

# AUTO SCALABLE AND MORPHABLE MICROPROCESSORS



by  
Nazlı Tokatlı

Submitted to Graduate School of Natural and Applied Sciences  
in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Philosophy in  
Computer Engineering

Yeditepe University  
2021

## AUTO SCALABLE AND MORPHABLE MICROPROCESSORS

APPROVED BY:

Prof. Dr. Gürhan Küçük  
(Thesis Supervisor)  
(Yeditepe University)

.....

Prof. Dr. Haluk Topcuoğlu  
(Marmara University)

.....

Prof. Dr. Sezer Gören Uğurdağ  
(Yeditepe University)

.....

Assist. Prof. Dr. Didem Unat  
(Koç University)

.....

Assist. Prof. Dr. Onur Demir  
(Yeditepe University)

.....

DATE OF APPROVAL: ....../....../2021

I hereby declare that this thesis is my own work and that all information in this thesis has been obtained and presented in accordance with academic rules and ethical conduct. I have fully cited and referenced all material and results as required by these rules and conduct, and this thesis study does not contain any plagiarism. If any material used in the thesis requires copyright, the necessary permissions have been obtained. No material from this thesis has been used for the award of another degree.

I accept all kinds of legal liability that may arise in case contrary to these situations.

Name, Last name Nazlı Tokatlı

Signature .....

## ACKNOWLEDGEMENTS

First, and most of all, I would like to thank my supervisor Prof. Gürhan Küçük, for his expertise, assistance, guidance, and patience throughout the process of this research. Without your help, this research would not have been possible. I would like to express my deep and sincere gratitude to my committee and jury members, Assist. Prof. Didem Unat, Prof. Haluk Topcuoğlu, Assist. Prof. Onur Demir, Prof. Sezer Gören Uğurdağ for their time in preparing and reviewing this dissertation.

I also like to express my appreciation to computer architecture laboratory research team Dr. İsa Ahmet Güney, Muhammed Emin Savaş, Merve Güney, Sercan Sarı, and Berk Kışınbay for their valuable contribution to this dissertation. I would also like to thank all academics and research assistants in Computer Engineering Department at Yeditepe University for their support and team working spirit during this research journey.

I would like to thank my parents Ertam (my hero) and Güler Nakip. There will be always a special place in my life and heart for you. They taught me to stand up for what I believe in and I have seen them go out of their way many times to help others, they give me roots and wings so I can fly, they taught me to reach for the stars and follow my dreams.

Last but not the least, I would like to thank my children Anja and Atilla who are the most precious persons in my life, who are the reason behind my happiness they are my endless treasure. Finally, special thanks to my husband Ali for his endless love and support.

The research work in this dissertation was supported by TÜBİTAK under grant number 117E866.

## **ABSTRACT**

### **AUTO SCALABLE and MORPHABLE MICROPROCESSORS**

This dissertation proposes the design and implementation of a single Out-of-Order superscalar processor capable of dynamic resource sizing and mode switching in response to the properties of running applications. While there are multi-core heterogeneous processor architectures in the literature, our single processor is capable of morphing to target different metrics at different times, such as performance and power. Our final processor proposes a two-parameter run-time switch between Out-of-Order and In-Order execution modes. Additionally, the processor's instruction queue, Re-order buffer, load/store queue, and physical register files are scaled dynamically at runtime to meet the needs of running applications. When the approaches described in this dissertation are used, we demonstrate that we can save an average of more than 42 percent power and achieve a 43 percent increase in efficiency (energy-delay square product) in exchange for a 5 percent performance penalty when compared to a baseline Out-of-Order superscalar.

## ÖZET

### OTOMATİK-ÖLÇEKLENEBİLEN VE ŞEKİL-DEĞİŞTİREBİLEN MİKROİŞLEMCİLER

Bu tez çalışmasında, sırasız mod çalışan süperskalar bir işlemcinin çalışan uygulamaların özelliklerine uyum sağlaması için otomatik olarak özkaynak ölçeklemesi ve çalışma modu değişimi yapması sağlanmaktadır. Çalışmamız, literatürde çok çekirdekli heterojen işlemci mimarilerine rastlanmasına rağmen, tek bir işlemcinin aniden kendini değiştirerek değişik zamanlarda performans ve güç gibi farklı ölçütlere yönelmesini hedefleyen bir çalışma bulunmaması nedeniyle özgündür. Önerdiğimiz sonuç işlemci, sadece iki çalışma-zamanı parametresi ile sırasız çalışma modundan sıralı çalışma moduna geçişe karar vermektedir. Ayrıca, işlemci içindeki komut kuyruğu, yeniden-sıralama belleği, yükleme/saklama kuyruğu ve fiziksel yazmaç dosyalarının boyutları yine çalışma-zamanında çalışan uygulamaların gereksinimleri yönünde ölçeklenmektedir. Bu çalışmada önerilen yöntemlerin uygulanması durumunda sırasız komut çalıştıran baz süperskalar işlemciye göre ortalama yüzde 5 civarında bir performans kaybı karşılığında ortalama yüzde 42'nin üzerinde bir güç tasarrufu ve yüzde 42'nin üzerinde daha iyi verimlilik (enerji-gecikme kare çarpanı) sağladığımızı göstermekteyiz.

## TABLE OF CONTENTS

ACKNOWLEDGEMENTS .....	iv
ABSTRACT.....	v
ÖZET .....	vi
TABLE OF CONTENTS.....	vii
LIST OF FIGURES .....	ix
LIST OF TABLES .....	xiv
LIST OF SYMBOLS/ABBREVIATIONS.....	xv
1. INTRODUCTION .....	1
1.1. MOTIVATION .....	3
1.1.1. Die Area.....	3
1.1.2. Thread Migration.....	3
1.1.3. Power Savings .....	4
1.2. OBJECTIVES .....	5
2. RELATED WORK .....	7
3. APPLICATION MONITORING STAGE IN ASMMP .....	14
3.1. ASMMP RESOURCES' OCCUPANCY PARAMETER .....	14
3.1.1. Occupancy Tests and Results .....	17
3.1.2. Effects of Sampling on Occupancy Values .....	20
3.2 INSTRUCTION DISPATCH RATIO (IDR).....	22
3.3. COMMIT OVER FETCH RATIO (CFR) .....	23
4. APPLICATION CLASSIFICATION/EXECUTION MODE .....	24
4.1. EXECUTION MODE SELECTION .....	24
4.1.1. ASMMP execution mode switching architecture.....	26
4.1.2. Mode Switching Decision Mechanism.....	28
4.1.3. Mode Switching Enforcement Mechanism .....	30
4.1.4. Periodic Operation.....	30

4.1.5. Hardware .....	31
5. APPLICATION CLASSIFICATION/RESOURCE PARTITIONING.....	34
5.1. PARTITIONING DECISION MECHANISM .....	35
5.1.1. Resource downsizing decision .....	36
5.1.2. Resource Upsizing Decision.....	37
5.1.3. The Use of Statistical Parameters .....	38
5.2. PARTITIONING ENFORCEMENT MECHANISM.....	39
5.2.1. Circular Queue Resources .....	40
5.2.2. Instruction Queue (IQ) .....	40
5.2.3. Register Files .....	42
5.3. ASMMP OPERATION.....	45
5.3.1. Issuing Right Processor Configuration.....	45
6. TESTS AND RESULTS.....	49
6.1. EXECUTION MODE SELECTION TESTS AND RESULTS.....	49
6.2. RESOURCE PARTITIONING TESTS AND RESULTS .....	55
6.2.1. Performance.....	56
6.2.2. Issue Queue .....	58
6.2.3. Re-order Buffer .....	60
6.2.4. Load and Store Queues.....	61
6.2.5. Register File.....	64
6.3. FINAL ASMMP TESTS AND RESULTS .....	67
6.3.1. Experimental methodology .....	67
6.3.2. Resource Partitioning Effects .....	69
6.3.3. Performance.....	71
6.3.4. In-Order Runtime .....	74
6.3.5. Power Savings .....	75
6.3.6. Energy-Delay product and Energy-Delay <sup>2</sup> product .....	77
7. CONCLUSION AND FUTURE WORK .....	81
REFERENCES .....	84



## LIST OF FIGURES

Figure 1.1. Basic architectural difference between ARM cores and ASMMP .....	3
Figure 1.2. Design flow of main ASMMP stages .....	5
Figure 3.1. Out-of-Order processor structures redesigned for adaptive processing. ....	14
Figure 3.2. H/W circuitry for collecting occupancy using sampling .....	15
Figure 3.3. Flowchart shows sampling in occupancy calculation. ....	16
Figure 3.4. Re-order buffer average occupancy for 1M cycle period.....	17
Figure 3.5. Load queue average occupancy for 1M cycle period.....	18
Figure 3.6. Store queue average occupancy for 1M cycle period.....	18
Figure 3.7. Integer Physical register file average occupancy for 1M cycle period .....	19
Figure 3.8. Floating point register file average occupancy for 1M cycle period.....	19
Figure 3.9. Issue queue average occupancy for 1M cycle period.....	20
Figure 3.10. Higher value, worse outcome /or deviation rate after applying L1 norm .....	21
Figure 3.11. Higher value, worse outcome /or deviation rate after applying L2 norm .....	21
Figure 4.1. Execution mode selection policy in ASMMP .....	26
Figure 4.2. Composite core architecture .....	27
Figure 4.3. Execution mode selection architecture in ASMMP .....	28

Figure 4.4. Periodic operation of execution mode selection in ASMMP .....	31
Figure 5.1. Resource partitioning in ASMMP .....	35
Figure 5.2. Partition decision mechanism flowchart for ASMMP .....	39
Figure 5.3. Partition enforcement mechanism for resource upsizing in ASMMP.....	43
Figure 5.4. Partition enforcement mechanism for resource downsizing in ASMMP.....	44
Figure 5.5. Flowchart for proposed ASMMP architecture .....	46
Figure 6.1. Runtime percentage of the In-Order mode in mode selection mechanism of ASMMP for various alpha thresholds .....	50
Figure 6.2. Percentage of performance drop in mode selection mechanism of ASMMP for various alpha thresholds.....	53
Figure 6.3. Percentage of power savings in mode selection mechanism of ASMMP for various alpha thresholds.....	54
Figure 6.4. Percentage of EDP savings in mode selection mechanism of ASMMP for various alpha thresholds .....	55
Figure 6.5. Percentage of <i>ED2P</i> savings in mode selection mechanism of ASMMP for various alpha thresholds.....	55
Figure 6.6. Performance of resource partition in ASMMP compared to baseline Out-of-Order processor. ....	57
Figure 6.7. Average performance for simulated benchmarks with different threshold values .....	57
Figure 6.8. Average performance drop percentage for simulated benchmarks with different threshold values .....	58

Figure 6.9. Instruction Queue entries used compared to baseline (64 entry no partition) configuration.....	59
Figure 6.10. Average used instruction queue entries for all simulated benchmarks compared to baseline (64 entry) configuration after resource partitioning .....	60
Figure 6.11. Re-order buffer entries used compared to baseline (192 entry no partition) configuration.....	61
Figure 6.12. Average used Re-order buffer entries for all simulated benchmarks compared to baseline (192 entry) configuration after resource partitioning. ....	61
Figure 6.13. Store queue entries used compared to baseline (32 entry no partition) configuration.....	60
Figure 6.14. Average used store queue entries for all simulated benchmarks compared to baseline (32 entry) configuration after resource partitioning. ....	63
Figure 6.15. Load queue entries used compared to baseline (32 entry no partition) configuration.....	63
Figure 6.16. Average used load queue entries for all simulated benchmarks compared to baseline (32 entry) configuration after resource partitioning. ....	64
Figure 6.17. Integer register file entries used compared to baseline (256 entry no partition) configuration.....	65
Figure 6.18. Average used integer register file entries for all simulated benchmarks compared to baseline (256 entry) configuration after resource partitioning. ....	65
Figure 6.19. Floating-point register file entries used compared to baseline (256 entry no partition) configuration.....	66

Figure 6.20. Average used floating-point register file entries for all simulated benchmarks compared to baseline (256 entry) configuration after resource partitioning. ....	66
Figure 6.21. Percentage of saved entries after applying partition in different ASMMP resources .....	67
Figure 6.22. Percentage of saved entries in final ASMMP for Spec2006 benchmarks during Out-of-Order execution mode.....	70
Figure 6.23. Average saving percentages for different ASMMP resources includes both out-of-order and In-Order execution mode .....	71
Figure 6.24. Percentage of performance drop of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds .....	70
Figure 6.25. Runtime percentage of the In-Order mode in final ASMMP for various alpha thresholds .....	73
Figure 6.26. Percentage of performance drop of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds .....	74
Figure 6.27. Runtime percentage of the In-Order mode in ASMMP for both partition and non-partition versions for various alpha thresholds.....	75
Figure 6.28. Percentage of power savings in final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds. ....	76
Figure 6.29. Percentage of power savings of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds .....	77
Figure 6.30. Percentage of EDP savings of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds.....	78

Figure 6.31. Percentage of EDP savings of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds ..... 79

Figure 6.32. Percentage of ED<sup>2</sup>P savings of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds ..... 79

Figure 6.33. Percentage of ED<sup>2</sup>P savings of ASMMP for partition and non-partition versions compared to baseline Out-of-Order processor for various alpha thresholds ..... 80

Figure 7.1. ASMMP implemented stages ..... 81



## LIST OF TABLES

Table 3.1. Specification of the simulated processor .....	17
Table 4.1. Transistor count for execution mode selection mechanism.....	33
Table 6.1. Specification of the simulated processor during execution mode switching .....	49
Table 6.2. Performance drop of In-Order mode .....	50
Table 6.3. Near-optimal thresholds.....	52
Table 6.4. Specification of the simulated processor during resource partitioning .....	56
Table 6.5. Specification of simulated ASMMP architecture .....	66

## LIST OF SYMBOLS/ABBREVIATIONS

ASMMP	Auto scalable and morphable microprocessor
AMP	Asymmetric multicore processor
B	Portion
CC	Composite core
CPU	Central processing unit
Cct	Circuit
CFR	Commit fetch ratio
D	Average resource occupancy
DP	Decision period
EDP	Energy delay product
ED <sup>2</sup> P	Energy delay square product
ESDP	Execution sampling decision period
EPI	Energy per retired instruction
G	Resource size
GHz	Gigahertz
GB	Gigabyte
H/W	Hardware
IO	In-Order
IQ	Instruction queue
IDR	Instruction dispatch ratio
ILP	Instruction level parallelism
IPC	Instruction per cycle
IPS	Instruction per second
ISA	Instruction set architecture
IDB	Instruction dispatch buffer
iMODE	Interactive mood detection engine
KB	Kilobyte
LQ	Load queue
LSQ	Load store queue
L1	Level 1

L2	Level 2
LAD	Least absolute deviation
LAE	Least absolute errors
MLP	Memory level parallelism
MSHR	Miss status holding register
MAC	Multiplier accumulator
OoO	Out-of-order
OSP	occupancy sampling period
PS	Power savings
PE	Power expenditure
PDP	Partition decision period
PRF	Physical register file
PC	Program counter
RAT	Register alias table
RFI	Integer register file
RFF	Floating point register file
ROB	Re-order buffer
S	Speedup ratio
SR	Slowdown rate
SP	Sampling period
SRE	Saved resource entries
SQ	Store queue
SMT	Simultaneous multi thread
SRAM	Static random access memory
T	Threshold
TLP	Thread level parallelism
W	In-order runtime percentage
$\alpha$	Alpha threshold



## 1. INTRODUCTION

Computers (processors) are responsible for running various types of applications. For a long time, speed was the only major concern of processors when applications are run. Nowadays, power-related concerns even surpass all performance-related concerns on any processor encountered on any type of system.

A power-aware processor plays an important role in improving the lives of many modern technologies (e.g., smartphones, laptops). It is somewhat less obvious that a power-aware processor also plays an important role in the success of a real-time, mission-critical system that explores space and other planets. Besides, it is predicted that data centers of the world will consume one-fifth of Earth's power by 2025, and a power-aware processor may provide enormous power savings when deployed on data centers in large quantities [32].

The well-accepted philosophy on today's processor design is still a "one-size-fits-all" kind of realization with a fixed set of data path structures and a fixed mode of execution. For instance, when an application has a program phase with a low Instruction Level Parallelism (ILP) degree runs on a 4-way superscalar, aggressively speculative, Out-of-Order processor, most processor resources are underutilized, and all power-saving opportunities are lost. On the contrary, when an application has a program phase with a high ILP degree that runs on a 2-way superscalar, In-Order processor, it receives a huge performance penalty due to insufficient processor resources. Both of these scenarios point out the inefficiency of the current fixed-mode processors.

Designing a power-aware processor with good performance depends on periodical monitoring and effectively analyzes the application's behavior. The application needs particular hardware resources such as caches, issue queues, and instruction fetch logic within a dynamic superscalar processor which can vary significantly from application to application and even within the different phases of a given application. The proposal of an adaptive processing approach to improving microprocessor energy efficiency dynamically resizes major microprocessor resources such as caches and hardware queues during execution to better match varying application needs.

This resizing operation usually involves reducing the size of a resource when its full capabilities are not needed, then restoring the disabled portions when they are needed again [5].

The next and natural step to reflect applications' diversity during their running phases is introducing an asymmetric multicore processor design, which uses cores of different types in the same processor and thus, embraces heterogeneity as the first principle. Different cores in an AMP may be optimized for power/performance or different application domains or for exploiting instruction-level parallelism or memory-level parallelism. Thus, AMPs promise to be beneficial for a broad range of usage scenarios [1].

The concept of heterogeneous cores within a processor is not novel. One of the best examples of such realizations is ARM's power-aware (big. LITTLE) architecture, which requires a physical area to keep multiple cores on the same chip. Furthermore, a 3-wide Out-of-Order Cortex-A15 superscalar microprocessor (large) and a 2-wide In-Order Cortex-A7 superscalar microprocessor (small) are located in the same core in this architecture. This is also a valid approach in almost all heterogeneous architectures proposed in the literature, and as a result, significant area unavoidably increases [8] [24] [25] [15] [30].

Our proposed Auto scalable and morphable processor design deviates from this traditional approach. We propose a specialized Out-of-Order core that can act either as a traditional Out-of-Order core with adaptive processing included or an In-Order core whenever it is suitable to reduce power consumption and keep performance degradation of running application below 5 percent. As a result, the area requirement of our proposed processor is almost identical to the area requirement of a traditional Out-of-Order processor. Figure 1.1 shows the basic architectural difference between both designs.

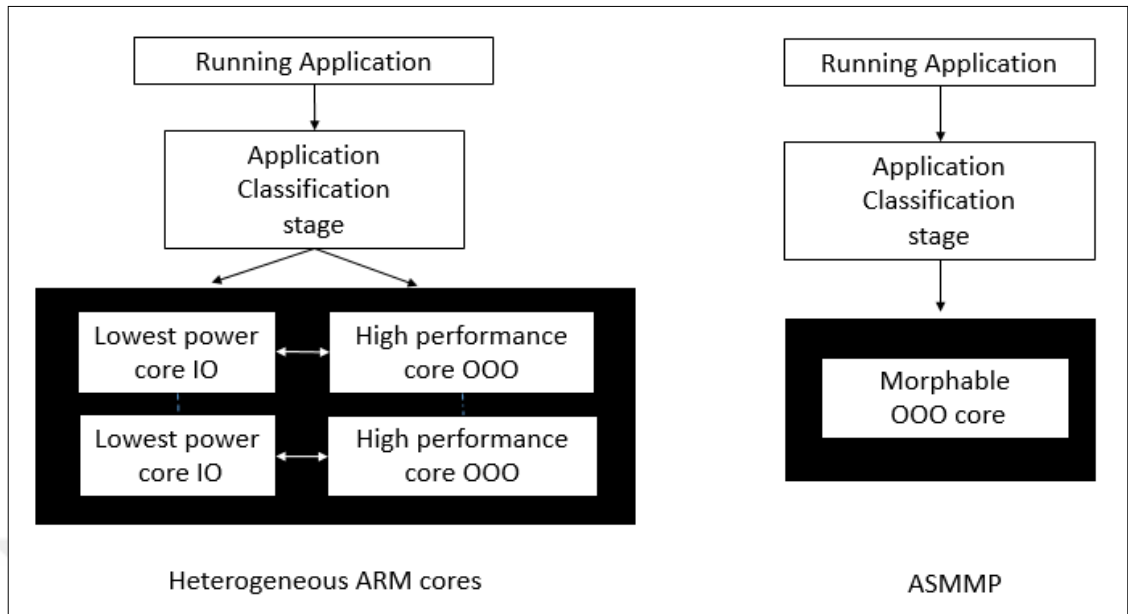


Figure 1.1. Basic architectural difference between ARM cores and ASMMP

## 1.1. MOTIVATION

As we stated above, one of the important points that motivate us to propose ASMMP design, its ability to reduce power consumption while keeping performance degradation of the running application below 5 percent. Also, achieving this goal without using complex and power-hungry circuit. The following subsections will examine the main advantages of ASMMP design when compared to heterogeneous architectures in terms of die area, thread migration and power savings respectively.

### 1.1.1. Die Area

The die area is the most important aspect of our proposed design, as illustrated in Figure 1.1. This is one of the most powerful areas of ASMMP design. Processors with heterogeneous cores house multiple cores on the same chip. There is apparently no issue when multiple active threads are assigned to each of these heterogeneous cores. However, when only one core is active, and the rest of the cores are passive at certain times [8], the processor's area efficiency is immediately questioned. ASMMP is an Out-of-Order processor (single core) that can also function as an In-Order processor. As a result, its area requirement is nearly identical to that of a traditional Out-of-Order processor. Also, the number of transistors that

required to implement the ASMMP will certainly less than the number of transistor count that needed to implement dual core processor (291 Million transistor/143 mm<sup>2</sup>) and then less die area required for ASMMP when compared to dual core processor [33]. Finally, our decision circuit for the execution mode switch does not necessarily involve complex methods such as machine learning, nor does it necessitate the storage of additional statistics tables in the hardware [8] [9].

### **1.1.2. Thread Migration**

Thread migration is an important feature of heterogeneous cores. These processors enable a running thread to migrate to a suitable core to save power or maintain an application's performance at its peak. On the other hand, Thread migration takes a long time, especially when the source and target cores are located away. The state of the source core should be transferred to the target core, and larger state data means longer migration times and higher power costs.

Data transfer is one of the most difficult challenges in computer architecture research, and it should be avoided as much as possible. In Pentium 4, Intel switched from separate physical and architectural register structures in Pentium III to a combined register file that holds both physical and architectural registers in one structure. We use the same strategy in ASMMP design, and we make both the source and target cores the same core. As a result, all data path structures holding instructions and processor state can retain their content during a thread migration, and there is no migration cost in terms of latency and power.

### **1.1.3. Power Savings**

ASMMP can remain in IO execution mode for an extended time. Structures used to ensure correct Out-of-Order execution (such as the register renaming mechanism, load queue, Re-order buffer, and physical register files) can be disabled while in In-Order mode. As long as the In-Order execution mode is enabled, ASMMP is a very low-power processor. We assumed that the power dissipated by the In-Order core is three to five times as much as the large Out-of-Order core [17]. Furthermore, when the running application requests an Out-

of-Order mode, the processor enters a performance mode, in which ASMMP continues to save less power than In-Order execution mode due to its ability to configure its resources dynamically.

## 1.2. OBJECTIVES

This dissertation is part of the TÜBİTAK project titled "Auto scalable and morphable microprocessor)" and the research work related to designing a single processor morphing itself to target various metrics, such as performance and power different times, even though there are multicore heterogeneous processor architectures in the literature. Figure 1.2 shows the design flow of ASMMP. As shown in the Figure, ASMMP contains three main stages, respectively. The flow starts with monitoring application behavior for a specific time interval called an epoch. The monitoring process relies on a periodical collection of statistics about the used processor resources and exploration of instruction-level parallelism of the running application. Then, the collected information will be used to classify applications into three main types. In the last stage, based on the application type, a suitable processor configuration will be chosen for running the application to save power without sacrificing too much performance.

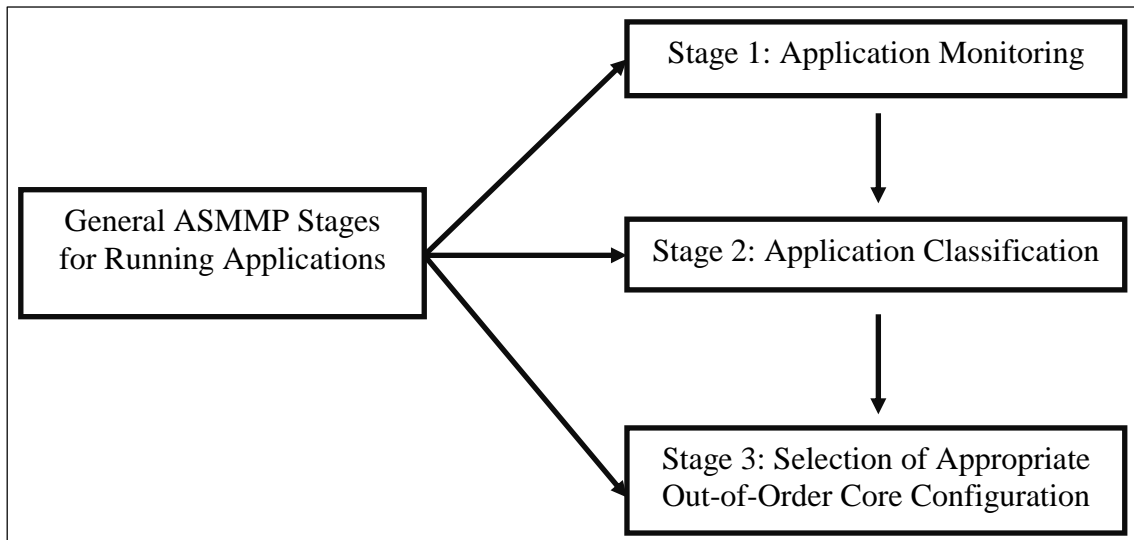
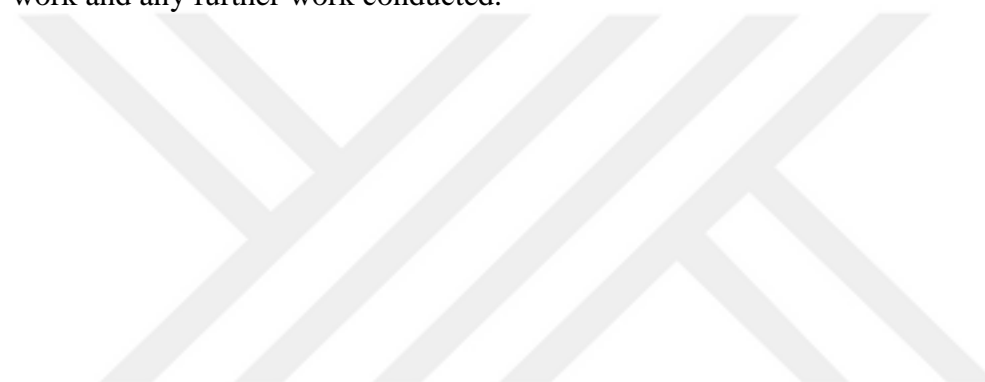


Figure 1.2. Design flow of main ASMMP stages

The remainder of the thesis is as follows: Chapter 2 summarizes the previous research in this field. Chapter 3 describes the application monitor statistics, followed by stage one of ASMMP's design and implementation details. Chapter 4 represents the details of ASMMP's dynamic execution mode selection (application classification stage) design and implementation. Chapter 5 represents the details of ASMMP's adaptive resource partitioning (application classification stage) design and implementation. Additionally, expressing ASMMP operation describes selecting an ASMMP configuration that is appropriate for the running application. Chapter 6 discusses the simulation results and their implications. Chapter 7 concludes by summarizing the research and discussing the significance of this work and any further work conducted.



