low energy consumption for SMT processors with shared-ROB and divided-ROB structures. Our resource adaptation schemes analyse not only the relative resource usage efficiency of each thread but also the real dynamic resource usage to adaptively control the number of powered-on ROB entries and partition shared re-
sources among threads. Energy savings are achieved by powering off resources which do not contribute to higher performance and only waste power.

Our experimental results show that for an SMT processor with a shared-ROB structure, our resource adaptation approach achieves 16.7% energy savings over ARPA, while performance loss is negligible across 42 sample workloads. For an SMT processor with a divided-ROB structure, our resource adaptation approach not only saves 12.4% energy but outperforms ARPA by 4.2%.
CHAPTER 5
SUMMARY AND FUTURE RESEARCH

5.1 Conclusion

This dissertation investigates the inefficient resource usage behaviors of various applications in general-purpose processors and studies resource usage controlling techniques targeting both high performance and low energy. We have demonstrated that relatively simple resource usage controlling approaches and methods can lead to considerable energy savings and performance improvements for both superscalar and SMT processors.

5.2 Future Work

Based on the results of this dissertation, we suggest the following related directions for future research.

- *Compiler-based adaptive resource partitioning in SMT processors.* This dissertation has investigated the compiler-based adaptive fetch throttling techniques for superscalar processors. It will be an interesting direction to use compile-time static information to guide multiple-threads resource sharing in SMT processors. At compile time, we estimate the IPC and number of memory accesses of each thread. A thread with low estimated IPC and high number of memory accesses can be allocated few resources since it is expected to clog resources more readily. We can also combine compile-time static information with some runtime information to control resource sharing for both performance and fairness.
• **Resource adaptation for multiple resources in SMT processors.** This dissertation applies resource adaptation only to the ROB structure. We can also apply the same resource adaptation rule to multiple resources simultaneously or develop different resource adaptation rules for different kinds of resources to save energy. The resource adaptation for one resource may impact other resources. It may be interesting to study the effect resource adaptation on different kinds of resources.

• **Efficient resource sharing or thread scheduling schemes on chip multiprocessors with simultaneous multithreading cores (CMT).** We have explored efficient resource controlling techniques for both superscalar processors and SMT processors. It would be interesting to study resource sharing on an increasingly popular architecture, CMT. CMT processors allow threads to run on different cores and share processor resources with different threads. The overall performance of a multithreaded core is sensitive to the set of jobs that are co-scheduled by the operating system job scheduler [75]. If two threads can share resources in a friendly manner, both of the threads can execute smoothly in the pipeline. However, if two threads have many resource sharing conflicts, the performance of both threads will deteriorate.

Conventional multiprocessor scheduling, applied to this architecture, will attempt to balance the thread load across cores. However, this will eliminate the ability to use unbalanced schedules to allocate the right amount of execution resources to each thread [23]. We therefore can consider unbalanced scheduling on CMT processors. Parekh *et al.* [63] use the IPC metric to determine whether threads have aggressive resource contention or not. They observe that the best performance is achieved by scheduling threads with the highest IPC together. However, the IPC measured for a given thread is highly dependent on the other threads running simultaneously. In contrast to the IPC metric, the committed instructions per resource entry (CIPRE) metric used in ARPA, can reflect the
characteristic of threads more accurately. We can apply the thread behavior analysis used in ARPA to determine the threads to be co-scheduled. In addition, we can try other scheduling schemes like resource utilization scheduling for CMT processors.
BIBLIOGRAPHY


[52] Liu, Chen, and Gaudiot, Jean-Luc. “Static Partitioning vs Dynamic Sharing of Resources in Simultaneous MultiThreading Microarchitectures”. In *Proceedings