## 2. RELATED WORK

Numerous studies on energy, power, and die area efficiency have been published in the literature. One of the most widely used techniques for reducing processor energy consumption is to dynamically turn on and off processor resources in response to the needs and behavior of the running workload [4] [5] [6] [7] [12] [13]. Another area of research is related to architectural techniques that employ simpler data path structures that consume less power and space while providing comparable performance to Out-of-Order processors [7] [9] [13] [16] [17] [18] [19] [20] [21] [22].

To give more details about adaptive processing and the use of simple data path structures to reduce power consumption. We will start with the work done by Manne et al. (1998) that discusses pipeline gating, a technique for lowering the processor's average activity [6]. They attempt to determine whether a branch is likely to mispredict and prevent wrong-path instructions from entering the pipeline by evaluating the quality of each branch prediction using a confidence estimator logic. Additionally, Ghiasi and his team (2000) looked for ways to save power by switching from an Out-of-Order instruction issue mechanism to In-Order or dynamically-gated (reduced width) modes [12]. A micro-architectural IPC matching mechanism combined with an external performance indicator to determine when the instruction issue mechanism should be changed. The operating system specifies a target IPC rate for the processor to achieve, and the processor uses various techniques to approximate the current IPC rate while executing committed instructions.

Bahar and Manne (2001) proposed a technique for balancing pipelines by splitting the issue width between two clusters and saving power by monitoring the program's issue needs (look at the Issue IPC history) of the running application [4]. A low-power mode is selected when the program does not require the processor's full issue capabilities by reducing the instruction issue width from 8 to 4 instructions.

As another option, Lebeck et al. (2002) propose classifying instructions that require a long-latency operation to reduce power consumption [19]. These instructions are moved out of the (relatively small) scheduling window into a (relatively large) waiting instruction buffer

(WIB) until the operation is complete, at which point they are returned to the scheduling window.

This combines the benefit of a large instruction window with the benefit of a small scheduling window in terms of latency tolerance. However, it requires a large instruction window (as well as a large physical register file), which comes at a cost.

Also, when there is a long-latency instruction as an L2 miss, the issue queue may be filled with instructions dependent on the L2 miss. As a result, the issue queue will not expose instruction-level parallelism until the miss is resolved. In the context of memory-latency tolerant processors, Morancho et al. (2007) propose delaying the insertion of instructions dependent on load instructions that are predicted to miss L2 into the issue queue [13]. Instead of being inserted into the issue queue, these instructions will be stored in an instruction buffer. The dependent instructions will be inserted into the issue queue after the L2 miss has been resolved.

Albonesi and his team (2003) investigate the problem of configuring a processor with two levels of caches, integer, and floating-point issue queues, load/store queues, register files, and a Re-order buffer [5]. To avoid the overwhelming number of possible configurations, they tune each component solely on its local usage statistics. They propose two heuristics for accomplishing this, one for tuning caches and the other for tuning buffers and register files. The caches they consider are selective-way ones, meaning that each of their multiple ways can be activated or deactivated independently. Every way has the most recently used (MRU) counter, which is incremented whenever a cache search hits the way. The cache tuning heuristic samples this counter at fixed intervals to determine the number of hits in each way and the total number of misses. The heuristic computes the energy and performance overheads for all possible cache configurations using these access statistics and dynamically selects the best configuration. The occupancy-based heuristic is used to control other structures such as issue queues. It determines whether to upsize or downsize the structure based on how frequently different components of these structures are filled up with instructions.



Huber and his colleagues recommend (2011) adjusting processor pipeline depth during runtime [7]. They added two new special instructions, MERGE and BREAK, to the instruction set to achieve this new level of adaptivity. The MERGE instruction merges two adjacent pipeline stages by making the pipeline register transparent, which means that the clock is ignored and the data path is passed directly to the next pipeline stage.

In most cases, merging two adjacent pipeline stages can save one cycle. The BREAK instruction splits two merged pipeline stages in half by reapplying the clock to the pipeline register. This method allows personalizing the processor pipeline to the specific application during runtime.

Carlson team (2015) tried to improve the performance of MLP-sensitive applications on In-Order processors [20]. Memory instructions and address computations are identified by the load slice core, placing them in a separate bypass queue (urgent and non-urgent instructions). As a result, loads can be executed ahead of time.

Another proposal by Sembrant et al. (2015) to reduce power consumption is to relieve pressure on Out-of-Order structures by parking instructions that are predicted to be non-critical for memory-level parallelism before renaming, thereby relieving pressure on the issue queue and PRF temporarily until those instructions are resumed [17].

Additionally, leading research attempts to select an appropriate core in multicore systems based on the requirements of running applications to reduce power consumption. Typically, a multicore processor is made of cores of the same type (homogeneous (symmetric) cores) or cores of different types (heterogeneous (asymmetric) cores). There are numerous multicore processor architectures with varying core counts and core types. Also, the running application may be assigned to a large core with high performance or to a small core with low energy consumption [8] [12] [14] [15] [16]. The majority of recent studies in the literature advocate heterogeneous cores in multicore systems [4] [8] [23] [24] [25]. Another approach also allows for heterogeneity in terms of core types and dynamic resource sharing between cores [14] [26]. Additionally, researchers have fused adjacent cores to create large, Out-of-Order cores in some cases, a process known as core-fusing [27] [28].

Kumar et al. (2003,2004) attempted to reduce power consumption in multicore architectures by dynamically estimating a program's resource requirements and mapping them to the most appropriate core [24]. A wide-issue superscalar processor, for example, can issue multiple instructions in each cycle and is thus best suited for a program with a high ILP. On this processor, mapping a program with a low ILP wastes resources. Their method allows them to optimize for different purposes, including performance and energy efficiency. To make thread scheduling decisions, the performance of a thread on different core types must be known by running threads on different core types to sample their performance.

Annavaram et al. (2005) proposed a technique to improve both sequential and parallel performance by adjusting the amount of energy consumed in processing each instruction based on the degree of parallelism available [25]. For a fixed power budget, a processor should spend more energy per retired instruction in phases of limited parallelism (low retired instructions per second) and vice versa, according to the power equation represented by multiplying EPI by IPS on all cores. To maintain a power budget constraint, less energy should be consumed in processing each instruction for phases with high parallelism and vice versa. On this basis, they map high IPS parallel phases to cores with low EPI and low IPS sequential phases to cores with high EPI. As opposed to an SMP, an AMP provides a more suitable platform with cores of different EPI and thus within the same power budget.

An architecture presented by Kim et al. (2007) is reconfigurable so that it allows simple cores to be dynamically combined into larger cores to optimize either performance, energy, or area efficiency [27]. They completely avoid physical sharing of resources to allow the processor to scale up to a large issue width (e.g., 64 wide), which precludes the use of traditional reduced/complex instruction set computing (RISC/CISC) ISAs. As a result, they employ nonstandard explicit data graph execution (EDGE) ISA, in which the order of dependence of instructions within a block is explicitly and statically encoded, and thus instruction dependence relations are known a priori. This allows the cores' I-cache capacity and instruction fetch bandwidth to scale linearly as the number of cores grows. As a result, their technique's scalable reconfiguration shows higher flexibility and performance than techniques that centralize some processor resources.

The core fusion architecture presented by Ipek et al. (2007) allows the formation of larger-issue Out-of-Order cores by fusing neighboring Out-of-Order cores, and it is described in detail in the paper [28]. As a result of their architecture, up to four cores, each with its own D/I cache, can be merged at runtime to form cores four times as much as the D/I cache size, four times as much as the branch target buffer size, and four times as much as the branch predictor size, and four times as much as the commit, issue, and fetch width. Its reconfigurable load-store queue (LSQ) and D-cache organization allow for conventional coherence while running a parallel program and does not generate any coherence traffic when running a serial program in fused mode. Cores execute independently of one another, and LSQs and D-caches of individual cores are used to avoid thread interference in L1 caches when this is the case.

In this design, a decentralized frontend and I-cache are used to feed the fused backend, rather than to require additional resources on different frontends on the same processor core. Core fusion can provide more powerful four-issue super cores to support coarse-grain parallelism, more powerful four-issue super cores to support fine-grain parallelism, and one eight-issue super core to support sequential execution, among other things. The performance of the fused configuration in the core fusion architecture is significantly lower than that of an iso-area monolithic Out-of-Order processor in the core fusion architecture.

Lukefahr and his team (2012,2016) proposed the Composite Core, a reconfigurable core with two execution engines ( $\mu$ engines). Both engines have a unique microarchitecture [8] [9]. The so-called "Big engine" has a Out-of-Order pipeline, whereas the "Small engine" has a pipeline that is in order. Both engines share a few resources, including the L1 cache, the fetch stage, and the branch predictor. At any given time, only one engine is active, resulting in an In-Order or Out-of-Order core. Multiple performance metrics are monitored by the Composite Core to determine when modes are switched. These metrics are collected from running engines and forecasted in the sleeping engine. Then, the switching algorithm determines which of the two engines are run during that particular program phase. An offline training stage is used to establish the optimal coefficient for each metric.

Asymmetry in a multicore processor can also be introduced by dynamically adjusting the resources available to a core in response to its workload. Homayoun et al. (2012)

investigated how microarchitectural structures can be shared across three-dimensional stacked cores [14]. They point out that the 2D design of reconfigurable AMPs results in inefficient resource pooling, which increases pipeline depth and communication delays. They propose a 3D design of reconfigurable AMPs that allows for optimized pipeline layout in a 2D plane. In addition, additional resources (e.g., LSQ, ROB, cache, registers, and instruction queue) are linked in the third dimension without disrupting the 2D pipeline layout. This enables fine-grained resource sharing to exploit both ILP and TLP while achieving shorter communication delays due to 3D stacking.

Khubaib et al. (2012) proposed another technique for power reduction called morphcore, which utilizes shared hardware resources to operate as an Out-of-Order core or as multiple In-Order SMT cores with unused resources such as the Re-order buffer and Out-of-Order wakeup/selection logic turned off [16].

To determine which core mode to use, the number of currently scheduled threads is multiplied by two. When the number of threads reaches a predetermined threshold, the SMT mode is activated to reduce the number of threads. If a group of high ILP threads is scheduled together or with many low ILP threads, this approach may cause the high ILP threads to suffer. If a small number of low ILP threads are scheduled together and run in the Out-of-Order mode, this can result in an unnecessary increase in overall power consumption. During mode switching, the instruction cache, data cache, and branch history do not need to be flushed because morphcore does not require them. The only switching overhead introduced by morphcore is the process of draining the pipeline.

Afram et al. also recommend the FlexCore architecture as a viable and workable alternative. In a multi-cluster architecture, the FlexCore architecture consists of two small cores [15]. These cores can transform into a wide and Out-of-Order processor, if necessary. The Out-of-Order processor can also be set up to function as a simultaneous multi-threading (SMT) core. However, sufficient thread-level parallelism (TLP) between the existing threads is required to achieve this configuration. Various run time statistics are collected like our proposed method to make an accurate estimate of the proper configuration. Cores can also be configured to run in a low power In-Order execution mode.

Finally, we proposed the interactive Mood Detection Engine as an early research project. (iMODE) is a similar processor that includes trial periods for both In-Order and Out-of-Order execution modes to determine the most appropriate execution mode [29]. There was no prediction logic in that study, and we relied entirely on the results of the trial execution modes, which averaged around 17 percent power savings. Due to the presence of trial execution modes in iMODE, performance is degraded unnecessarily, or valuable power-saving opportunities are missed. As a result, when designing our ASMMP architecture, we decided to incorporate an accurate mode prediction logic and eliminate those trial execution modes periods. The ShapeShifter architecture used in this thesis for mode switching mechanism is proposed, an Out-of-Order superscalar processor that sometimes acts as if it is a smaller In-Order superscalar processor [31]. At certain decision points, the mode switching circuit decides which execution mode is more suitable for running an application until the next decision point. The ShapeShifter achieves better power and energy-delay savings with just two processor statistics than the relatively complex and power-hungry control circuit within a composite core architecture previously proposed in the literature.

### 3. APPLICATION MONITORING STAGE IN ASMMP

This chapter is organized as follows: Section 3.1. explains the ASMMP resources' occupancy parameter used to collect information about the running application needs and its behavior periodically. Section 3.2 shows how we predict instruction-level parallelism for the running application, followed by Section 3.3, which explains how we measure the success rate of branch prediction in a speculative processor with a dynamic branch prediction mechanism. Lastly, all collected information will be used as input data to the next stage of ASMMP, which is the application classification stage.

#### 3.1. ASMMP RESOURCES' OCCUPANCY PARAMETER

Monitoring and collecting important Out-of-Order processor structures occupancy will help decide the approximate needs of processor resources that can be dedicated to the running application to maximize energy savings while limiting any performance loss below a specified level, which is 5 percent in our case. The shaded sections in Figure 3.1 shows Out-of-Order processor structures that can be redesigned for adaptive processing [5].

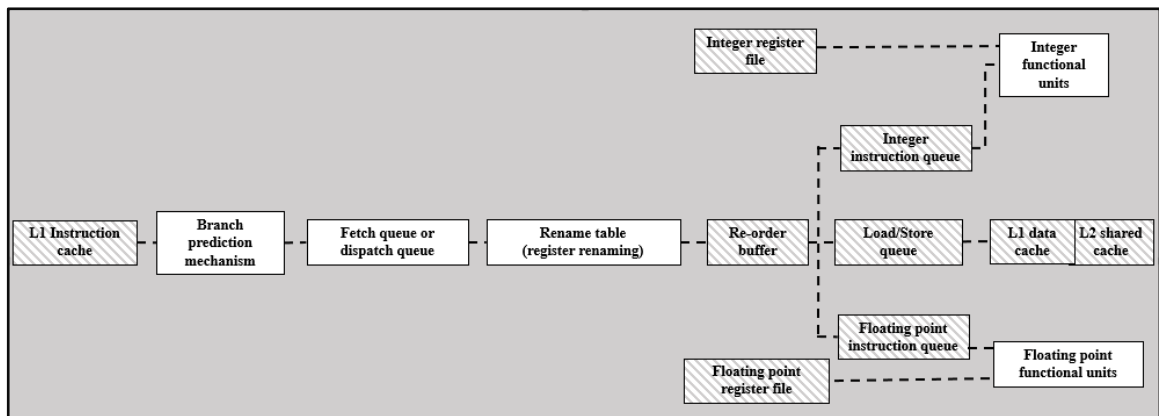


Figure 3.1. Out-of-Order processor structures redesigned for adaptive processing.

To gather resource occupancy in hardware each cycle, we need to check each structure's number of filled entries. This can be easily done by checking each resource entry valid bit. Also, a counter is needed to increment when the resource entry is not empty. Performing

these calculations at every application execution cycle for all resources may lead to huge hardware complexity. Thus, we decide to use the sampling method.

The sampling method is similar to capturing a picture of the resource occupancy not for every cycle but specific running periods. The sampling method helps to reduce monitoring overhead with near-accurate occupancy values of different resources. Also, to apply the sampling technique, we need to add an extra structure (queue) near each resource to periodically collect application occupancy data of different resources. Figure 3.2 shows the H/W circuitry of collecting resources occupancy using the sampling technique. Figure 3.3 shows a flowchart of how to use sampling in occupancy calculation.

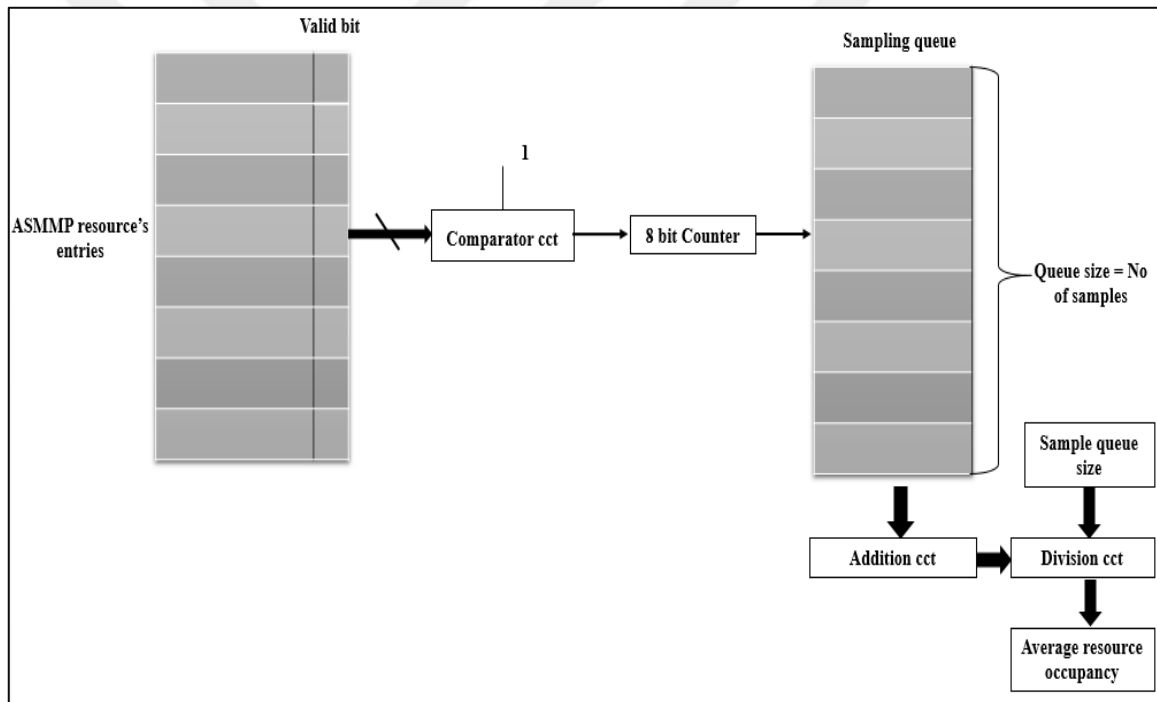


Figure 3.2. H/W circuitry for collecting occupancy using sampling

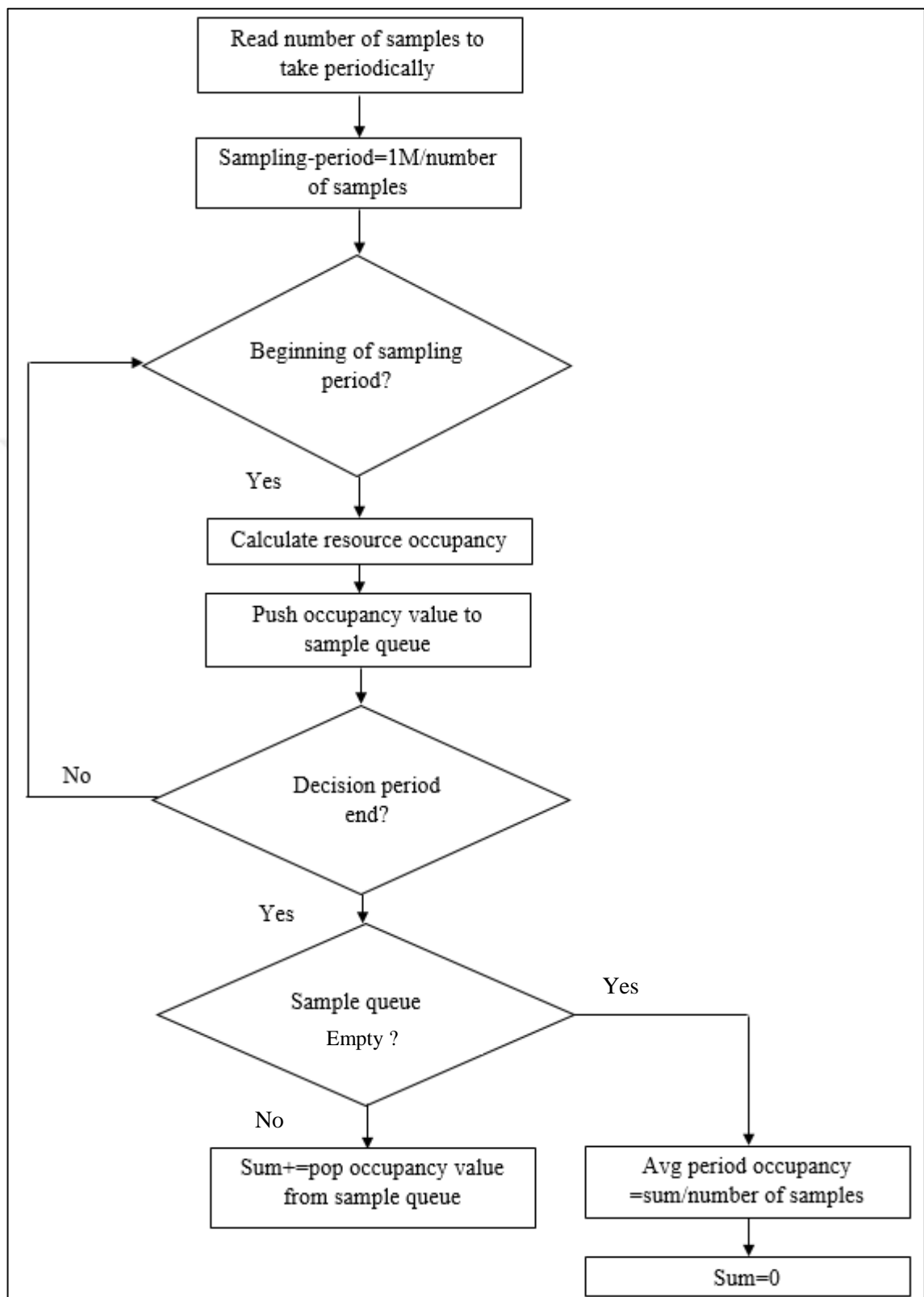


Figure 3.3. Flowchart shows sampling in occupancy calculation.



### 3.1.1. Occupancy Tests and Results

We collect occupancy values for different processor structures by running benchmarks from the Spec2006 benchmark suite using the Gem5 simulator with x86 ISA [2] [3]. Each benchmark is executed for 100M instructions. Configuration parameters for the simulated processor are given in Table 3.1. Lastly, Figures 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9, respectively, show occupancy of different processor resources with different numbers of samples taken periodically at the end of each epoch.

Table 3.1. Specification of the simulated processor

L1 I- and D-Caches	16Kb, 4-way, 64-byte line size, 2 cycle latency
L2 Cache	128Kb, 8-way, 64-byte line size, 20 cycle latency
CPU Frequency	2 GHz
Pipeline	4-way issue bandwidth ROB: 192 entry IQ: 64 entry LSQ: 32 entry RF: 256 registers
Decision period (1 epoch)	1 M cycles
No of samples collected every 1M cycles	1M, 256, 64 samples
No of executed instructions	100 M

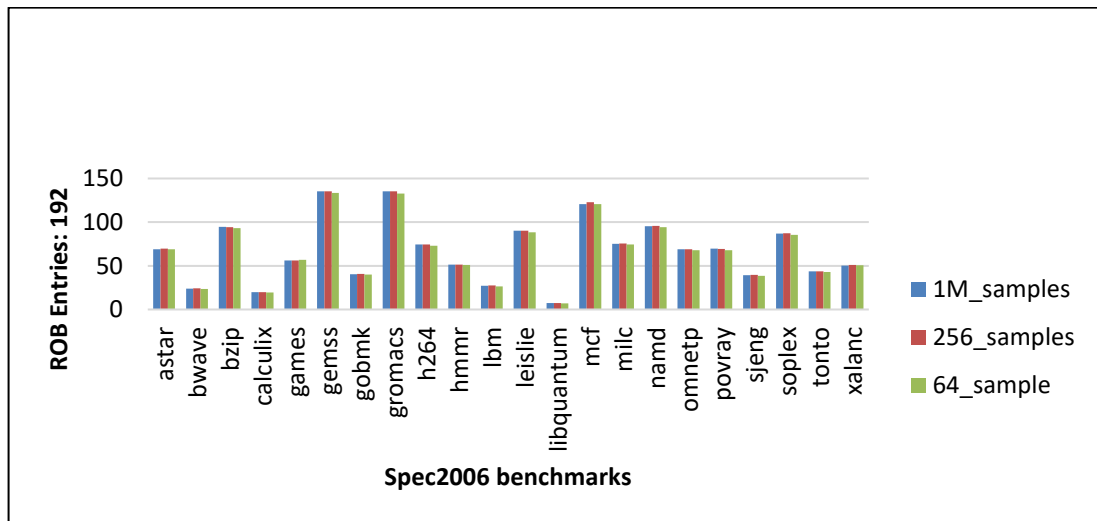


Figure 3.4. Re-order buffer average occupancy for 1M cycle period

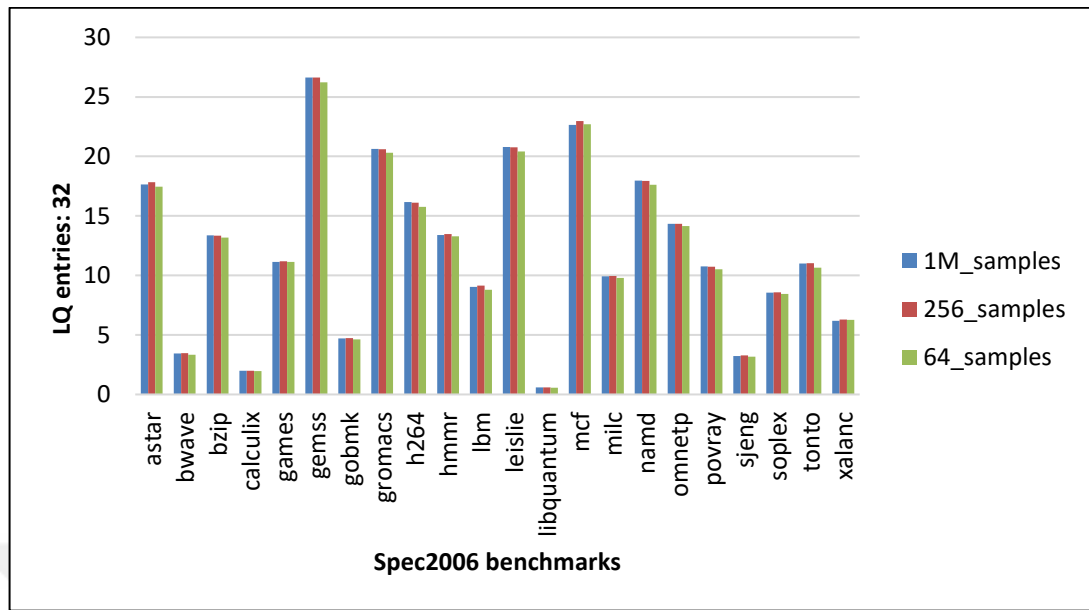


Figure 3.5. Load queue average occupancy for 1M cycle period

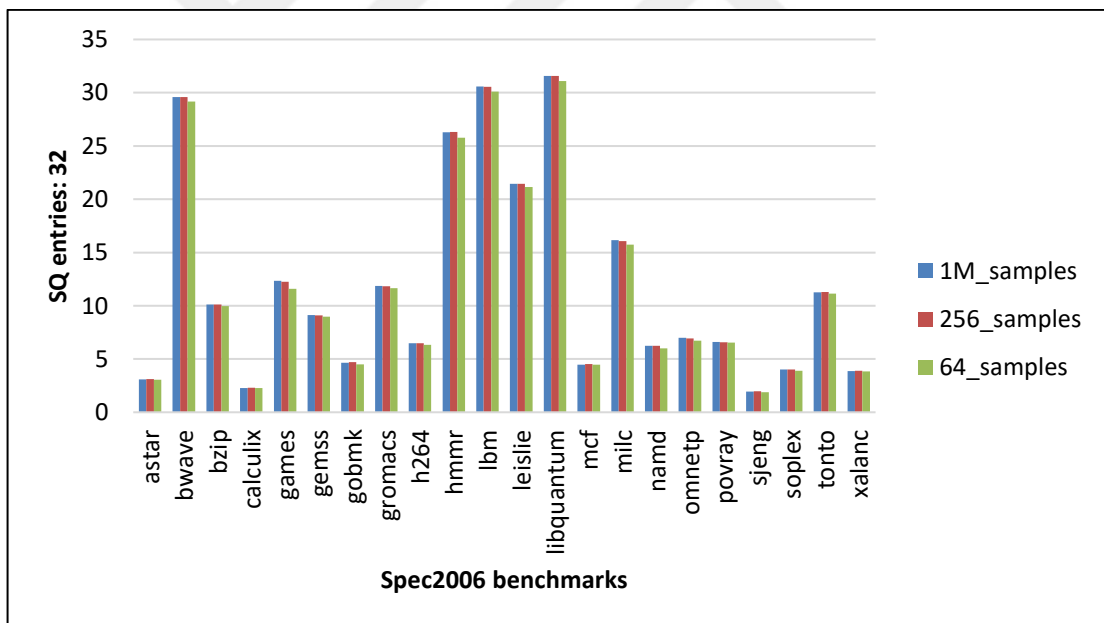


Figure 3.6. Store queue average occupancy for 1M cycle period

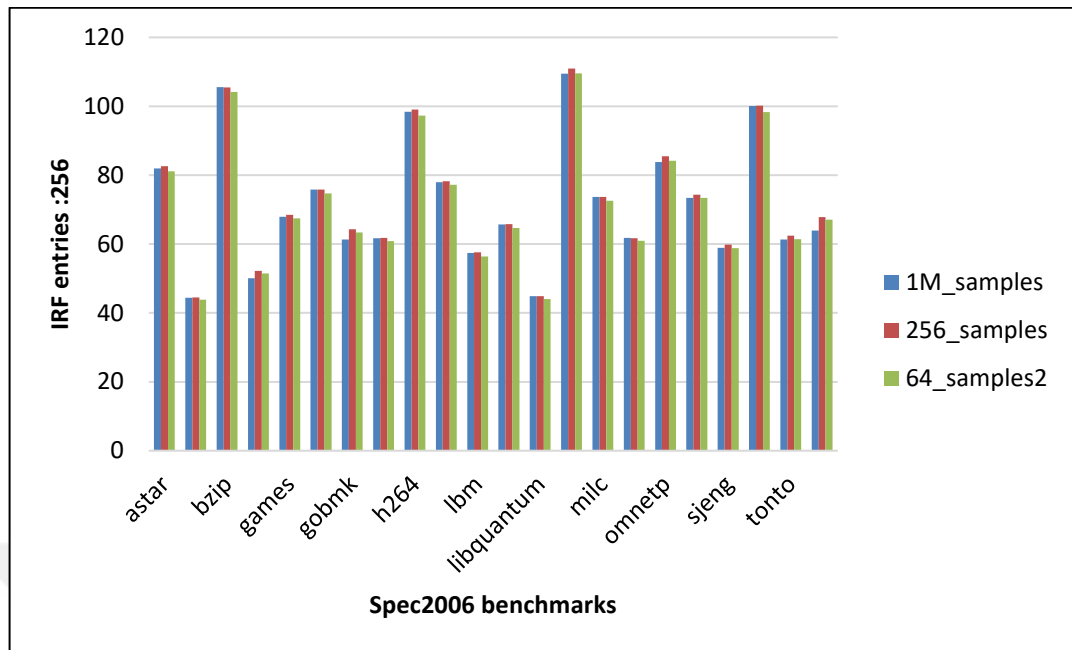


Figure 3.7. Integer Physical register file average occupancy for 1M cycle period

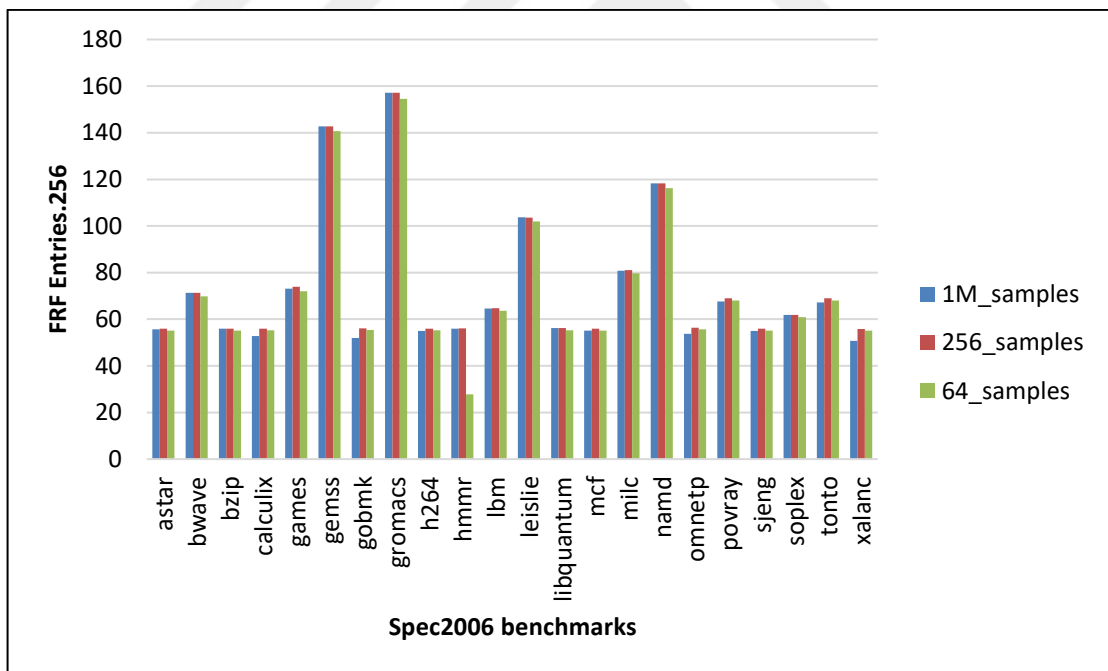


Figure 3.8. Floating point register file average occupancy for 1M cycle period

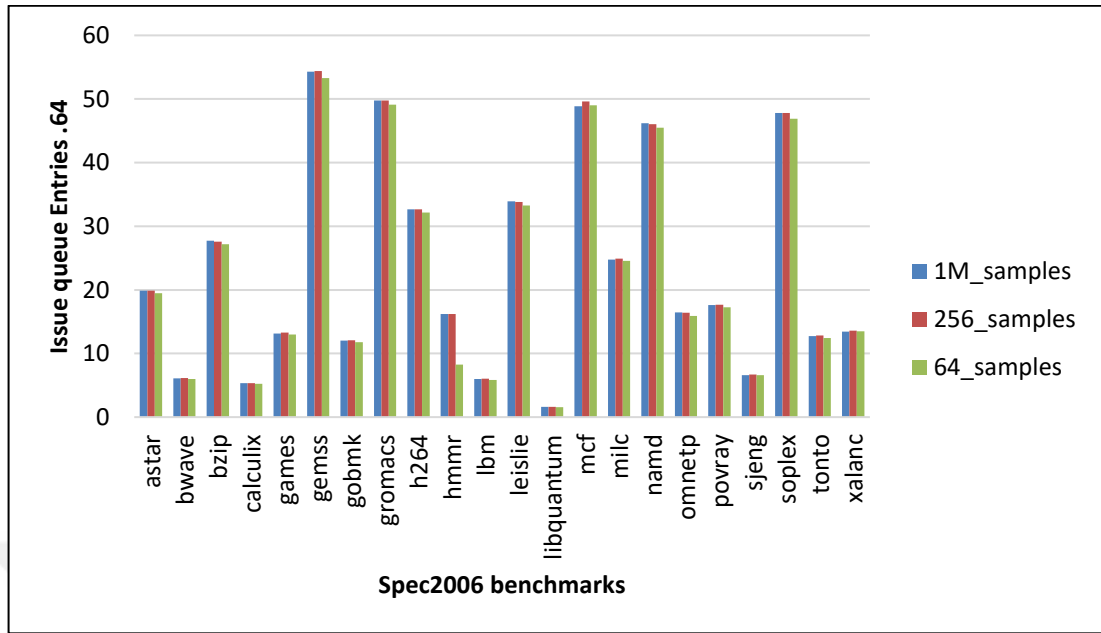


Figure 3.9. Issue queue average occupancy for 1M cycle period

### 3.1.2. Effects of Sampling on Occupancy Values

The average occupancy of different processor structures obtained from the sampling technique showed deviation from actual occupancy values. The actual average occupancy represents the occupancy of different processor structures collected every cycle during one epoch. Also, The changes in deviation values directly depend on the number of samples taken.

Moreover, the deviation between actual occupancy means and sampled occupancy mean should not exceed a certain rate. We collected occupancy data for 16, 32, 64, 128, 256, and 1 million samples to determine the error rate between actual and sampled occupancy values. We then applied the L1 norm (Least Absolute Deviations or LAD) and L2 norm (Least Absolute Errors or LAE) formula as shown in Equation 3.1 and Equation 3.2.

The results show that increasing the number of taken occupancy samples leads to more accurate and closer results to actual occupancy values for Spec2006 benchmarks. Figures 3.10 & 3.11, respectively, show the deviation of occupancy samples from actual occupancy values, which is equal to one million samples in our case.

$$L1 \text{ norm} = \sum \frac{\text{abs}(\text{actual value before sampling} - \text{new value after sampling})}{\text{Resource size or no of resource entries}} \quad (3.1)$$

$$L2 \text{ norm} = \sqrt{\sum (\text{actual value before sampling} - \text{new value after sampling})^2} \quad (3.2)$$

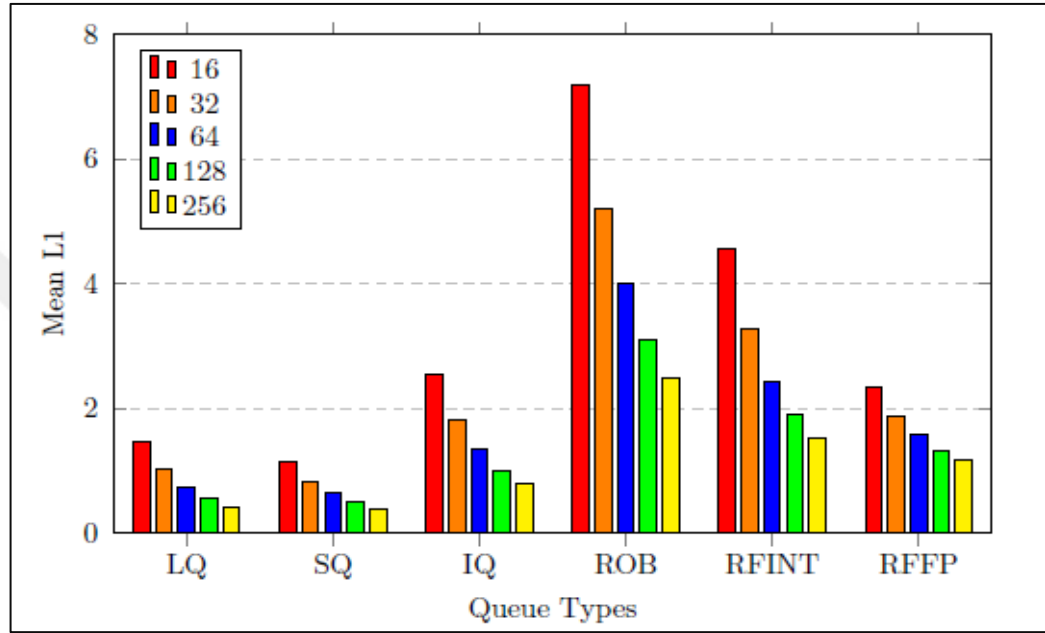


Figure 3.10. Higher value, worse outcome /or deviation rate after applying L1 norm

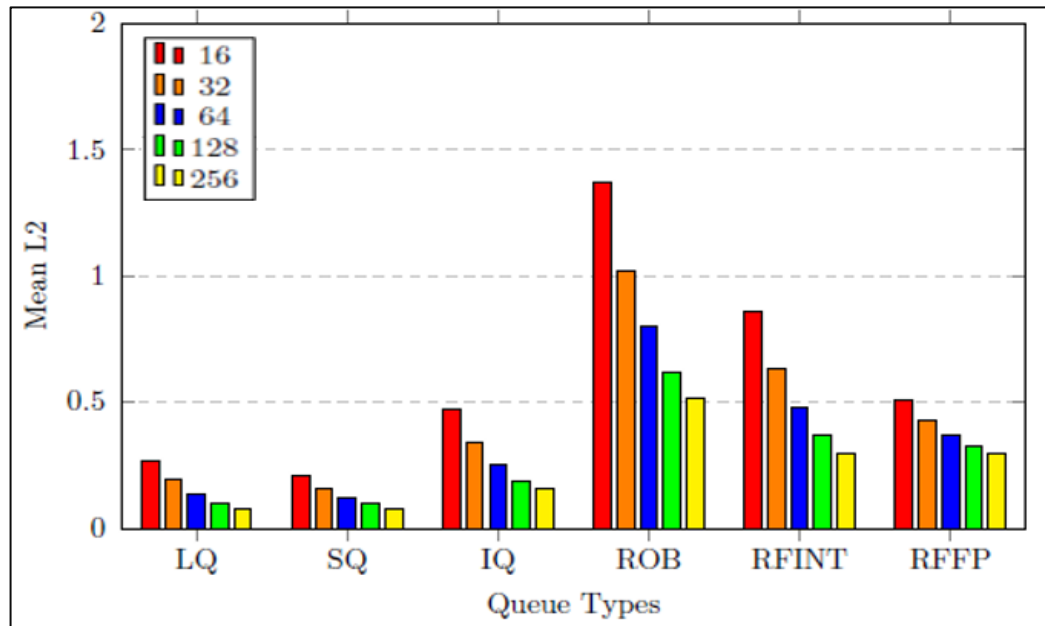


Figure 3.11. Higher value, worse outcome /or deviation rate after applying L2 norm

### 3.2 INSTRUCTION DISPATCH RATIO (IDR)

The main difference between Out-of-Order and In-Order execution modes is that Out-of-Order processors can schedule any of the ready instructions waiting in the IQ regardless of the original program order, whereas In-Order processors must schedule them in strict program order. For example, if any of the customers entering a market finishes their shopping, they can complete their transaction at the cash register regardless of the order they entered. Out-of-Order processors have logic that is identical to this. When the cash register begins dealing with customers in the order they enter the market, the shopping process can be streamlined. However, we may need to delay some customers who finished their shopping earlier than those who entered the market earlier. This second scenario is perfectly matched by the logic of In-Order execution mode. We can collect the IDR value with a counter used to measure the ILP level for running the application.

Therefore, dividing the number of instructions currently awaiting Out-of-Order execution by those awaiting In-Order execution allows us to determine how faster an Out-of-Order processor could be compared to an In-Order execution. We can obtain a sufficiently precise IDR value by sampling enough ( $n$  times) over some time, as shown in Equation 3.3 below.

$$IDR = \frac{\sum_{i=0}^n \text{Number of Ready Ins. in IQ}_i}{\sum_{i=0}^n \text{Number of Ready Ins. at the head of IQ}_i} \quad (3.3)$$

The most important thing to remember when collecting samples is that if you sample too often, the same instructions that are ready in the same IQ configuration can be counted multiple times, resulting in an IDR value that is misleading. During the tests, for example, we set the sampling interval to 100 cycles. A slow-running application, such as *bwaves* from the SPEC2006 CPU suite, can stall the IQ for hundreds of cycles. In this case, we risk counting and accumulating the number of ready instructions that we have already counted at consecutive sampling times.

To solve this problem, we recall the program counter (PC) of the instruction waiting on top of the IQ during a sampling period and compare it to the PC value of the instruction waiting on top of the IQ during the next sampling period. If these two PC values are the same, we

know that the same instruction is still waiting on top of the IQ and that nothing has changed since the last sampling period.

### 3.3. COMMIT OVER FETCH RATIO (CFR)

The final stage of the instruction completion process is the commitment stage. Some instructions are unable to progress to this stage. When a branch instruction is mispredicted in speculative processors with a dynamic branch prediction mechanism, many instructions following that branch instruction on the mispredicted path must be discarded to continue with the application's sound execution. At this point, we look at the ratio of instructions committed to instructions fetched over a given period, as shown in Equation 3.4 below. We can say that all instructions that entered the processor were successfully committed if this ratio is close to one. However, if this ratio is close to zero, we can conclude that the processor's predictions were consistently incorrect, and only a small number of instructions made it to the commitment stage. The remaining instructions should be removed from the processor in this case.

$$CFR = \frac{\text{Number of Instructions commit in a period}}{\text{Number of Instructions that are fetched in a period}} \quad (3.4)$$

## 4. APPLICATION CLASSIFICATION/EXECUTION MODE

This chapter is organized as follows: Section 4.1. with its subsections explains the implementation and design of ASMMP execution mode selection mechanism based on collected information from stage one. Also, choosing the most suitable instruction execution mode for running the application dynamically.

### 4.1. EXECUTION MODE SELECTION

To compare and clarify the advantages of the proposed ASMMP execution mode selection mechanism, we will first overlook and examine the composite core study proposed by Lukefahr and his team [8]. A composite core is made up of both large and small heterogeneous microengines. The ability of composite cores to select the most appropriate micro engines for running applications to save power without sacrificing too much performance is their most important property. To accomplish this, a composite core collects various processor statistics from the active microengine during runtime and attempts to predict the performance of the passive microengine. The microengine, which is more suitable for running the rest of the application, is determined by a migration decision circuitry bound to collect statistics and complex functions that are run sequentially in the method proposed in the literature.

Lukefahr et al. presented the first study on composite cores. ARM's big.LITTLE heterogeneous multi-core architecture is an excellent example of the composite core architecture. A 3-wide Out-of-Order Cortex-A15 superscalar microprocessor (large) and a 2-wide In-Order Cortex-A7 superscalar microprocessor (small) are located in the same core in this architecture. Lukefahr et al. define a composite core as a heterogeneous multi-core architecture consisting of one large and one small compute microengine that can provide both high performance and high energy savings. Core migration is coarse-grained in ARM's existing big.LITTLE architecture.

Many power-saving opportunities are lost in this architecture when running applications that frequently change behavior, as cross-core application migration occurs after billions of



instructions are executed. Similarly, an application that requires instant high performance while running on the small microengine remains connected to the small microengine for an extended time.

In conclusion, the current method lacks the reflex to react to sudden changes in the behavior of running applications. Lukefahr et al. now propose a fine-grained decision mechanism for core-to-core migration and a method that is more agile in adapting to the needs of running applications. The work proposed by Lukefahr and his team is referred to as the Composite Core (CC) study here. The method proposed in the CC is based on predicting the performance of the passive microengine using a function obtained through machine-learning methods that incorporate statistics from the active microengine.

The statistics gathered range from the L2 cache miss and hit rates to the branch prediction mechanism's misprediction percentage and from the instruction-level parallelism (ILP) calculated from the issue queue (IQ) to the roughly calculated memory level parallelism (MLP) using the miss status holding register (MSHR). To collect these statistics, complex tables, such as a dependency table, should be integrated into the small core. The performance estimator, threshold controller, and core switching control mechanisms are all part of the decision circuit, known as the reactive online controller.

Instead of using more than two heterogeneous cores, we propose the ASMMP architecture, which uses a single Out-of-Order core to switch between Out-of-Order and In-Order modes of instruction execution. As shown in Figure 4.1, ASMMP's execution mode circuitry comprises a mode switching decision mechanism and a mode switching enforcement mechanism. Furthermore, by sacrificing less than 5 percent of performance, our simple mode change decision circuitry, which is bound to only two processor statistics (IDR and CFR), can save more than 25 percent power, more than 21 percent on energy-delay products, and more than 16 percent on energy-delay-square products on average.

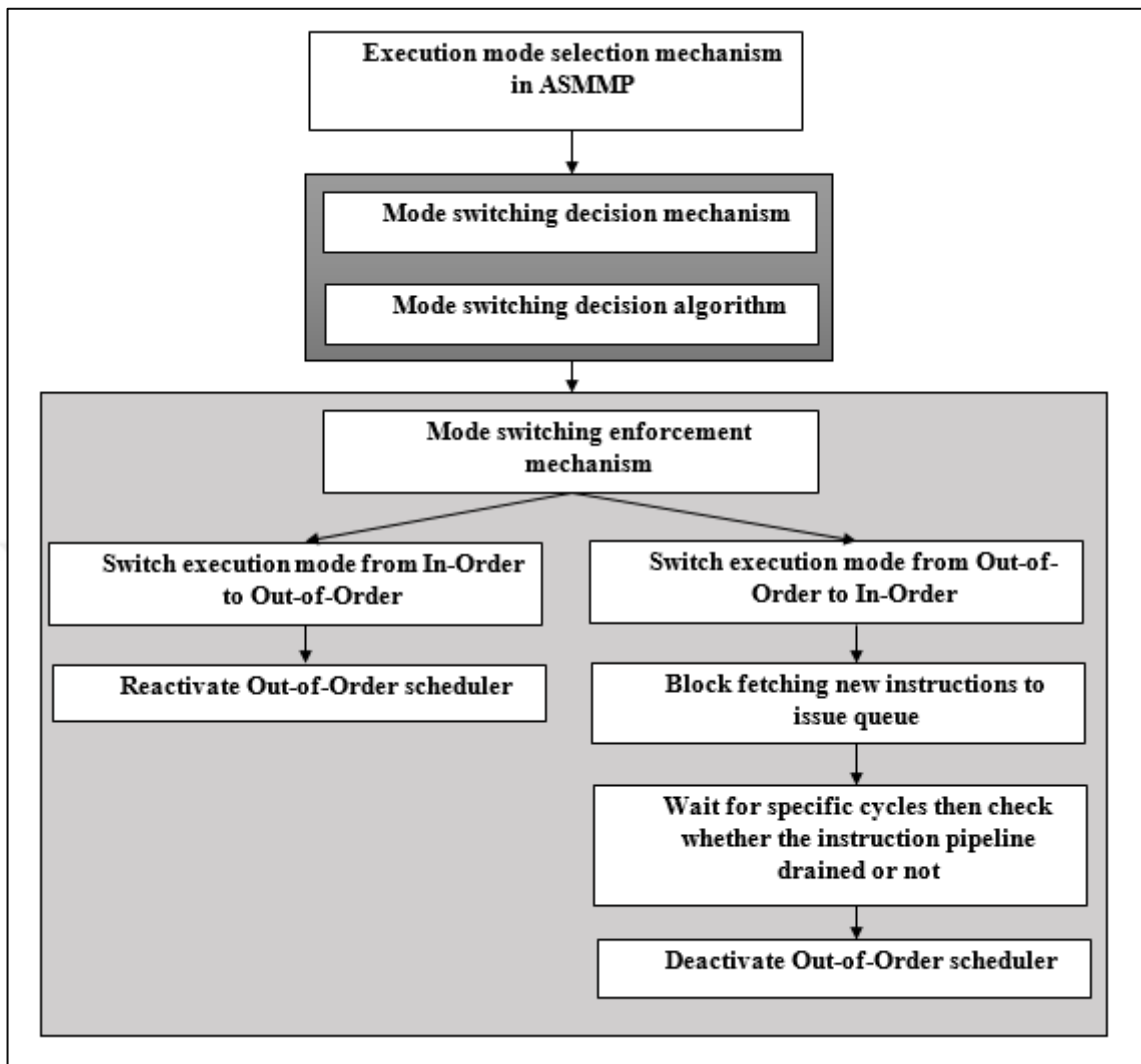


Figure 4.1. Execution mode selection policy in ASMMP

#### 4.1.1. ASMMP execution mode switching architecture

The ASMMP core architecture differs from Lukefahr and et al.'s composite core architecture [8]. Two heterogeneous microengines, one small and one large are tightly connected in the composite core architecture. The fetch logic, the branch prediction unit, the L1 instruction, and data caches are all shared in Lukefahr and his team's study [8]. However, data path structures such as the register alias table (RAT), physical register files (PRF), and load/store queue (LSQ) that are only used by the Out-of-Order microengine are not shared. The authors also suggest adding a small L0 cache to the small microengine in the article version of the CC, which is published by the same authors later [9]. A dependency table embedded in the In-Order core is used by the reactive online controller proposed in the CC.

Since it requires  $128 \times 32 \times 2$  bits, the dependency table requires a static random access memory (SRAM) space almost as large as the Re-order buffer (ROB). The reactive online controller's main function is to collect various performance metrics and use them to determine which core should be active for the next epoch. The CC architecture is depicted in Figure 4.2.

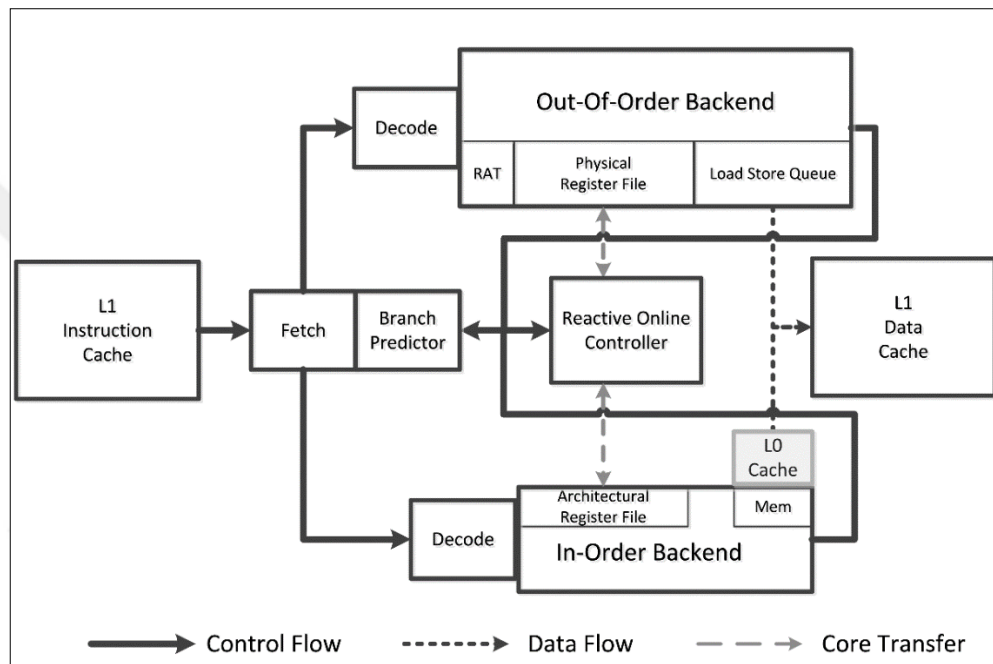


Figure 4.2. Composite core architecture [8]

As illustrated in Figure 4.3, the execution mode selection in ASMMP is based on the aforementioned composite core architecture. ASMMP architecture includes a single Out-of-Order microengine that functions as an In-Order microengine when needed. The ASMMP requires less space and no additional structures for the prediction hardware. Furthermore, because only one core can switch between execution modes, no core migration logic is required. Since the data path structures holding live instructions can continue working without interruption, the mode switch process from In-Order execution mode to Out-of-Order execution mode happens almost instantly. For example, when the Out-of-Order scheduler is activated, the Issue Queue (IQ) that schedules instructions in program order can continue its work. However, switching from Out-of-Order execution mode to In-Order execution mode may take some time. In such a case, the fetch stage must be throttled, and

all data path structures that are currently running instructions Out-of-Order must be drained first. Then, the In-Order execution mode can be used.

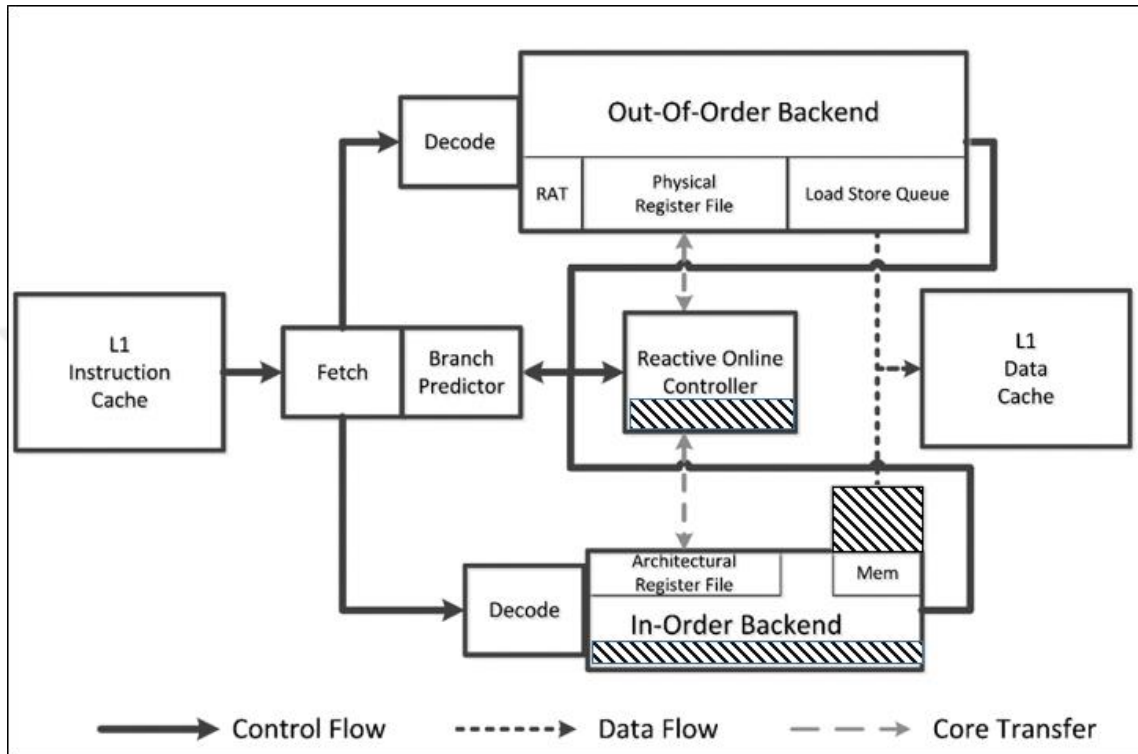


Figure 4.3. Execution mode selection architecture in ASMMP

#### 4.1.2. Mode Switching Decision Mechanism

Mode switching decision mechanism identifies the most appropriate instruction execution mode for the running application for a specific period. The decision depends on two main statistics: instruction dispatch ratio and instruction commit over fetch ratio of the running application. The used statistics are already obtained in the first stage of the study to classify applications based on their behavior into three main groups. These groups are as follows:

1. An application with high Instruction Level Parallelism (ILP) phases needs an Out-of-Order execution mode to perform well with less power-saving opportunities.
2. An application with low ILP phases needs an In-Order execution mode to perform well and many power-saving opportunities.

3. A hybrid application with both high and low ILP phases during its lifetime needs dynamic switching in execution mode to save power without sacrificing too much performance.

To determine instruction execution mode for the running application, we need the product of instruction commit over fetch ratio with instruction dispatch ratio to determine how well the Out-of-Order mode performs compared to its In-Order counterpart. Equation 4.1 shows us the speedup value ( $S$ ) of the Out-of-Order mode over the In-Order mode.

$$S = IDR * CFR \quad (4.1)$$

First and foremost, given the obvious performance advantage of the Out-of-Order mode over the In-Order mode, we do not expect the speedup value obtained from Equation 4.1 to be less than one. On the other hand, the  $S$  value can get very close to 0 due to the value of CFR, which indirectly expresses the percentage of instructions inserted into the processor due to mispredictions in the dynamic branch prediction mechanism. In such cases, we can deduce that the Out-of-Order execution mode loses its advantage over the In-Order mode, and that is now the best time to switch from Out-of-Order to In-Order mode in order to save more power. Equation 4.2 shows the effect of the  $S$  value on the mode selection stage of the decision circuit.

The decision circuit selects the Out-of-Order mode when the calculated  $S$  value is greater than a fixed threshold value ( $\alpha$ ). In its absence, the In-Order mode is selected. Low threshold values result in more Out-of-Order mode selections, while high threshold values result in more In-Order mode selections. The impact of this threshold value on performance is reported in the tests and results section. Another option is to match the threshold value with the needs of applications running in the background. In this case, it may be possible to prevent applications' performance from degrading beyond a certain point while revealing a more stable and reliable decision mechanism.

$$Mode = \begin{cases} Out - of - order, & S > \alpha \\ In - order, & S \leq \alpha \end{cases} \quad (4.2)$$

#### 4.1.3. Mode Switching Enforcement Mechanism

In ASMMP, switching between execution modes must be as smooth as possible. In this case, the enforcement mechanism takes responsibility and adopts mode-change decisions as soon as possible. The issue queue is one of the key data path structures distinguishing between Out-of-Order and In-Order execution modes. IQ is a queue structure in an In-Order processor, where the head of the queue holds the oldest instruction and the tail of the queue holds the youngest instruction in program order. The IO mode instruction scheduler is straightforward enough to schedule instructions from the IQ's head once they are ready (i.e., all source operands of the instruction become valid).

The Out-of-Order mode instruction scheduler, on the other hand, is far more complicated, as it is responsible for locating ready instructions that may be located anywhere in the IQ. IQ is no longer a queue structure in an Out-of-Order processor, and it is now known as the instruction dispatch buffer (IDB). The instructions in the IQ must be in program order when switching from Out-of-Order to In-Order execution mode. As a result, when a mode switch decision is made, the ASMMP enforcement mechanism stops fetching new instructions and waits for the instruction pipeline to empty.

If the IQ contains long-latency (such as floating-point divide and square root) and non-deterministic latency (such as memory) instructions, this process may take a long time. If the mode transition is not completed within five thousand cycles, the pipeline contents are flushed to avoid long delays in the enforcement mechanism (an empirical time, which gives us good feedback in our experiments). The Out-of-Order instruction scheduler is deactivated when the pipeline is empty, and a simpler In-Order instruction scheduler is activated. Meanwhile, instructions are being dispatched into the IQ in program order.

It is much easier to switch from In-Order to Out-of-Order execution mode. An IQ organization is not required for the Out-of-Order scheduler because it can dispatch instructions in any arbitrary sequence and issue them promptly. A simple reactivation of the complex Out-of-Order scheduler and deactivation of the In-Order scheduler is needed to switch from In-Order mode to Out-of-Order mode.

#### 4.1.4. Periodic Operation

When collecting statistics, we use two distinct periods: the sampling period (SP) and the decision period (DP). There are many SPs in each DP. We collect and accumulate the number of ready instructions in the IQ at the end of each SP, as shown in Figure 4.4. We calculate IDR and CFR values at the end of a DP and compare the S value with the threshold value. Then, we choose which execution mode is best for running our application until the next DP arrives. We will continue to work in the target mode if a mode switching decision is made.

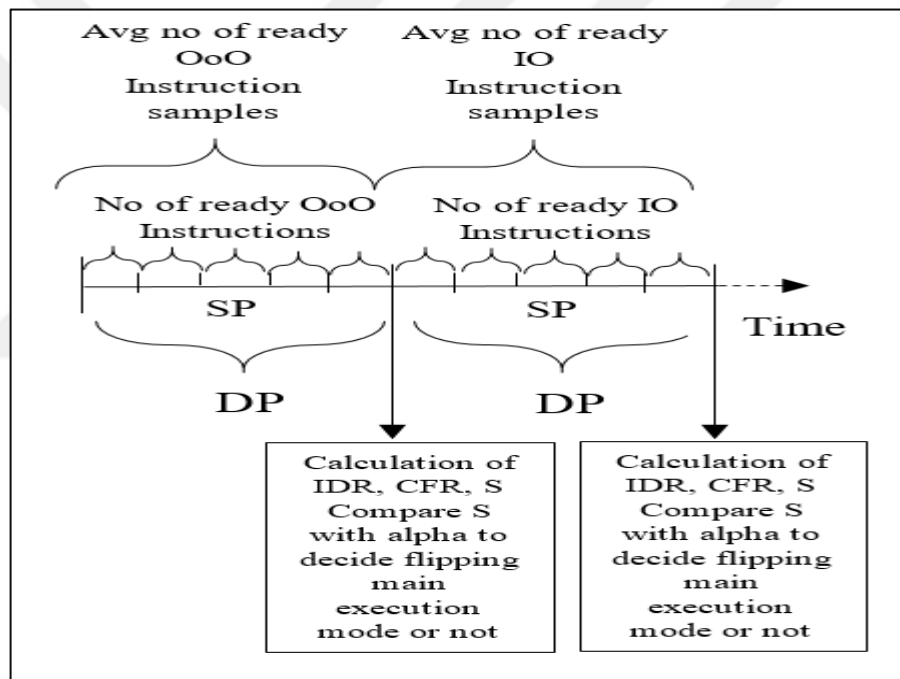


Figure 4.4. Periodic operation of execution mode selection in ASMMP

#### 4.1.5. Hardware

The CC contains the miss and hit rates collected from the L2 cache, the misprediction rate of the dynamic branch prediction mechanism, counters that collect parallelism levels at the instruction and memory levels, a multiplier-accumulator (MAC) that performs the performance prediction function trained by the offline machine-learning method, and a reactive online controller circuit that also takes into account error rate, and a dependency table to collect ILP information missing in the In-Order processor require extra hardware

that takes up about 3 percent of the total processor area. The execution mode selection architecture described in ASMMP, on the other hand, comprises a pipelined division circuit for IDR and CFR computations, a multiplication circuit for calculating the S value, and a comparison circuit for comparing it with the threshold value.

In ASMMP, the execution mode selection architecture is implemented using a hardware solution. However, we can explain the functioning of the Execution mode selection architecture in terms of pseudocode, as shown in Algorithm 4.1. The variables displayed as current commit count and current fetch count in the pseudocode represent the commit and fetch counts already collected by many modern processors, as depicted in the pseudocode. Instead of sampling the fetch and commit counts in every sampling period in H/W (which is more frequent than the decision period), the execution mode selection circuit remembers the fetch and commit counts from the previous period at the beginning of the new decision period (represented as prev fetch and prev commit).

The decision algorithm requires two subtraction operations and two-division operations. These operations can be performed consecutively to minimize hardware complexity, requiring only a single subtractor and divider to complete the algorithm. A comparator is essentially a subtractor. The same unit can be used for both subtractions and comparisons because they are independent operations. For the sampling periods, this subtractor can also be used as an adder. As a result, the Execution mode selection algorithm can be implemented using only a comparator (used in the algorithm for subtraction, comparison, and addition), a divider, and a multiplier. It is also worth noting that these units do not necessarily need to be full 32-bit units. Simpler computation units may suffice due to the short sampling and decision periods. Table 4.1 shows approximate transistor count needed to implement execution mode selection mechanism in ASMMP.

In summary, except for the two statistical variables previously collected in the CC, there is no MAC circuit or dependency table complexity as we proposed. As a result, compared to the composite core circuit, the suggested execution mode selection approach in ASMMP is more advantageous in terms of delay values, area, and power consumption.



Algorithm 4.1. Operation of execution mode selection in ASMMP

```

if current cycle mod 10000 == 0 then
    total_commit  $\leftarrow$  current commit count – prev_commit
    total_fetch  $\leftarrow$  current fetch count – prev_fetch
    prev_commit  $\leftarrow$  current commit count
    prev_fetch  $\leftarrow$  current fetch count
    IDR  $\leftarrow$  total_ready / total_ready_head
    CFR  $\leftarrow$  total_commit / total_fetch
    S  $\leftarrow$  IDR * CFR
    total_ready  $\leftarrow$  0
    total_ready_head  $\leftarrow$  0
    if S >  $\alpha$  and current execution mode is In-Order then
        Switch to Out-of-Order execution mode
    else
        if S  $\leq$   $\alpha$  and current execution mode is Out-of-Order then
            Switch to In-Order execution mode
else
    if current cycle mode 100 == 0 then
        if current PC head  $\neq$  prev_pc_head then
            total_ready += number of ready instructions in IQ
            total_ready_head += number of ready instructions at the head of IQ

```

Table 4.1. Transistor count for execution mode selection mechanism

Hardware circuit	Approximate transistor number ( bit number * transistor count)
13 bit full adder	13*20=260
15 bit comparator	15*10=150
6 bit division	6*144= 864
4 bit multiplication	4*24=96
6 bit counter	6*20=120
15 bit register	15*6=90

## 5. APPLICATION CLASSIFICATION/RESOURCE PARTITIONING

This chapter is organized as follows: Sections 5.1 and 5.2 explain the implementation and design of ASMMP resource partitioning based on collected occupancy information from stage one. Then, the method behind assigning suitable processor resources to the running application is explained.

The design stage determines the size of various structures within the processors. The size of these structures is determined by the design of a high-performance processor. However, regardless of the processor's power, some applications will not always require full structure capacity. In some cases, using only a small portion of the structures will be sufficient to achieve very close performance to the highest possible from the running application.

In such cases, at the design stage, although a significant part of these structures, which is decided to be large, is not needed, they will continue to consume energy. In similar cases, it may be possible to reduce structure size without adversely affecting the performance and reduce the processor's energy consumption. To achieve this goal, we need to perform resource partitioning or adaptive processing. Adaptive processing is a straightforward solution to dynamically disable portions from processor resources according to the runtime needs of workloads [4], [5], [6], [7]. This thesis will follow the coarse grain resource partitioning (open/close for one large partition) path rather than the fine-grain (open/close for a small number of entries) path. Coarse grain partition has lower hardware complexity and provides more functional resource access methods than the fine-grain option. Figure 5.1 shows implemented resource partitioning policy and its components. The following subsections give a brief explanation of these components.

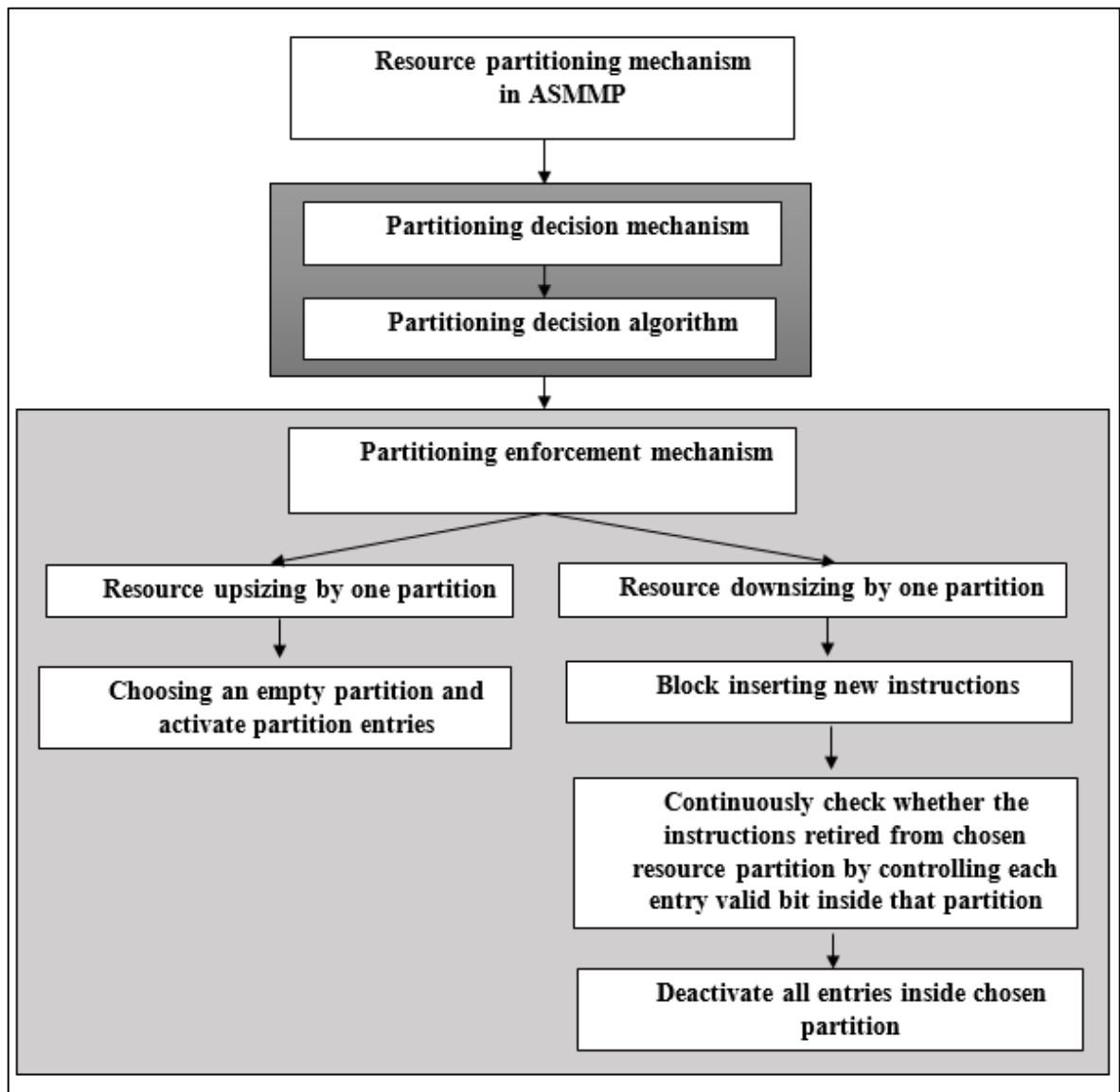


Figure 5.1. Resource partitioning in ASMMP

### 5.1. PARTITIONING DECISION MECHANISM

The partition decision mechanism decides to upsize or downsize processor resources depending on the collected statistics at the end of each epoch for running the application. We use resource occupancy statistics obtained in the first stage of the study to classify applications based on their resource demands into two main groups. These groups are as follows:

1. High utility applications which use most of the processor resources.
2. Low utility applications which use few of the processor resources.

The purpose of dividing applications into two main categories is to help use processor resources more efficiently and save power without noticeable loss of application performance. The partition decision mechanism for resource upsizing/downsizing is examined in detail in the following subsections.

### **5.1.1. Resource downsizing decision**

We need to look for a sign that not all resource entries are used by an application to trigger resource downsizing operations. In this thesis, we examine the occupancy parameter of different processor resources. For example, if the average number of elements contained in the resource does not exceed half the total number of elements of this structure in certain periods, the resource size will be reduced, and energy savings will be achieved. However, at this point, we need to choose the type of size reduction method too. Following an aggressive size reduction method that deprecates half of the processor resource size can result in more than 5 percent performance degradation. In this case, we will use a more cautious method of size reduction, which is the closure of one partition and the closure of other partitions according to the average number of elements observed subsequently to prevent performance degradation.

Algorithm 5.1 shows our partition decision mechanism for resource downsizing. At the end of each period, we check the difference between resource size( $G$ ) and average resource occupancy( $D$ ) of the running application. If it is larger than the specific threshold( $T$ ), we decrement resource size by one portion( $B$ ), which is equal to eight entries in our study. After downsizing decision, we allow instructions to retire from the resource while preventing inserting new instructions to the resource. Furthermore, this operation may take a long time for a short duration at the beginning of the new period to assign a new resource size ( $G-B$ ) to the running application.

Algorithm 5.1. Resource downsizing in ASMMP

1. *If DP end is not reached go to step 7*
2.  $D \leftarrow \text{Average resource occupancy}$
3.  $G \leftarrow \text{Resource size}$
4.  $B \leftarrow \text{One partition size}$
5.  $T \leftarrow B+2$
6. *If  $(G-D) > T$  true then downsize resource by  $B$  entries*
7. *End*

### 5.1.2. Resource Upsizing Decision

In the resource upsizing algorithm, here, we can choose aggressive or cautious resource upsizing methods. In the aggressive method, when resource occupancy value exceeds a certain threshold value, a decision can be made to increase resource size by one portion containing eight entries in our case immediately without waiting for the end of the period, or multiple portions can be opened instantly (again without waiting for the end of the period). On the contrary, in the cautious method, the resource upsizing process can be performed by adding only one portion at the end of each decision step (waiting until the end of the epoch). As a result, it is possible to try a wide range of alternative algorithms for resource upsizing.

We will follow an algorithm similar to the resource downsizing algorithm. When the average resource occupancy for the running application through a specific period is larger than the specific threshold, we increase resource size by one portion. Also, to perform resource upsizing, we do not have to wait till the end of the period to prevent performance loss of the running application. Algorithm 5.2 shows our partition decision mechanism for resource upsizing.

Algorithm 5.2. Resource upsizing in ASMMP

1.  $D \leftarrow$  *Average resource occupancy*
2.  $G \leftarrow$  *Resource size*
3.  $B \leftarrow$  *One partition size*
4.  $T \leftarrow B+2$
5. *If  $(G-D) \leq T$  true then upsize resource by  $B$  entries*
6. *End*

### 5.1.3. The Use of Statistical Parameters

In the first stage of ASMMP, we monitor and collect different statistical parameters such as average resource occupancy, how many time a resource has been full or not, etc., in every period. For each new decision period, we can follow two ways of using the statistical parameters. The first way is to consider old values from the previous periods (partial transfer of parameters history) and give them a certain weight (most studies give them 50 percent impact). Then, merge these values with the new ones for determining resource upsizing or downsizing decisions. The second way is to reset parameter values and not merge them with new period data. We follow the second method in this thesis to reduce the hardware complexity of ASMMP.

Moreover, we believe that the threshold value for both resources upsizing and downsizing should be different to prevent oscillation phenomena. Using the same threshold value may lead to continuous activation of resource scaling.

This oscillation may result in performance degradation of the running application with fewer energy-saving opportunities. Lastly, Figure 5.2 shows a flowchart for ASMMP partition decision mechanism.

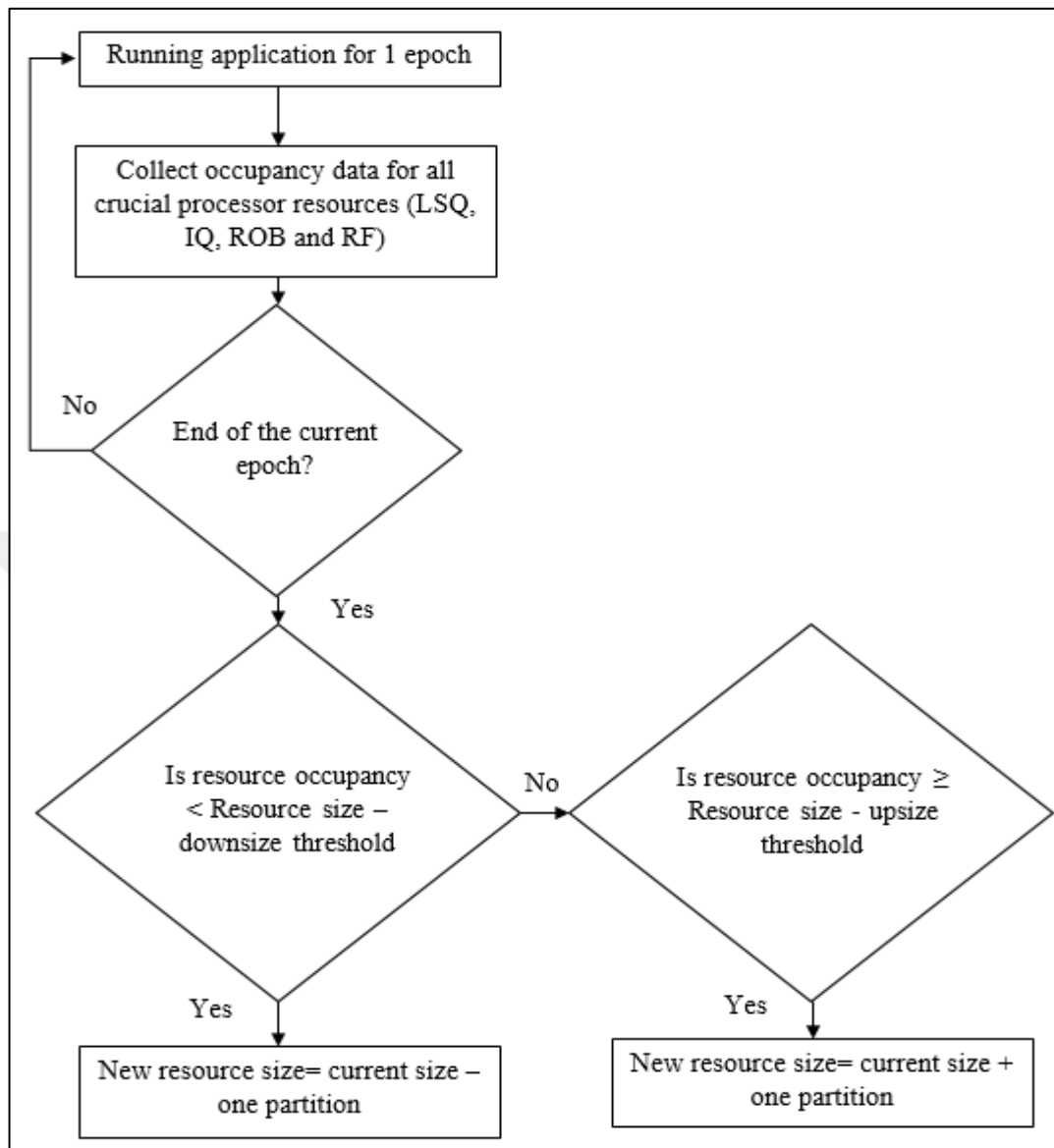


Figure 5.2. Partition decision mechanism flowchart for ASMMP

## 5.2. PARTITIONING ENFORCEMENT MECHANISM

The partition enforcement mechanism in ASMMP has two major purposes. The first purpose is to take information from the partition decision mechanism and then continuously check whether the conditions are satisfied to turn on/off different processor resources. The second purpose is to enforce that the new target amount of resource entries do not exceed the maximum allowable resource size and vice versa.

In this section, we classify ASMPP resources based on their different characteristics, such as resource nature, occupancy, and frequency of resource use. Then, we define suitable conditions based on these characteristics. Then, we will check these conditions during the partition enforcement operation. The fulfillment of these conditions will help complete the scaling process with minimum hardware complexity. The most important condition we will follow is to divide the resource into partitions. Each partition consists of eight entries. Also, at the end of each scaling decision, only one partition can be turned on or off ( different available entries from multiple partitions are not allowed to be turned on/off ). In the following subsections, we will examine ASMMP resource structures.

### **5.2.1. Circular Queue Resources**

ASMPP resource structures such as Re-order buffer, load queue, and store queue are circular queues where instructions are inserted into the queue in an In-Order manner and instruction removal in the same order as insertion. A circular queue has two pointers (Head pointer) that point to the beginning of the queue (oldest inserted instruction ) and (Tail pointer) pointing to the end of the queue (newest inserted instruction). So, every newly arrived instruction will be placed at the end of the queue, and every retired instruction will be removed from the beginning of the queue.

The conditions needed to be satisfied by the partitioning enforcement mechanism to downsize circular queue structures in ASMMP are continuously to check all entries in the specified partition and mark each empty entry in that partition and second, to block the arrival of the new instructions to that partition.

Moreover, on each instruction, removal from that partition keeps marking the entry and repeats the process till all entries in that partition become empty. Finally, the partitioning enforcement mechanism deactivates the chosen partition.

The conditions needed to be satisfied by partitioning enforcement mechanism to upsize circular queue structures in ASMMP is to allow insertion of sequential and one-piece of instructions to available partition without creating gaps between resource partitions. Furthermore, we can achieve the mentioned condition by checking the position of the head



pointer and make sure that it should be positioned before the tail pointer. After satisfying the above conditions, our partition enforcement mechanism will activate the resource partition, and it will be ready to be used by running the application.

### **5.2.2. Instruction Queue (IQ)**

The most important functioning of Out-of-Order execution is the execution of instructions based on the "data flow" graph (rather than program order) by keeping the semantics of the original program. The HW examines a sliding window of consecutive instructions (The "instruction window"). Later, ready instructions get picked up from the window( instruction queue keeps these instructions) and executed out of program order. Lastly, instruction results are committed to the machine state (memory and architectural register file) in original program order. Due to the nature of the instruction queue, ready instructions can be selected out of program order. This increases the complexity of the instruction selection circuit, but at the same time, it allows instructions to be stored in a mixed order inside the instruction queue.

The instruction queue scaling operation is easier than other resource scaling processes because the instructions do not need to be stored in a certain order. Thus, the Partition enforcement mechanism can directly (upscale) activate and add a partition to the end of the instruction queue without the need to check any conditions. Whereas, during the downsizing process, the partition enforcement mechanism blocks fetching new instructions to the instruction queue and waits until all partition entries become empty. After satisfying the mentioned conditions partition enforcement mechanism turn off that partition.

Another remarkable point here is that the processor replaces new instructions at the beginning of the instruction queue, accumulated in density at the beginning of the queue with relatively empty spaces at the end of the queue. The mentioned point helps partition enforcement mechanism to complete the size reduction process more quickly.

### 5.2.3. Register Files

Register files are structures that play a critical role in the register renaming mechanism, which helps eliminate all false data dependencies among instructions. To perform all these jobs, register files consume more power than other resources. Instructions (operands) are stored in the register file out of program order, similar to the instruction queue case. However, the main difference between both structures is that instructions in the instruction queue can be removed from this structure as the instructions are executed, while the data in the register files must remain in place while the corresponding instructions are on the processor. As a result, performing register file scaling requires either waiting until the instructions retire from the queue or doing nothing by waiting for a specific period and then flushing the structure.

There are two ways we can use to reduce register file size. The first method is to transfer the values inside a specific partition of the register file to other entries in other available partitions and then update the register rename table. The second method is to wait for the oldest instructions in the register file to commit totally from the processor, which later leads to evacuation of its data from the register file. In this thesis, we follow the second method to turn off the registers since the first option requires additional circuit, time, and energy consumption to be implemented. Lastly, the partition enforcement mechanism can directly upsize the register file without any specific condition. The upsizing process can simply be done by marking activated registers as available in the register rename table. Figures 5.3 and 5.4 show a flowchart for the scaling process followed by the partition enforcement mechanism in ASMMP.

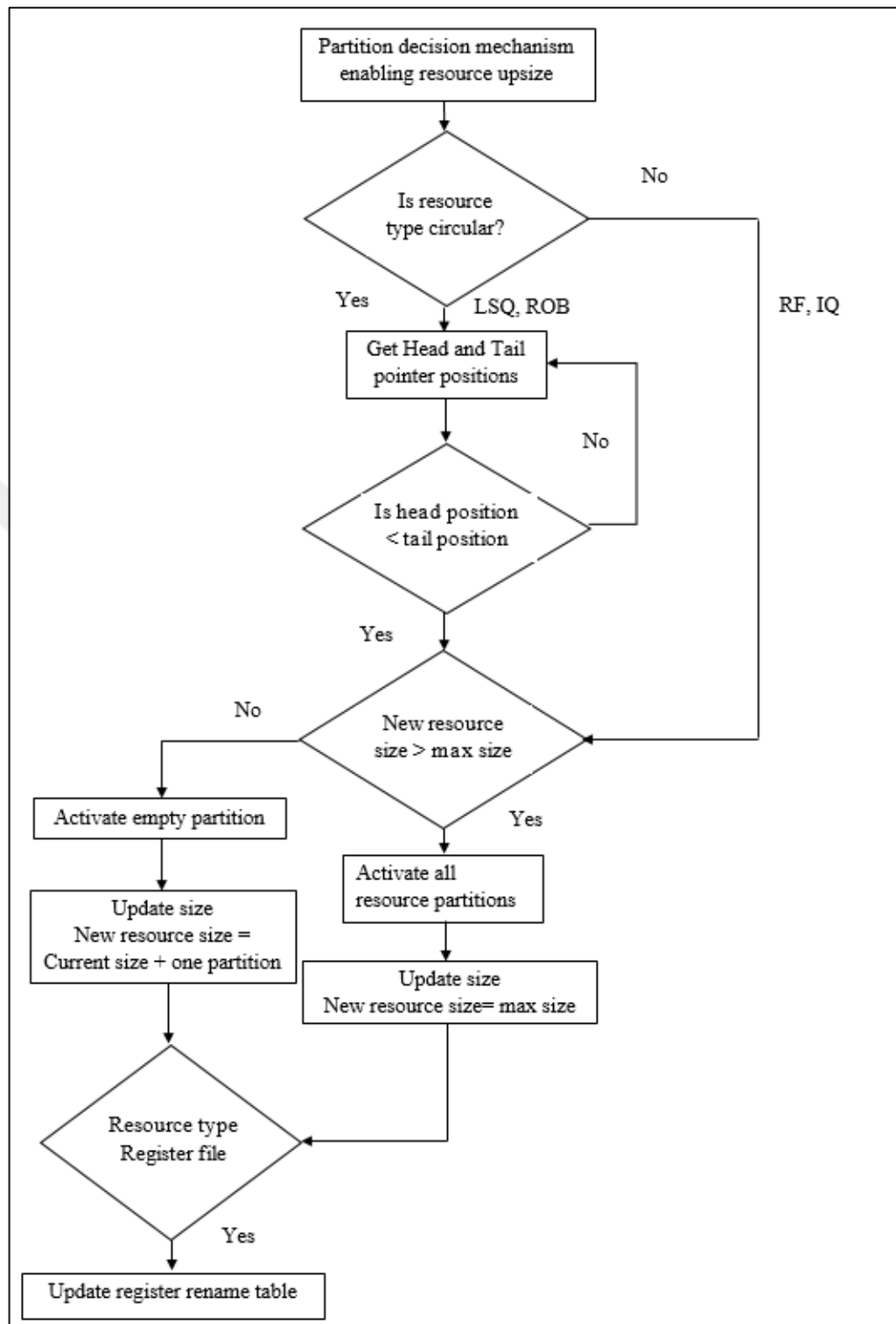


Figure 5.3. Partition enforcement mechanism for resource upsizing in ASMMP

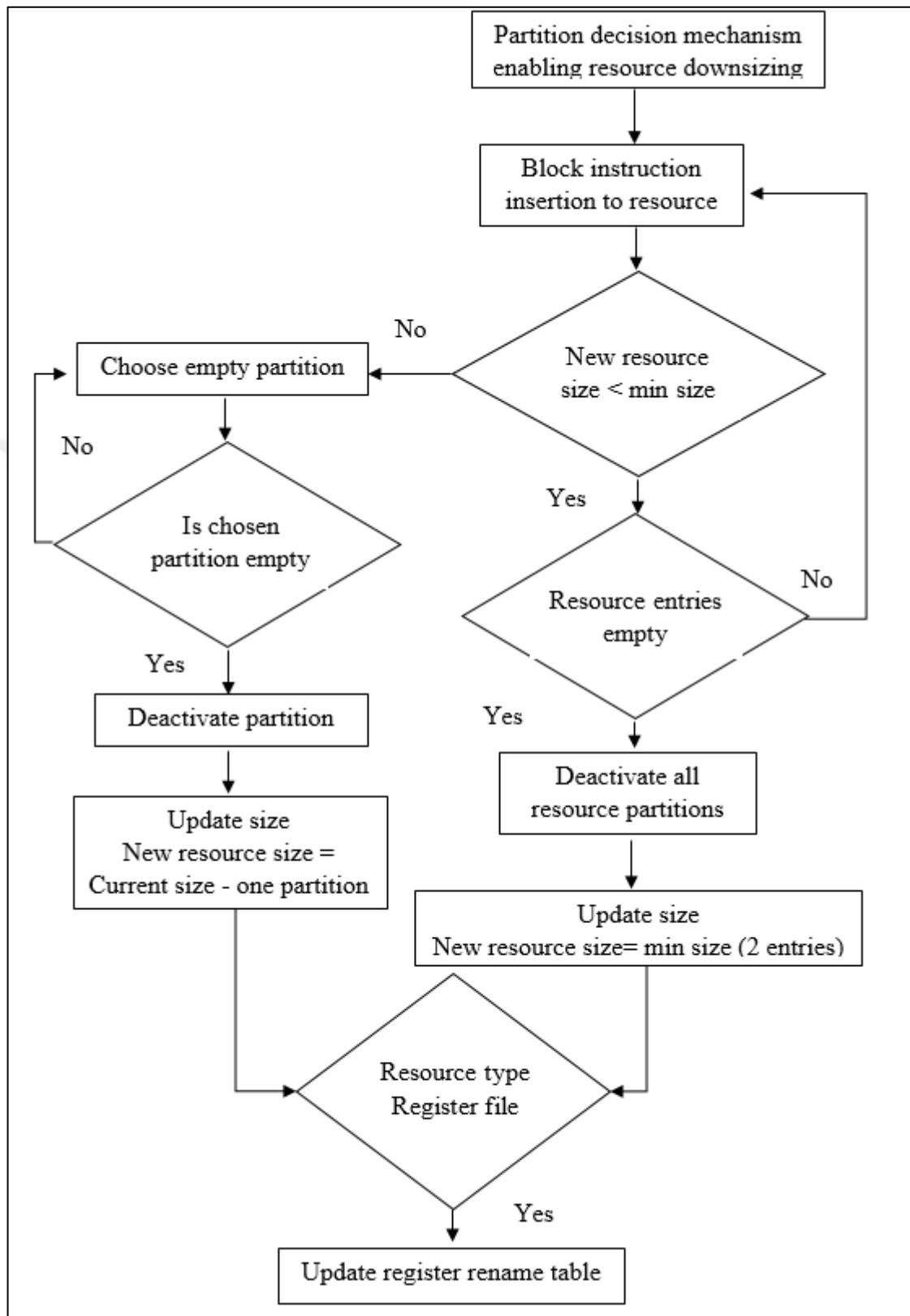


Figure 5.4. Partition enforcement mechanism for resource downsizing in AS MMP

### 5.3. ASMMP OPERATION

The purpose of this section is to examine and discuss ASMMP's operation and the results obtained by integrating the resource partitioning mechanism with the execution mode selection mechanism to save power not only through dynamic execution mode switching but also by turning off idle processors resources during Out-of-Order execution mode.

#### 5.3.1. Issuing Right Processor Configuration

The primary objective of this thesis is to design a single hybrid processor capable of limiting application performance degradation to less than 5 percent and minimizing power consumption. This objective is accomplished by using the two distinct methods. The first is to dynamically partition resources by monitoring occupancy information during an application's Out-of-Order execution mode and saving by turning off unused processor resources. The second is periodic switching between In-Order and Out-of-Order execution modes, with power savings resulting from the absence of complex Out-of-Order execution mode structures for a while. Mode switching is accomplished by monitoring certain processor statistics, such as the number of committed and fetched instructions in the running application, which are not as difficult to collect and monitor as in CC [8]. Additionally, the proposed method overcame the disadvantages of iMODE [29]. iMODE is another technique for changing execution modes based on trial periods [29]. The primary disadvantage of this method is that it uses brief trial periods at the end of the main Out-of-Order execution mode, which slows down high-performance applications. Although iMODE attempts to mitigate the slowdown caused by this problem by postponing the next decision point based on a specified delay threshold value, applications that require full speed will still experience a slowdown.

These two methods can be combined by selecting the most appropriate processor configuration for the running application. Figure 5.5 shows a flowchart for the final proposed ASMMP architecture and how these two methods are combined to operate together.

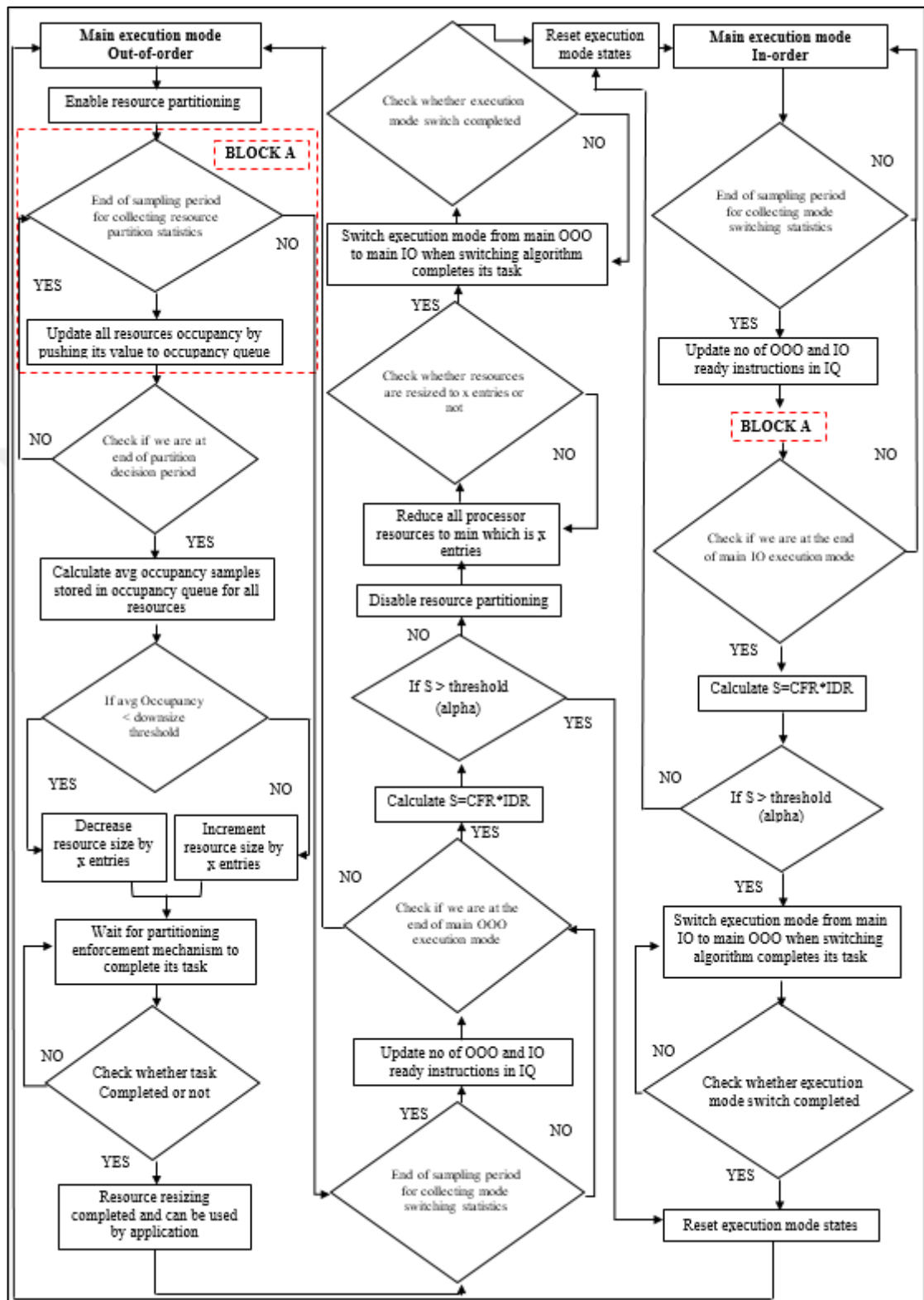


Figure 5.5. Flowchart for proposed AS MMP architecture

As can be seen from the flowchart, the top of our flowchart specifies distinct steps for Out-of-Order and In-Order execution modes. The left section illustrates the steps to be taken

while in the Out-of-Order mode, and the right section illustrates the steps to be taken while in the In-Order mode. The mode switching decision mechanism described in Section 4.1 governs the selection of an Out-of-Order execution mode. ASMMP collects the necessary statistical data required by mode decision mechanisms (IDR, CFR) values to calculate system speed up (S) using the sampling method. Additionally, resource partitioning in ASMMP is dependent on the mechanism for resource partitioning described in Sections 5.1 and 5.2, respectively. The average occupancy of the Re-order buffer, load queue, store queue, issue queue, and register files is determined using the same sampling method. The occupancy data is used by the resource partitioning decision mechanism, which helps allocate sufficient resource space to run applications and shut down unused entries to conserve energy.

At first, the application ran in an Out-of-Order mode with adaptive resource processing. At the ending of each partition decision period, the average occupancy value for all processor resources is calculated and compared to the threshold value (downsize threshold), based on the comparison result of either resource downsizing or upsizing by one partition applied after satisfying all partition enforcement mechanism requirements. Following that, at the ending of each execution mode decision period, the system speed up (S) value is calculated and compared to the threshold value (alpha threshold). We retain the Out-of-Order execution mode if the speed-up value exceeds the specified alpha threshold. Otherwise, we begin the process of switching the execution mode to In-Order mode. Before initiating the switching process, we must disable the resource partitioning mechanism, which will result in only one partition per resource being activated. All remaining partitions will be disabled (the partition enforcement mechanism responsible for this operation) since we do not require extensive use of these resources in the In-Order execution mode. Additionally, by deactivating most resource partitions, we expect to achieve approximately fourfold power savings when an application runs in Out-of-Order mode, as mentioned in the CC study.

Later, we wait for the mode switching enforcement mechanism to complete the process of switching by allowing active instructions to complete their execution and be retired. Finally, two additional points should be made. The first is related to the resource occupancy collection mechanism, which remains active when we are in the In-Order mode, and at the ending of each sampling collection period, a value for resource occupancy is inserted into

the sampling queue. The final and the most critical point concerns collecting statistical data that will be used to forecast the next execution mode. The sample period is critical in this situation because a short sample period results in frequent sampling.

Additionally, we will continue to count the same ready instructions in the instruction queue, which will result in inaccurate information about the number of ready instructions. For example, collecting ready instructions every 100 cycles may cause issues for slow applications, such as *bwaves*, which can sit in the instruction queue for hundreds of cycles without progressing. In this case, we risk counting and accumulating the same number of ready instructions as we did during the initial sampling time.

To address this issue, we store the program counter value of the instruction waiting in the instruction queue at the end of the sampling period in a register and compare it with the program counter value of the instruction waiting in the queue at the beginning of the next sampling period. If these two program counter values match, we know that the same instruction is still waiting at the top of the queue and that the instruction queue situation has not changed since the previous sampling period, and thus, we avoid counting the number of ready instructions multiple times. By employing this technique, we were able to identify slow benchmarks (*bwaves*, *leslie3d*, and *sjeng*) and execute them in order with minimal performance degradation and significant power savings in our proposed ASMMP.



## 6. TESTS AND RESULTS

This chapter presents the simulation results for ASMMP in terms of performance, power savings, and efficiency. The results of evaluating the execution mode selection mechanism in terms of performance, power savings, and energy efficiency are presented and discussed in Section 6.1. The resource partitioning mechanism is evaluated in Section 6.2 as a power-saving technique that can be applied to various processor resources. Section 6.3 summarizes and discusses the results of the performance, power savings, and energy efficiency evaluations of the final ASMMP design.

### 6.1. EXECUTION MODE SELECTION TESTS AND RESULTS

The Gem5 simulator is still being used to test and evaluate the proposed execution mode switching architecture in ASMMP [2]. SPEC CPU2006 benchmarks are compiled in the x86 architecture and run with reference input files until each benchmark completes 100 million instructions [3]. Table 6.1 shows the properties of the simulated In-Order and Out-of-Order processors.

Table 6.1. Specification of the simulated processor during execution mode switching

Processor Microarchitecture	x86
Processor Frequency	2 GHz
Machine width	4
Re-order Buffer size	192
Issue Queue size	64
Load Queue/Store Queue size	32
Physical Register File size	256
L1 instruction and data cache	4-way LRU, 16 KB each
L2 cache	8-way LRU, 128 KB
Decision Period (DP)	10000 cycles
Sampling Period (SP)	100 cycles
$\alpha$ Thresholds	1, 3, 5

Three criteria are used to evaluate simulation results: performance, power savings, and energy efficiency. To calculate the processor's energy consumption per unit time, it is assumed that the Out-of-Order mode consumes four times the energy of the In-Order mode, as stated in the CC [8]. Power expenditure (PE) is calculated as stated in Equation 6.1, where  $W$  is the proportion of time spent in the In-Order processor. The reported power savings (PS) in the results section are determined using Equation 6.2. In terms of the average instructions per cycle (IPC) metric, Equation 6.3 presents the calculation for the slowdown rate (SR) of the proposed execution mode switching architecture in ASMMP compared to the baseline Out-of-Order processor. Finally, Equations 6.4 and 6.5 reflect the efficiency formula in terms of energy-delay product (EDP) and energy-delay-square product (ED<sup>2</sup>P), respectively.

$$PE = \frac{1}{4} * W + (1 - W) \quad (6.1)$$

$$PS = 1 - PE \quad (6.2)$$

$$SR = \frac{IPC_{baseline}}{IPC_{adaptive}} \quad (6.3)$$

$$EDP = PE * SR \quad (6.4)$$

$$ED^2P = PE * SR^2 \quad (6.5)$$

First, we run the spec 2006 benchmarks in both Out-of-Order and IO execution modes and collect performance data. Table 6.2 compares the performance of applications running In-Order with their Out-of-Order counterparts. In some applications, the performance degradation is almost imperceptible (bwaves 0.3 percent , leslie3d 1.4 percent, and sjeng 1.8 percent ). When running such applications, we expect the mode selection circuit in ASMMP to prefer the In-Order mode in order to save more power.

Other benchmarks exhibit a significant performance degradation (gromacs 68.5 percent and games 65.6 percent ) when In-Order execution mode is chosen. When running such applications, we anticipate that the Out-of-Order mode will provide the best performance.

Table 6.2. Performance drop of In-Order mode

Application	Performance drop(%)	Application	Performance drop (%)
astar	59,4	h264	62,5
bwaves	0,3	hmmer	65,1
bzip2	16,1	leslie3d	1,4
calculix	50,3	libquantum	17,4
games	65,6	mcf	59,7
gems	44,6	milc	50
gobmk	23,3	namd	43,2
gromacs	68,5	sjeng	1,8
		average	39,3

We simulate the proposed execution mode switching circuit with various threshold values (1, 3, and 5) and then record the percentage of In-Order runtime as illustrated in Figure 6.1. As expected, our proposed mechanism succeeded in most cases in selecting an In-Order execution mode when the performance difference between the two execution modes is close to zero for the running application. This statement holds true for the bwaves, leslie3d, and sjeng benchmarks. However, in applications where the performance difference between the two modes is significant, as expected, the Out-of-Order mode is preferred.

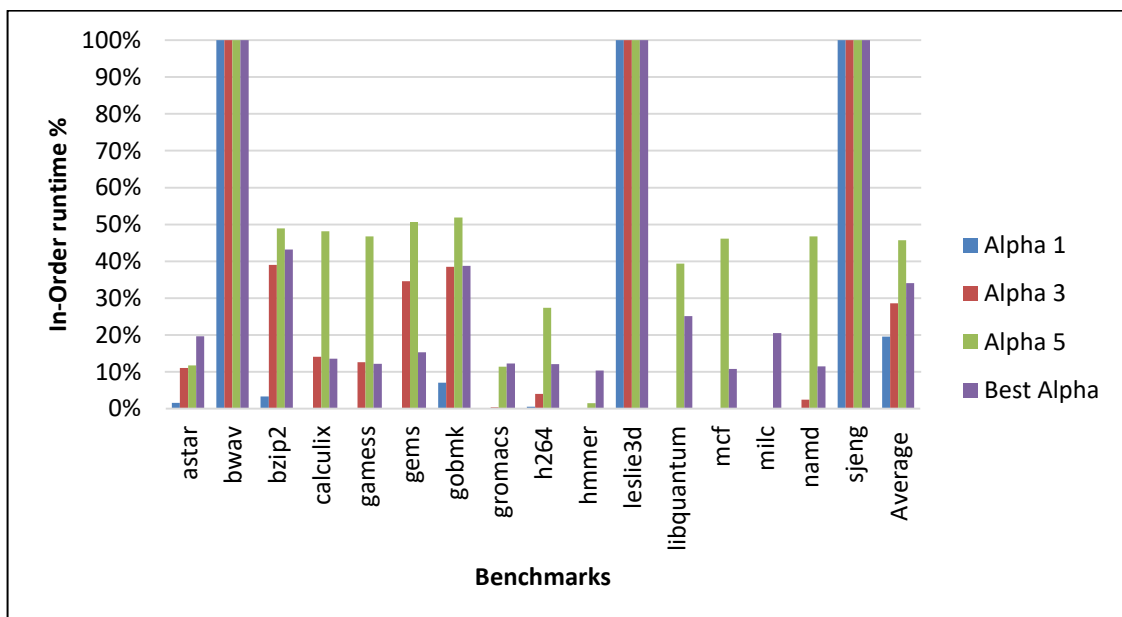


Figure 6.1. Runtime percentage of the In-Order mode in mode selection mechanism of ASMMP for various alpha thresholds

We also notice that as the threshold value increases and performance losses can be tolerated more, the In-Order mode is given a better chance. Each benchmark's final column depicts the effect of the near-optimal alpha threshold (best alpha) value retrieved from Table 6.3. As the Graph indicates, the results for the fixed alpha threshold of 3 and the best alpha threshold that always results in a performance penalty of around 5 percent are very similar. Undoubtedly, using the fixed threshold value of 3 prevents us from taking advantage of some power-saving opportunities in the astar, bzip2, gromacs, h264, hmmer, libquantum, mcf, milc, and namd benchmarks. On the contrary, in the gems benchmark, we provide more opportunities for power savings in the fixed threshold case and compensate for the significant performance impact, as illustrated in Figure 6.2. The degree of performance degradation for three fixed threshold values and the near-optimal threshold (best alpha) that we studied is depicted in Figure 6.2.

The obtained results showed a small loss of around 0.25 percent on average in system performance when using a small threshold value ( $\alpha = 1$ ), with the highest performance loss being 1.8 percent in the sjeng benchmark. On the other hand, because of this threshold value, many power-saving opportunities are lost.

Table 6.3. Near-optimal thresholds

Application	Alpha threshold ( $\alpha$ )	Application	Alpha threshold ( $\alpha$ )
astar	7,0	h264	4,0
bwaves	insensitive	hmmer	7,0
bzip2	5,0	leslie3d	insensitive
calculix	3,0	libquantum	4,5
games	3,0	mcf	3,7
gems	2,5	milc	35,5
gobmk	3,0	namd	3,6
gromacs	5,0	sjeng	insensitive

The performance loss tolerance of applications in the In-Order mode varies greatly, as shown in the chart. For example, the bzip2 application spends nearly half of its time in In-Order mode, but we only see a 0.12 percent performance drop compared to the baseline Out-of-

Order processor. Calculix suffers a significant performance loss with a fixed threshold of 3, despite only running in the In-Order mode for 14 percent of its total runtime.

Additionally, we noticed that some benchmarks such as milc and hmmer are almost insensitive to all used threshold values and prefer the Out-of-Order execution mode during their lifetime.

The proposed mode switching circuit in ASMMP managed to stay safe by choosing the Out-of-Order execution mode when the loss of performance will be high if we switch to the In-Order mode. The only exceptional case is encountered with the gems benchmark. The fixed alpha threshold of 3 seems to be a little too much for this benchmark (the best alpha value is 2.5, as shown in Table 6.3), and we observe more than a 15 percent performance drop. Finally, the mode switching circuit in ASMMP also identifies slow benchmarks that are insensitive to any alpha threshold (i.e., bwaves, leslie3d, and sjeng) and run them in In-Order mode 100 percent of the time.

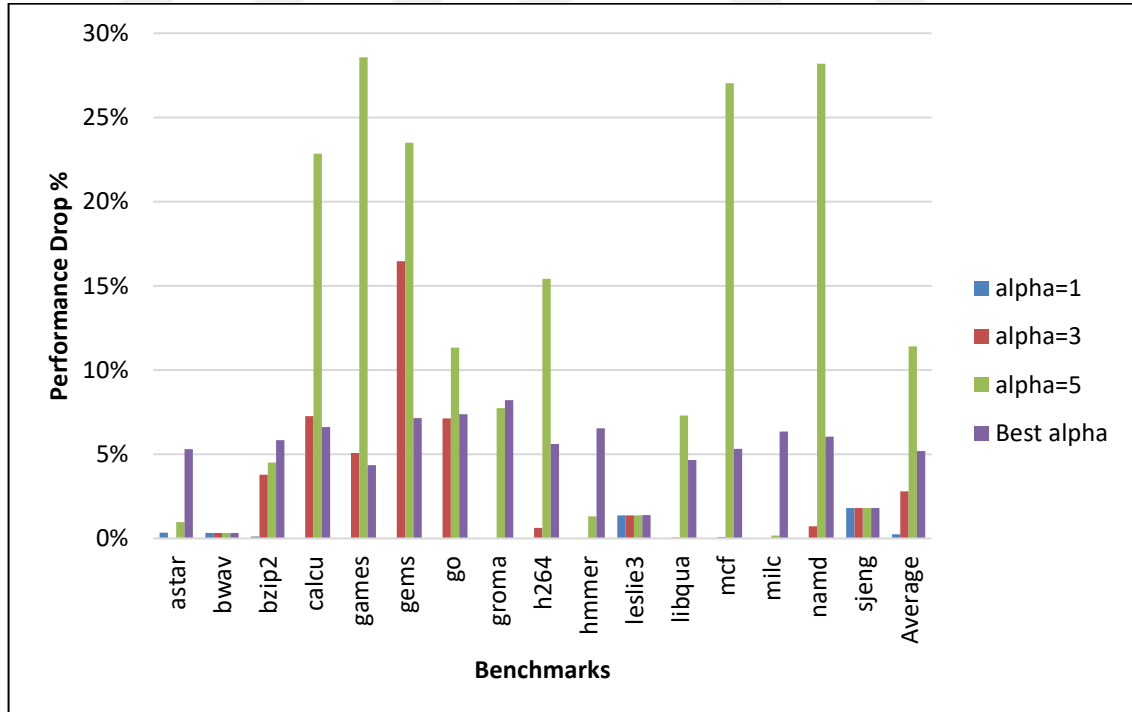


Figure 6.2. Percentage of performance drop in mode selection mechanism of ASMMP for various alpha thresholds

Additionally, we recorded the power savings, energy-delay-product (EDP), and energy-delay-square product (ED2P) savings associated with the mode switching mechanism

implemented in ASMMP. Figures 6.3, 6.4, and 6.5 show the obtained results for three different threshold values. As the Figures show, while the threshold value is 1, the average power and EDP savings are nearly 15 percent. However, when the threshold value is set to 3, the average power savings jump to 21.5 percent, while the average performance loss remains around 2.7 percent. The same observation can be made about the EDP criterion. When the threshold value increases from 1 to 3, the average EDP savings rise to 19.1 percent. It is worth noting that these results are very close to the near-optimal alpha thresholds that have been empirically determined for each benchmark. Of course, the average power and EDP savings are the greatest for the alpha threshold of 5. On the other hand, this threshold value comes with an 11.4 percent average performance penalty, which may be tolerable in battery-powered devices such as laptops and smartphones.

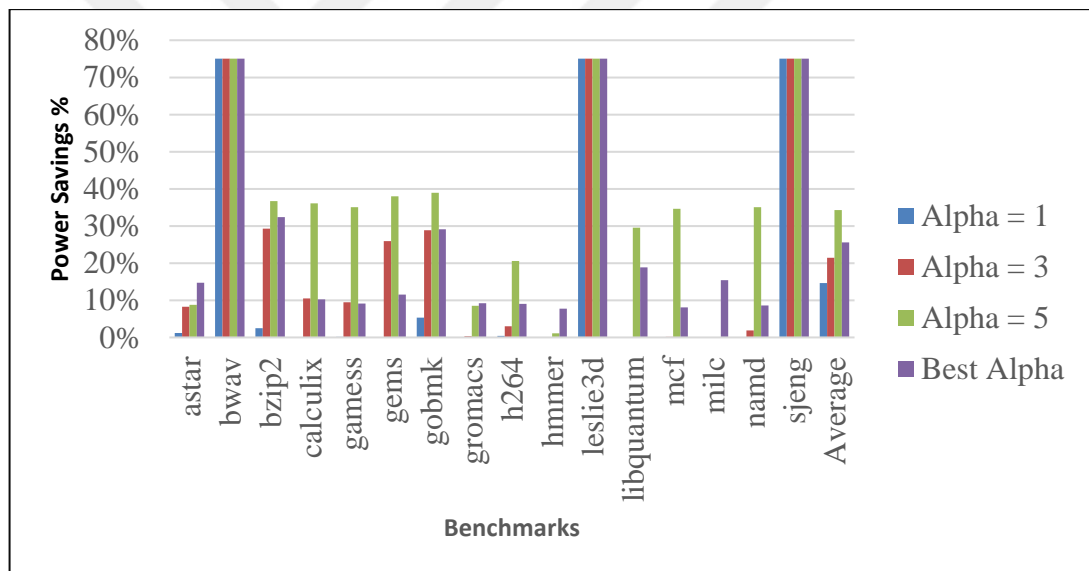


Figure 6.3. Percentage of power savings in mode selection mechanism of ASMMP for various alpha thresholds

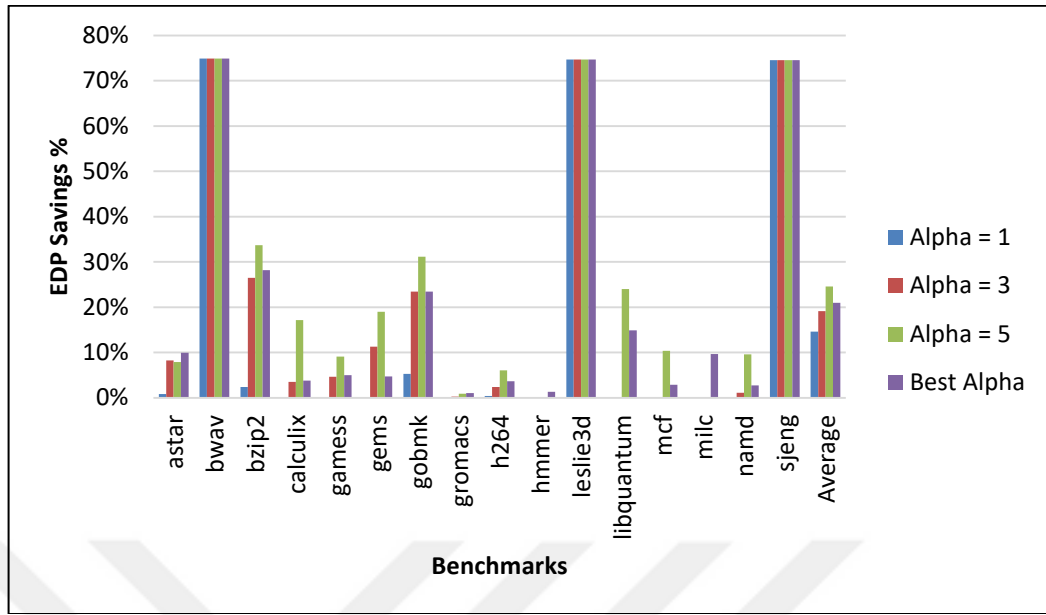


Figure 6.4. Percentage of EDP savings in mode selection mechanism of ASMMP for various alpha thresholds

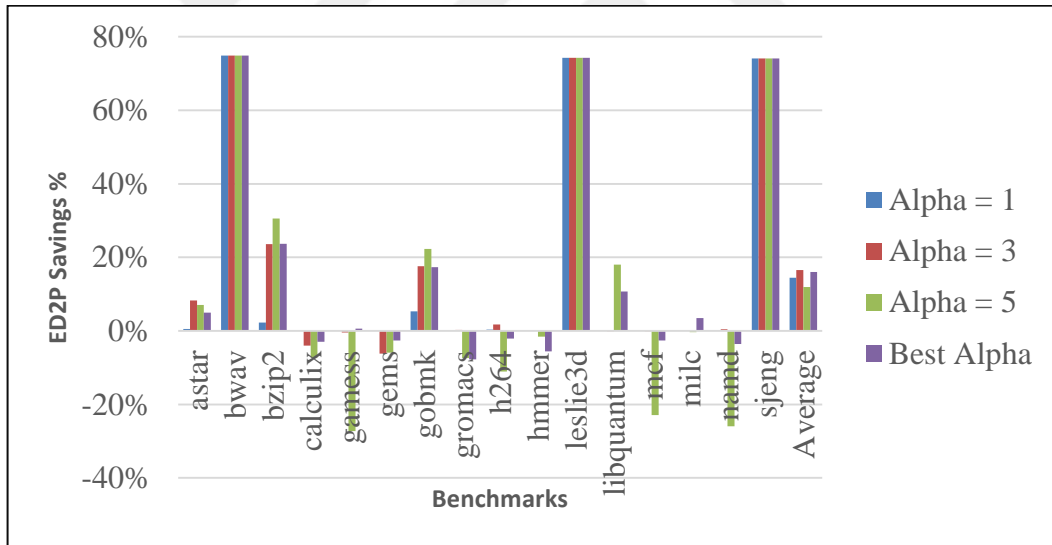


Figure 6.5. Percentage of  $ED^2P$  savings in mode selection mechanism of ASMMP for various alpha thresholds

## 6.2. RESOURCE PARTITIONING TESTS AND RESULTS

We run the proposed resource partitioning mechanism then record its effects on performance and consumed power. Also, collect average occupancy values for different processor structures using different upsize and downsize threshold values by running benchmarks from the Spec2006 benchmark suite using the Gem5 simulator with x86 ISA [2] [3]. Each

benchmark is executed for 100M instructions. Configuration parameters for the simulated processor are given in Table 6.4. Lastly, we will examine these results in terms of performance and achieve savings by turning off entries from different processor resources'.

Table 6.4. Specification of the simulated processor during resource partitioning

L1 I- and D-Caches	16Kb, 4-way, 64-byte line size, 2 cycle latency
L2 Cache	128Kb, 8-way, 64-byte line size, 20 cycle latency
CPU Frequency	2 GHz
Pipeline	4-way issue bandwidth ROB: 192 entry, 24 partitions IQ: 64 entry, 8 partitions LSQ: 32 entry, 4 partitions RF: 256 registers, 32 partitions
Decision period (1 epoch)	1 M cycles
No of occupancy samples collected every 1M cycles	256 samples
No of executed instructions	100 M
Instruction execution mode	Out-of-Order

### 6.2.1. Performance

As we expected from the obtained results, we observe an improvement in application performance compared to baseline with losing chances in saving power, after making the downsizing process harder to happen by choosing high downsize threshold values. We had more opportunity to save power after performing more frequent downsizing processes by choosing low downsize threshold values while a performance decline for all running benchmarks was noticed.

This degradation is related to performing resource partitioning very frequently, which results in continuous oscillation of resource size. Lastly, Figures 6.6 to 6.8 show the proposed resource partitioning results in terms of running application performance (IPC).

The obtained results in Figure 6.8 when compared to the non-partition baseline configuration, the proposed resource partitioning mechanism with a high downsize threshold value results in a 0.25 percent performance degradation. However, more than 48 percent performance degradation occurs when the resizing process is performed too frequently due to low threshold values.



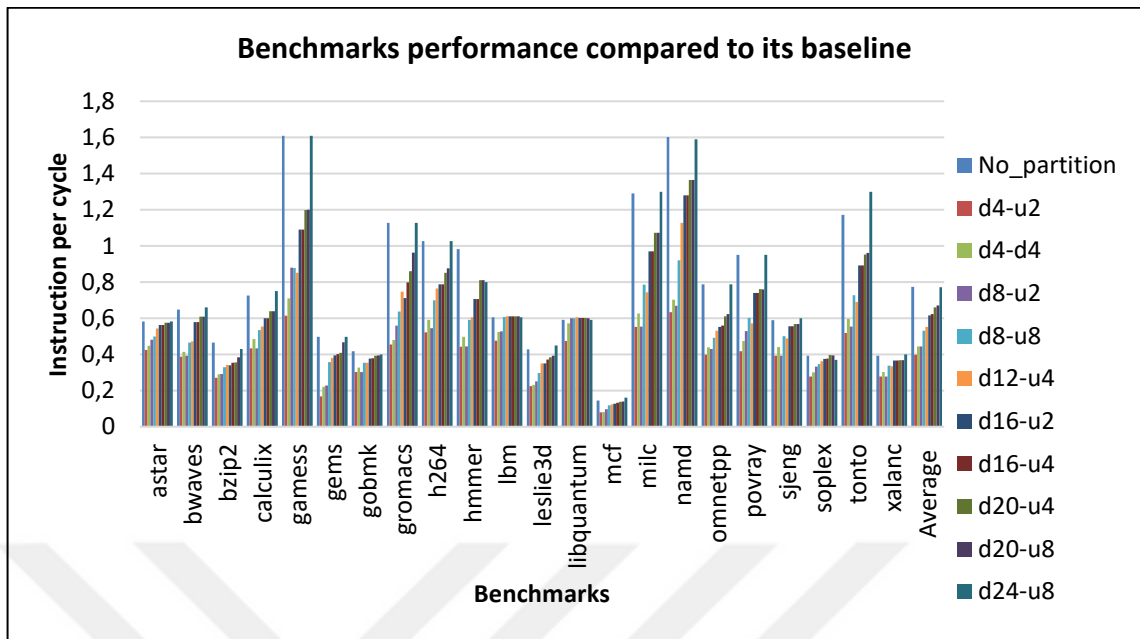


Figure 6.6. Performance of resource partition in ASMMMP compared to baseline Out-of-Order processor.

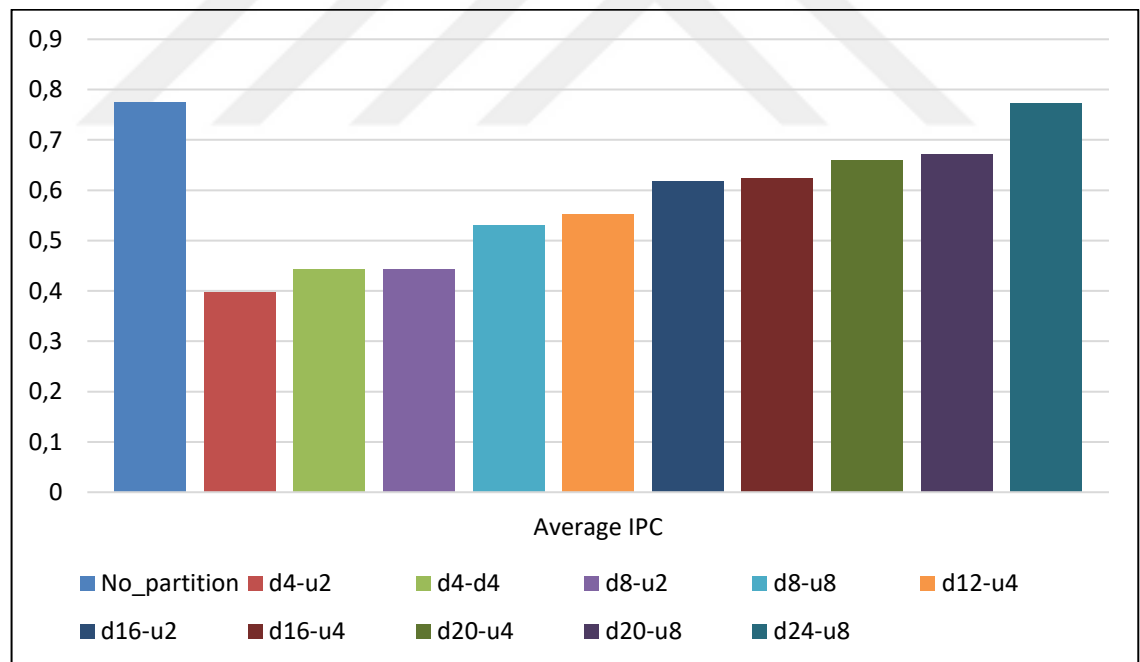


Figure 6.7. Average performance for simulated benchmarks with different threshold values

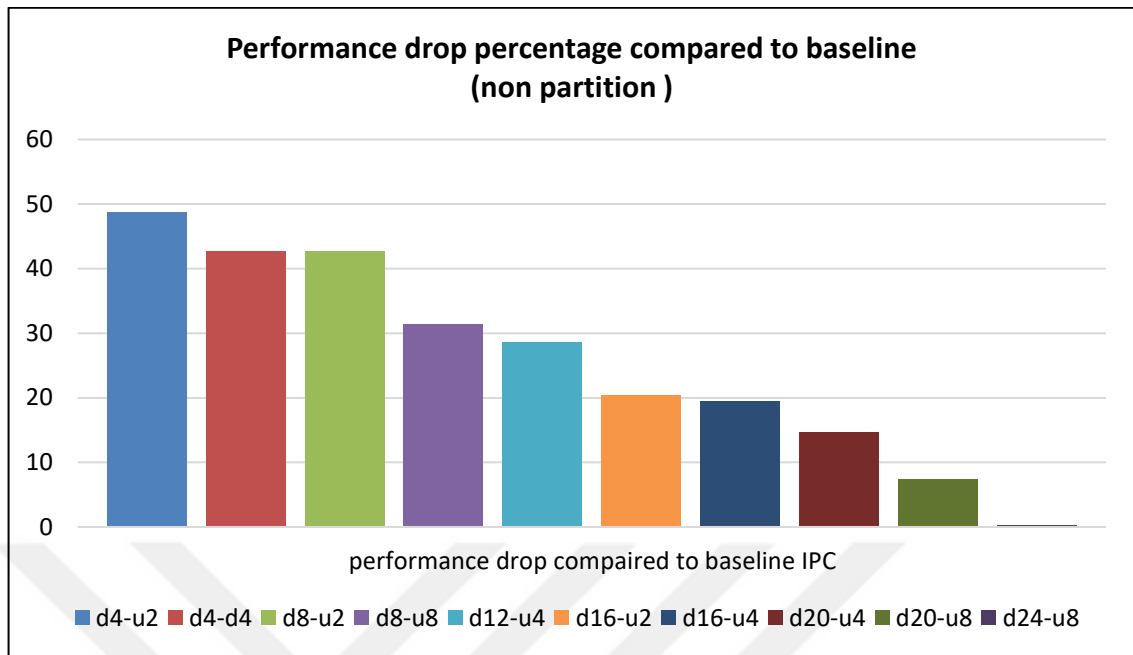


Figure 6.8. Average performance drop percentage for simulated benchmarks with different threshold values

### 6.2.2. Issue Queue

The proposed resource partition mechanism successes in turning off more than 53 percent of issue queue entries after using a high downsize threshold value. On the other hand, more than 84 percent of issue queue entries were saved when using low threshold values, but we will not use these low threshold values in ASMMP resource partitioning mechanism to prevent performance degradation of all running applications. Lastly, Figures 6.9 and 6.10 respectively show the used issue queues entries by Spec2006 benchmarks followed by the average used issue queue entries after applying resource scaling.

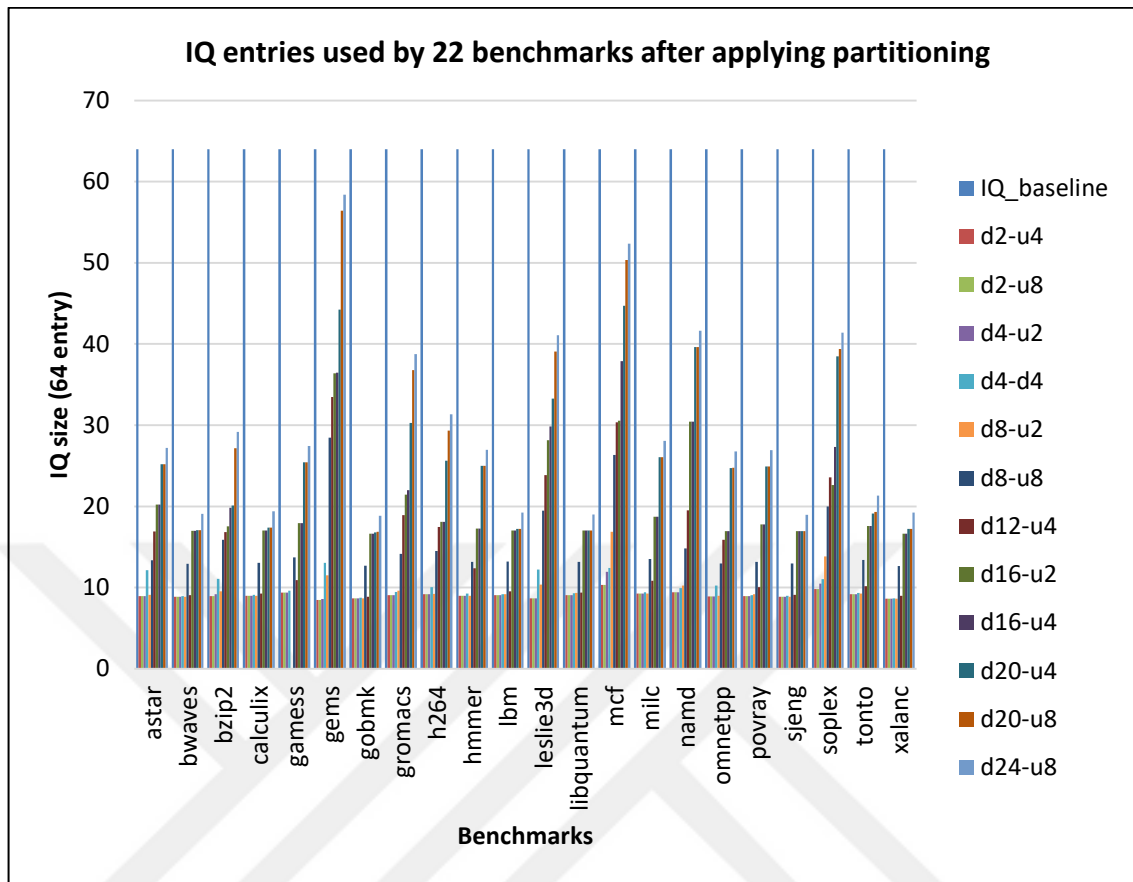


Figure 6.9. Instruction Queue entries used compared to baseline (64 entry no partition) configuration

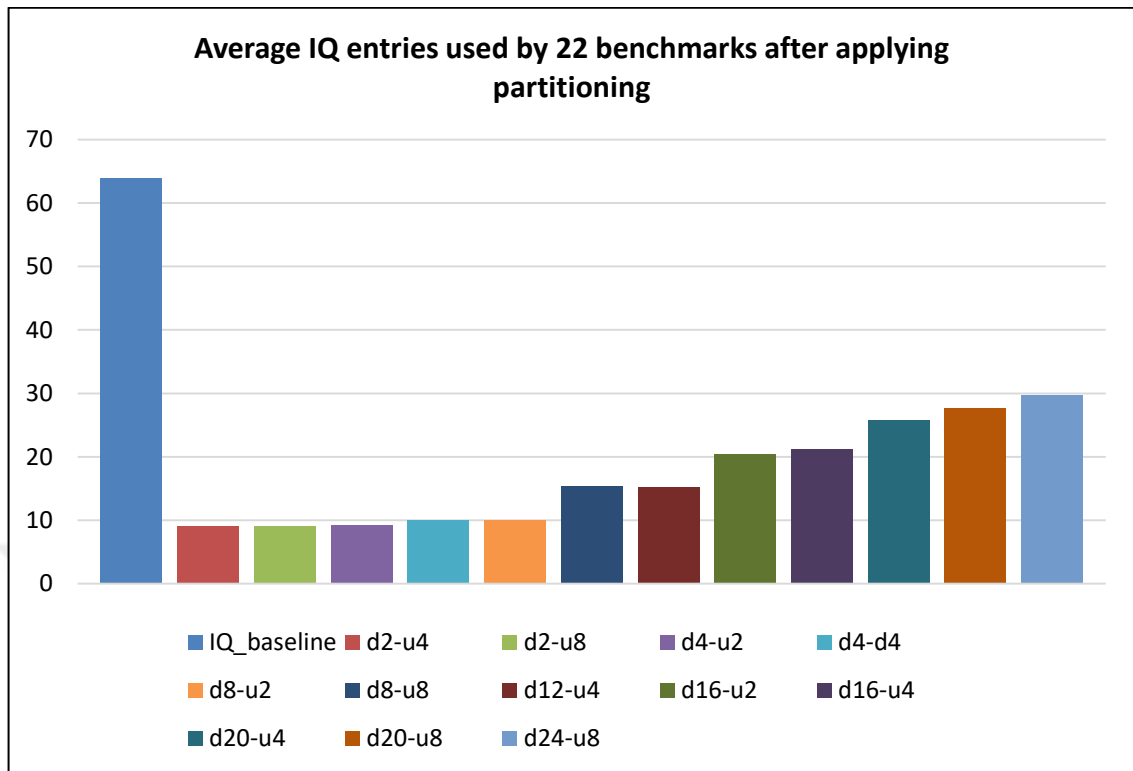


Figure 6.10. Average used instruction queue entries for all simulated benchmarks compared to baseline (64 entry) configuration after resource partitioning

### 6.2.3. Re-order Buffer

Our simulation results for the resource Re-order buffer were identical to those for the other resources. The effectiveness of the proposed resource partitioning mechanism is demonstrated in Figures 6.11 and 6.12. Whenever the downsize threshold value is 24, and the upsize threshold value is 8, an average saving of more than 63 percent is achieved. On the contrary, when a downsize threshold value of 2 and an upsize threshold value of 4 are used, enormous average savings of around 90 percent occur. Finally, as previously stated, maintaining a high downsize threshold value will benefit our design by achieving significant average savings and limiting performance degradation to less than 0.25 percent.

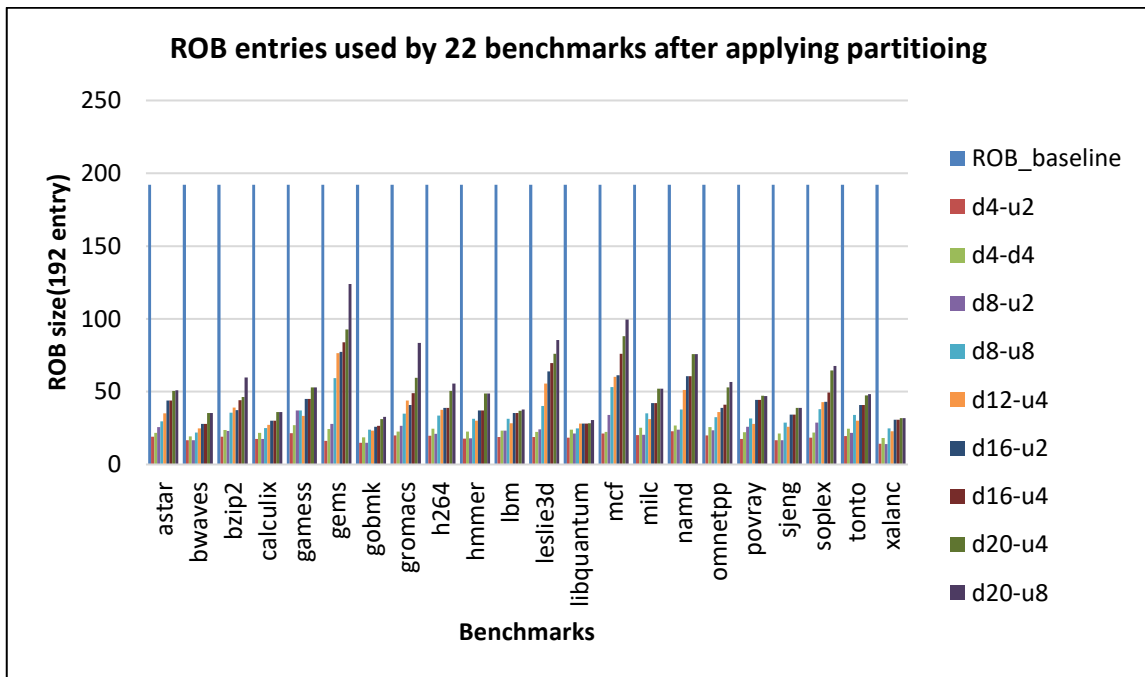


Figure 6.11. Re-order buffer entries used compared to baseline (192 entry no partition) configuration.

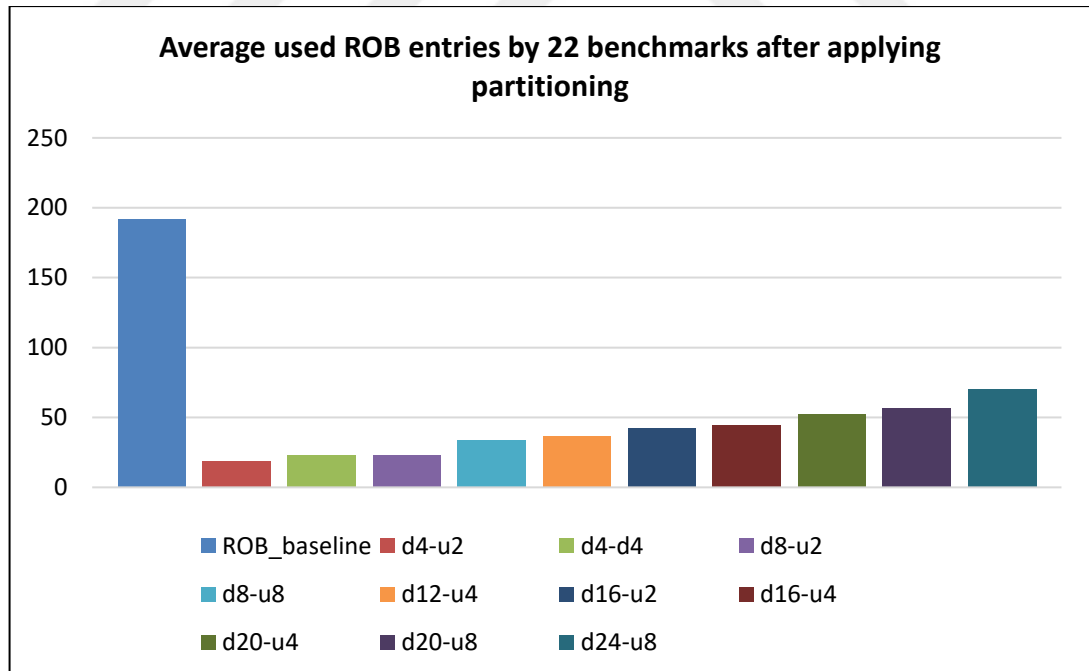


Figure 6.12. Average used Re-order buffer entries for all simulated benchmarks compared to baseline (192 entry) configuration after resource partitioning.

#### 6.2.4. Load and Store Queues

Figures 6.13–6.16 illustrate the results of resource partitioning when applied to both load and store queues. their size was mostly emphasized in both queues. Both resources contain 32 entries, which is a small number in comparison to other ASMMP resources. We saved approximately 25 percent of the store and load queue entries using a downsize threshold of 24, and an upsize threshold of 8. Additionally, as illustrated in Figure 6.13, no savings in the store queue were achieved when using high downsize threshold values for some benchmarks such as libquantum, lbm, and bwave. Finally, as illustrated in Figure 6.15, our resource partitioning mechanism could only disable two load queue entries in several benchmarks, including leslie3d and mcf.

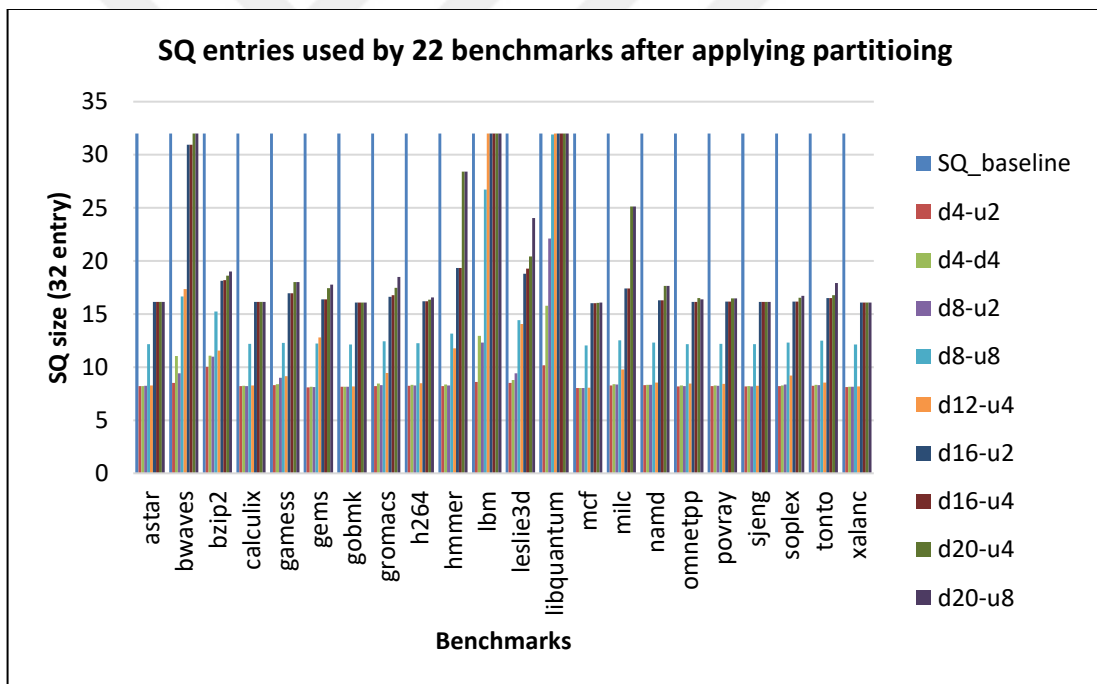


Figure 6.13. Store queue entries used compared to baseline (32 entry no partition) configuration.

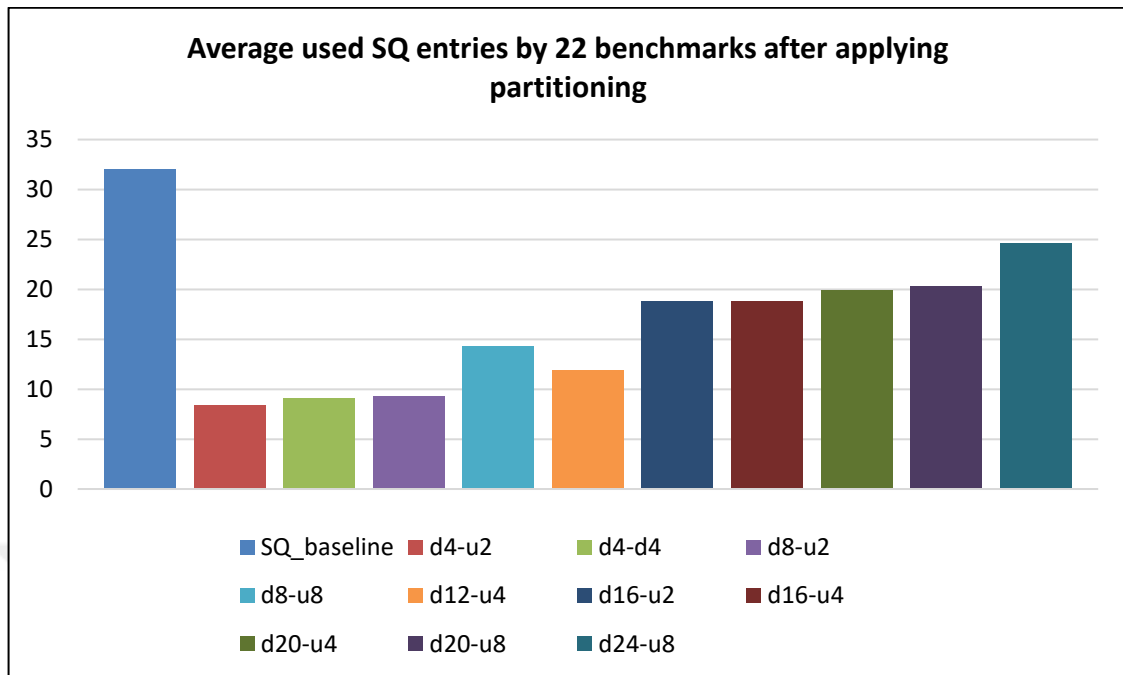


Figure 6.14. Average used store queue entries for all simulated benchmarks compared to baseline (32 entry) configuration after resource partitioning.

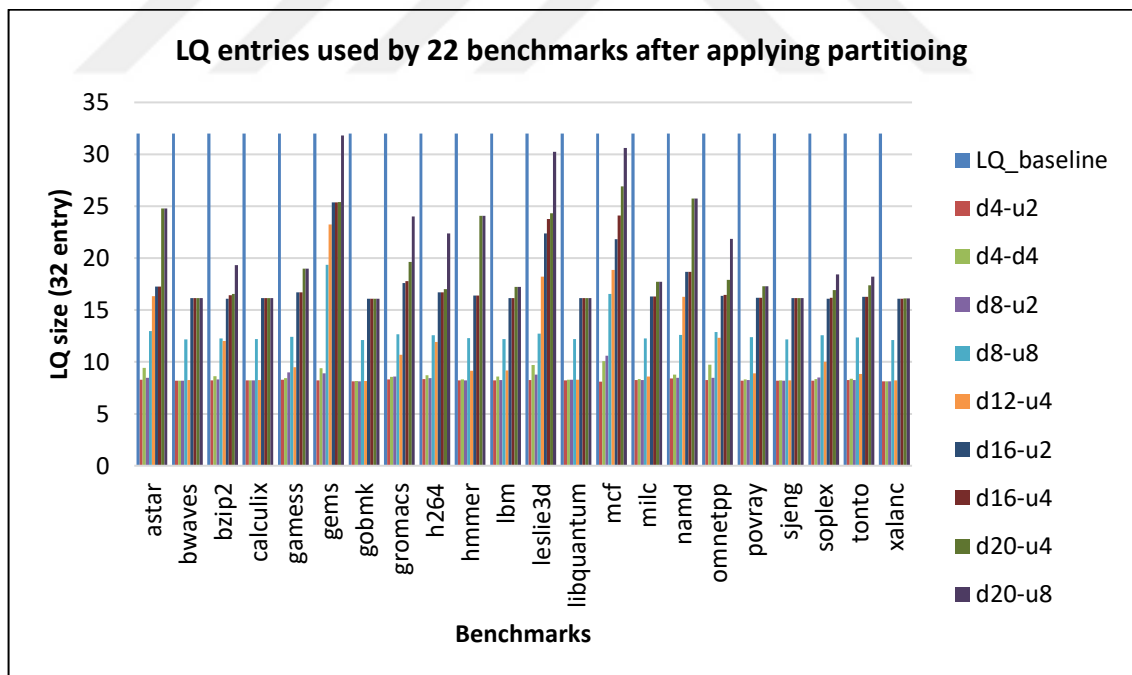


Figure 6.15. Load queue entries used compared to baseline (32 entry no partition) configuration.

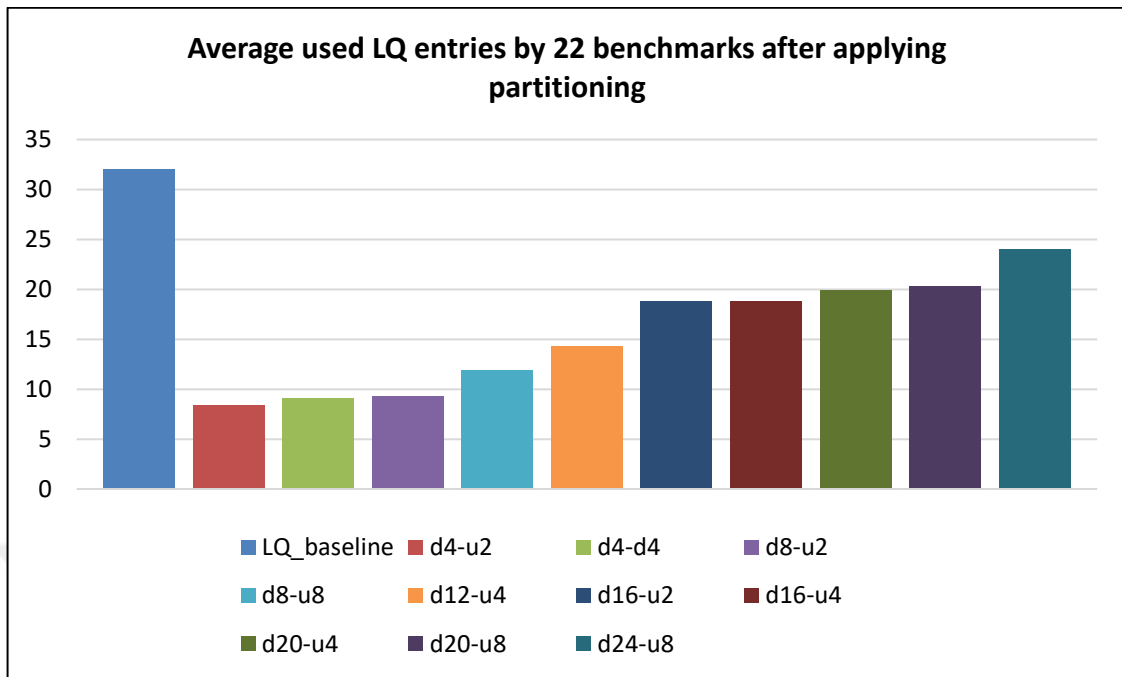


Figure 6.16. Average used load queue entries for all simulated benchmarks compared to baseline (32 entry) configuration after resource partitioning.

### 6.2.5. Register File

Figures 6.17–6.20 illustrate the resource partitioning simulation results when applied to both integer and floating-point register files. For downsize threshold value 24 and upsize threshold value 8, the proposed partitioning mechanism succeeded in saving more than 57 percent of integer register file entries. Additionally, savings in the floating-point register file exceeded 58 percent.

Finally, Figure 6.21 illustrates savings percentages across all ASMMP resources. Additionally, we can conclude that our proposed resource partitioning mechanism successfully allocated sufficient resource entries to each running application to maintain performance close to the baseline case while reducing power consumption by turning off unused resource entries.



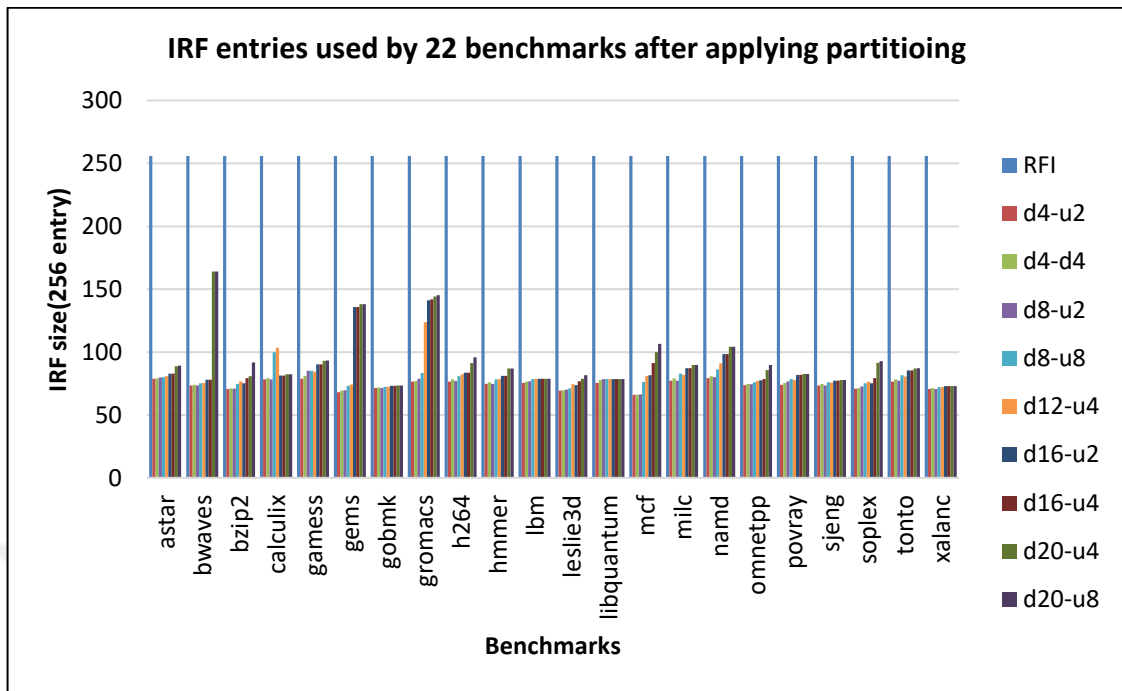


Figure 6.17. Integer register file entries used compared to baseline (256 entry no partition) configuration.

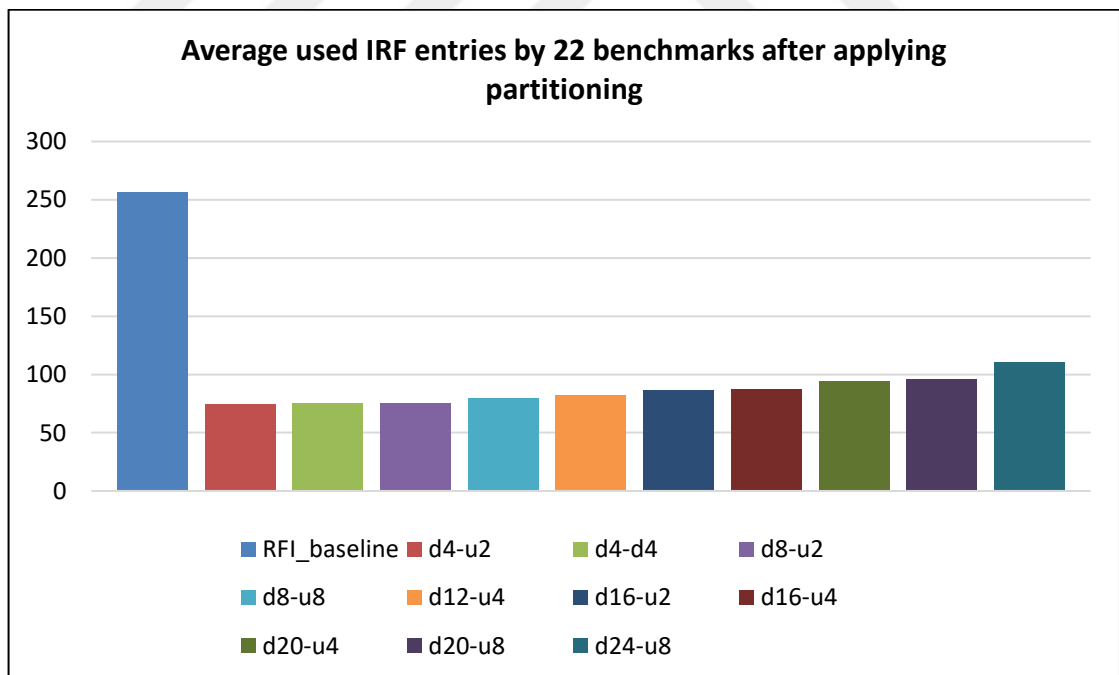


Figure 6.18. Average used integer register file entries for all simulated benchmarks compared to baseline (256 entry) configuration after resource partitioning.

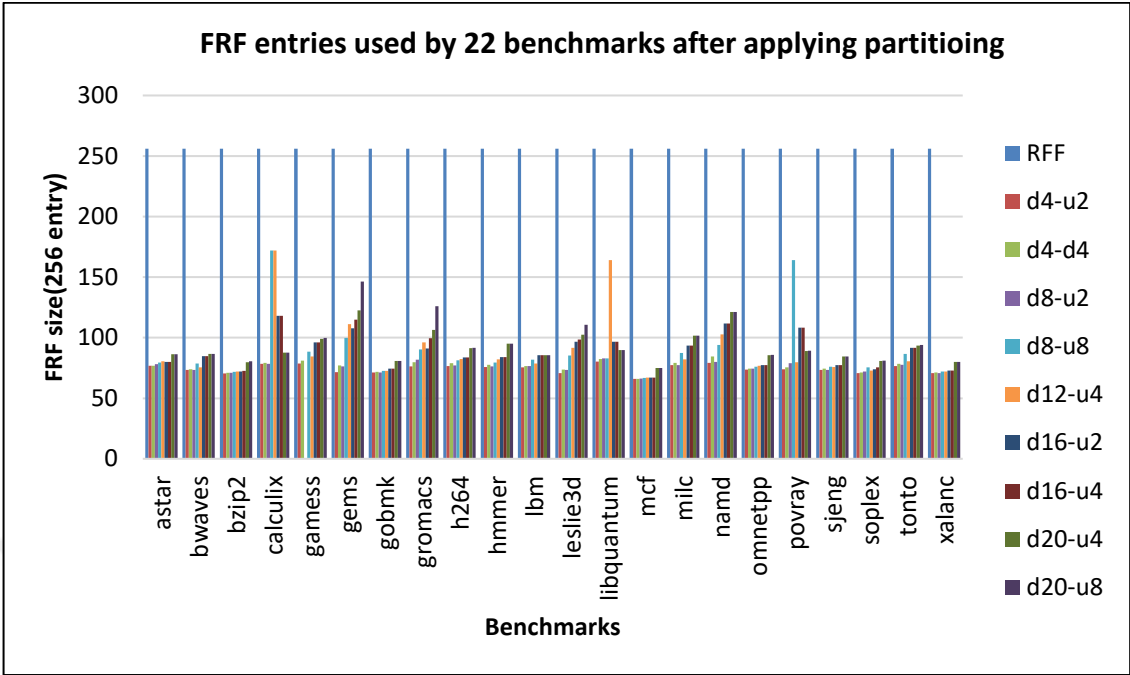


Figure 6.19. Floating-point register file entries used compared to baseline (256 entry no partition) configuration.

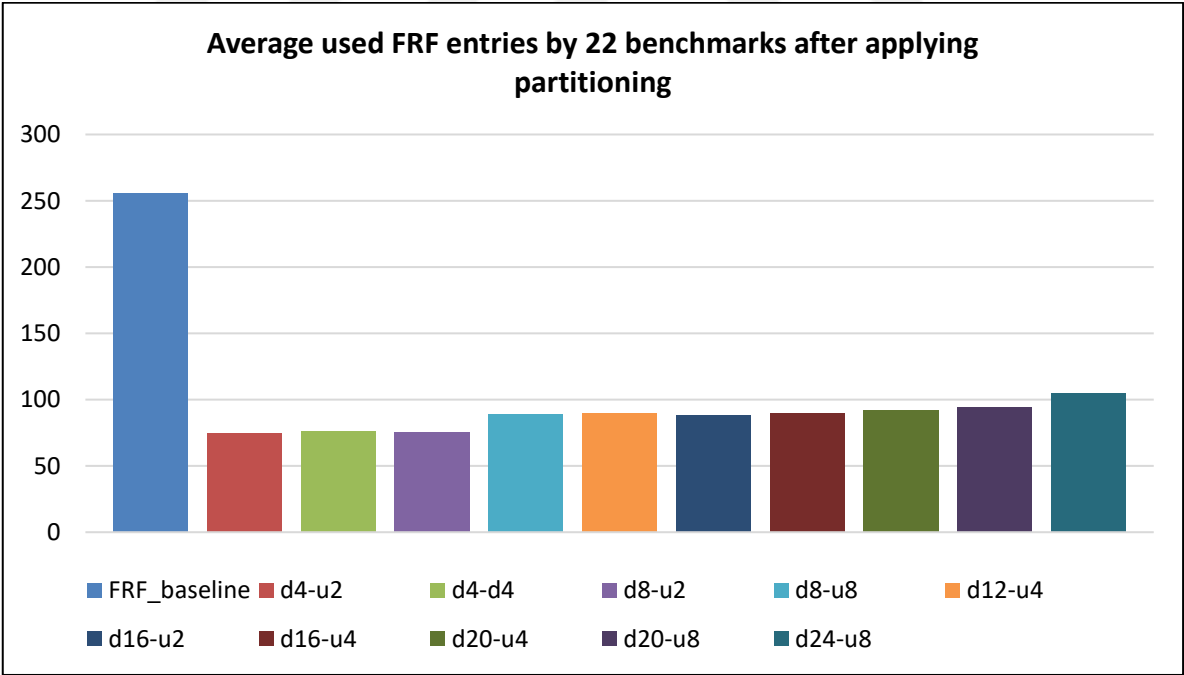


Figure 6.20. Average used floating-point register file entries for all simulated benchmarks compared to baseline (256 entry) configuration after resource partitioning.

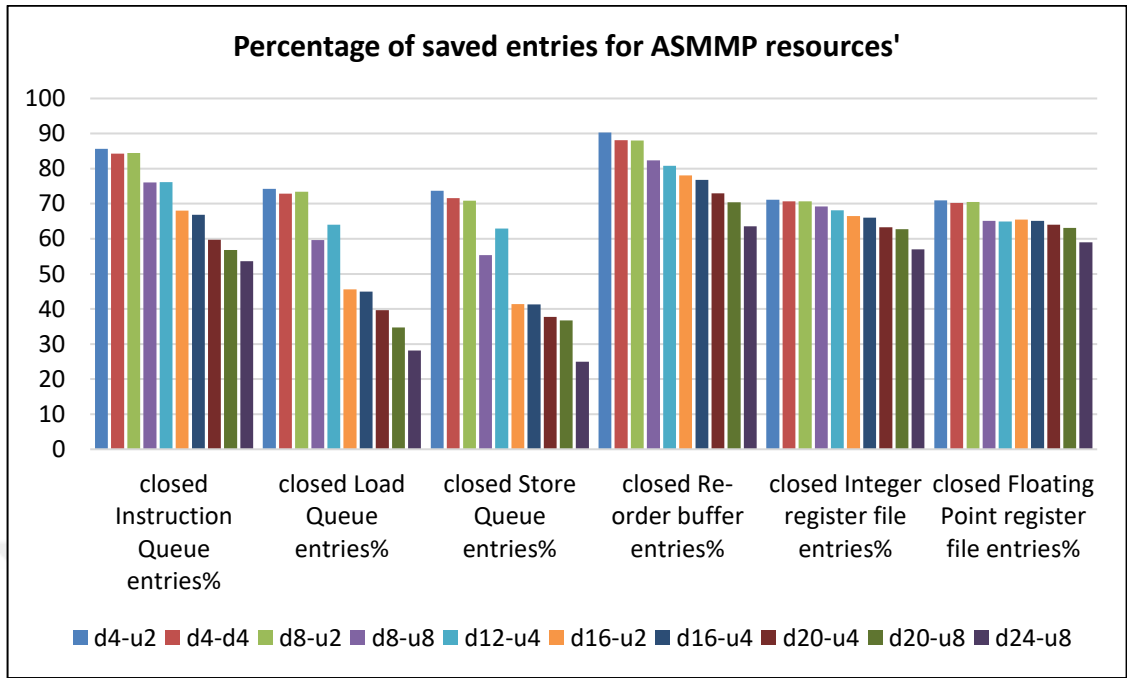


Figure 6.21. Percentage of saved entries after applying partition in different ASMMMP resources

### 6.3. FINAL ASMMMP TESTS AND RESULTS

In this subsection, we will go through the used experimental methodology, then evaluate and discuss our final ASMMMP architecture design results after combining both execution mode switching and resource partitioning mechanism in terms of performance, power savings, EDP, and  $ED^2P$  metrics.

#### 6.3.1. Experimental methodology

Like our previous design evaluation process, we continue using the Gem5 simulator to test and evaluate the overall ASMMMP model [2]. The experimental methodology used in evaluating resource partitioning and mode selection designs was also repeated here by running the SPEC CPU2006 benchmarks for 100 million instructions on x86 architecture [3]. The used simulation parameters of ASMMMP are given in Table 6.5.

Table 6.5. Specification of simulated ASMMP architecture

Microprocessor architecture	x86
Processor frequency	2GHz
Machine width	4
Re-order buffer size	192 entries
Issue queue size	64 entries
Load and Store queue size	32 entries
Physical register file size	256 entries
L1 instruction and data cache	4-way LRU, 16 KB
L2 cache	8-way LRU, 128 KB
Execution switch decision period (ESDP)	10000 cycles
Sampling period (SP)	100 cycles
Partition decision period (PDP)	2M cycles
Occupancy sampling period (OSP)	8000 cycles (250 samples)
Alpha threshold values	1,3,5
Downsize threshold value	40
Upsize threshold value	10
Resource scale size	1 partition (resource size/8)

Performance, power savings, and energy efficiency are all criteria used to evaluate simulation results in this setting. Furthermore, it has been assumed that the processor consumes the same energy per unit of time as it did in the CC investigation. It was assumed that a microprocessor operating in Out-of-Order execution mode would consume four times as much as the amount of energy as a microprocessor operating in In-Order execution mode. Using Equation 6.6, the power expenditure (PE) can be calculated by taking the percentage of time spent on the In-Order processor (W). Additionally, we add a parameter for the number of saved entries (SRE) for all ASMMP resources in Out-of-Order execution mode. This value is calculated based on the number of unnecessary entries closed by the resource partitioning mechanism in ASMMP. To make things even better, we continue to use Equation 6.2 to calculate the power savings (PS). In addition, we continue to use Equation 6.3, which provides the formula for the slowdown rate (SR) of the proposed ASMMP architecture compared to the baseline Out-of-Order processor in terms of the average instructions per cycle (IPC) metric, as previously stated.

Lastly, there was no modification in Equations 6.4 and 6.5, used to measure efficiency metrics represented by the energy-delay product (EDP) and energy-delay-square product (ED2P) formulas, respectively.

$$PE = \frac{1}{4} * W + (1 - W) * (1 - SRE) \quad (6.6)$$

### 6.3.2. Resource Partitioning Effects

In the first phase of our tests, we wanted to determine how much performance degradation occurred when we used ASMMP's resource partitioning mechanism with Out-of-Order execution. The results of resource scaling and the percentage of entries that can be disabled using a downsize threshold of 40, and an upsize threshold of 10 for the Spec2006 benchmarks in ASMMP are shown in Figure 6.22. Additionally, alpha threshold values of 1, 3, and 5 were used to switch the execution mode from Out-of-Order to In-Order.

In general, lower-alpha values result in more Out-of-Order execution mode periods, and resource scaling or resource partitioning can result in higher savings. Large alpha values imply longer periods in order execution mode while saving power is derived from avoiding Out-of-Order complex structures.

We achieved power savings by shutting down approximately 30 percent of ASMMP resources during Out-of-Order execution mode periods with an alpha value of one and a performance loss of approximately 1.87 percent. In comparison to alpha value one, we continue to save less power. Additionally, 15 percent of entries are saved when the alpha value is three, resulting in a 2.96 percent performance degradation. When the alpha value is set to five, approximately 10 percent of entries are saved at a performance cost of more than 12 percent. These results were expected given the decline in Out-of-Order execution mode periods and the fact that ASMMP operates in an In-Order execution mode for most of the time (choosing high alpha values makes our execution mode switching mechanism harder to switch from In-Order to Out-of-Order).

Also, no entries were saved from the resource partitioning mechanism in *bwave*, *leslie3d*, and *sjeng* since no performance difference between both execution modes and our execution mode switching mechanism is chosen in order execution mode for these benchmarks during their lifetime. Whereas, near 40 percent savings were achieved for *milc* and *gems* benchmarks because they are not sensitive to any alpha threshold value, and they keep running in Out-of-Order execution mode.

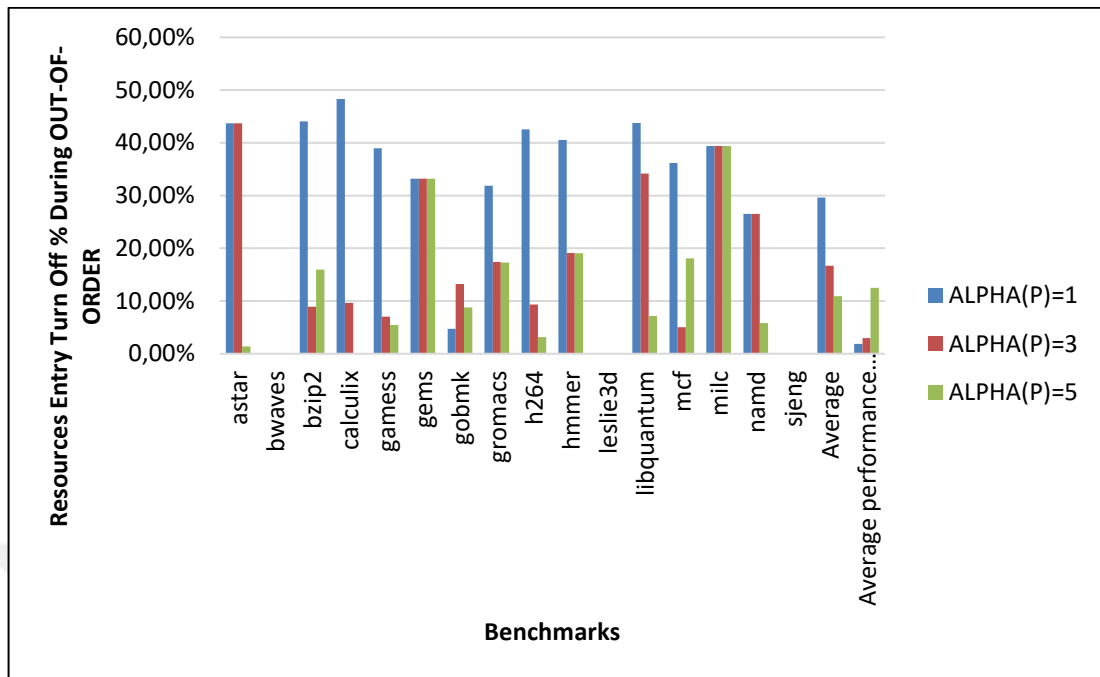


Figure 6.22. Percentage of saved entries in final ASMMP for Spec2006 benchmarks during Out-of-Order execution mode

Figure 6.23 shows the percentage of entries saved for different ASMMP resources during the benchmark lifetime. The savings in the Re-order buffer and register file come mainly from the resource partitioning mechanism when the alpha value is one, where all benchmarks run most of the time in Out-of-Order execution mode. Thus, our partitioning algorithm assigns enough entries from both structures based on the running application's needs and turns off the unused entries to save power.

The main savings in load, store and issue queue are achieved when applications start switching to an In-Order execution mode and stay in several periods in the same execution mode (alpha 3,5). In this case, as we mentioned before, saving comes not from resource scaling but comes from not intensive use of Out-of-Order complex structures by applying direct resource downsize to one partition, as we showed in Figure 5.5.

Additionally, the chosen LSQ (32 entry) and IQ (64 entry) sizes were insufficient for the running applications during Out-of-Order execution mode (alpha 1), resulting in the loss of the opportunity to save power in these structures, and the partition mechanism did not perform resource downsizing to avoid performance degradation of the running application. On the contrary, alpha 3 and 5 increase the likelihood of saving additional LSQ entries due

to declining average occupancy at the start of the new Out-of-Order execution period. When the In-Order runtime increases, the resource occupancy for several periods will be limited to a single partition. When the application is switched to Out-of-Order, the LSQ occupancy equals the resource's full size, which is 32 entries. At the ending of PDP, ASMMP's resource partition mechanism will perform one upsizing operation by eight entries no more than once or twice and then revert to In-Order execution mode. Finally, we can say that when there is switching from In-Order to Out-of-Order, more upsizing operations occur for small resource sizes, and resource downsizing occurs for large resource sizes. In this case, selecting an alpha threshold value of 3 results in the average savings of 27 percent in ASMMP resources and performance degradation of less than 5 percent, or approximately 2.96 percent. Finally, this value enables efficient and balanced utilization of ASMMP's resource partitioning and mode switching mechanisms. Otherwise, selecting alpha one will result in the loss of the mode switching mechanism, while selecting alpha 5 will have the opposite effect, reducing the impact of resource partitioning on ASMMP with a performance degradation of more than 5 percent of the running application.

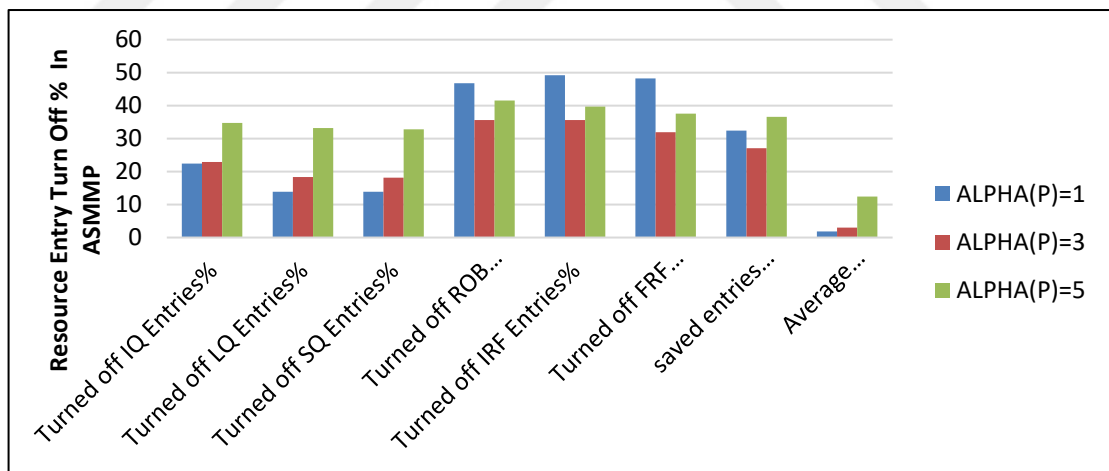


Figure 6.23. Average saving percentages for different ASMMP resources includes both out-of-order and In-Order execution mode

### 6.3.3. Performance

Figure 6.24 shows our final ASMMP architecture performance with all its mechanisms, including resource partitioning and mode switching. A performance degradation by 1,92

percent and 2.96 percent, respectively, when choosing alpha values of one and three. Besides, performance degradation is more than 12.47 percent when alpha value 5 is chosen.

When alpha one is chosen, we observe a 7.5 percent performance degradation in bzip2. This is related to the accuracy of the average occupancy samples collected at the end of the partition decision period (250 samples in our case). Our partitioning mechanism reaches the more accurate needs of the running application after increasing the number of samples (bzip2). The main goal of using 250 samples for two million cycles is to reduce hardware complexity.

Figure 6.25 depicts the percentage of In-Order execution modes for the Spec2006 benchmarks. The results in Figures 6.24 and 6.25 are consistent with our expectations, as more In-Order periods have a greater negative impact on performance and higher power savings (the case with the gobmek benchmark). However, the primary objective of this thesis was to conserve power and maintain a performance loss of less than 5 percent.

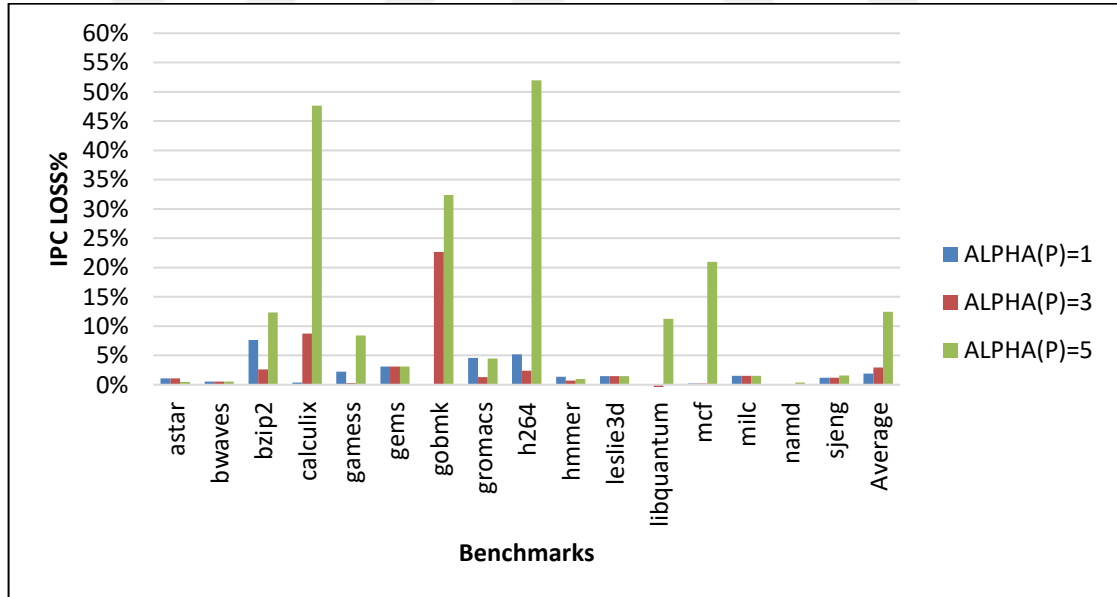


Figure 6.24. Percentage of performance drop of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds



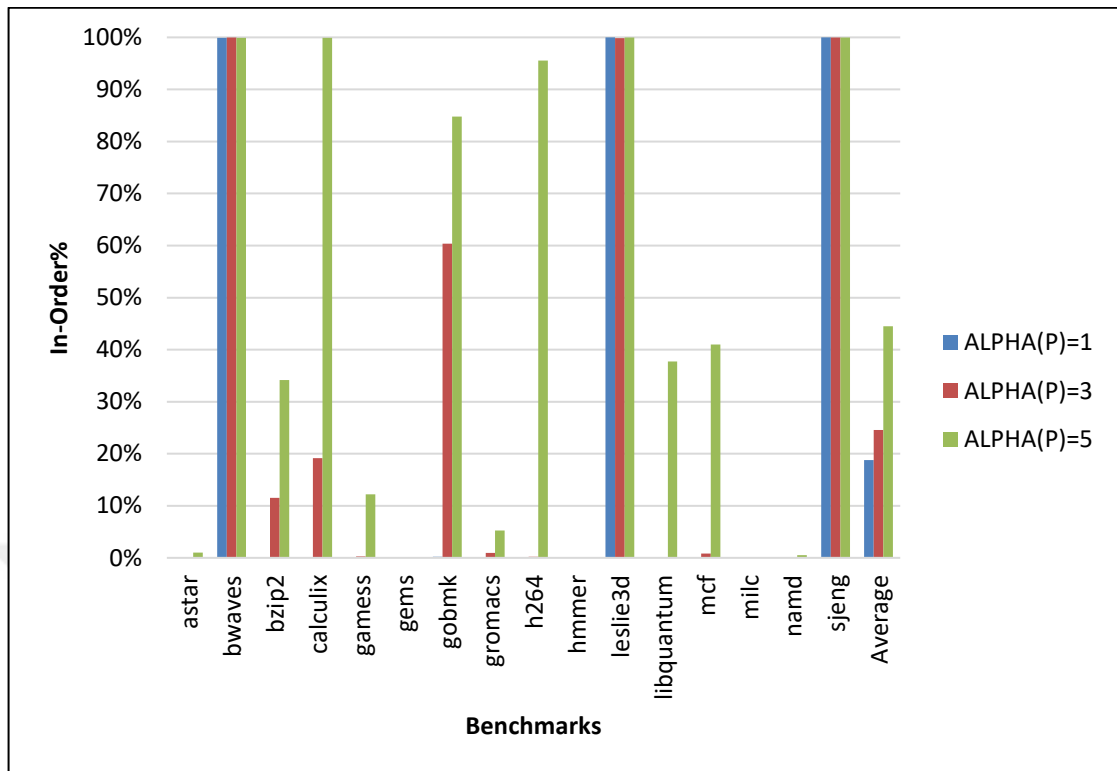


Figure 6.25. Runtime percentage of the In-Order mode in final ASMMP for various alpha thresholds

Finally, Figure 6.26 shows the overhead of adding the implementation of resource partitioning on performance in ASMMP when compared to the non-resource partitioning version (only mode switching mechanism). There is negligible impact on performance around 1.9 percent at maximum when alpha 1 is chosen. This threshold value makes our resource partitioning mechanism active for a long time since most of the execution mode will be in Out-of-Order mode, and continual resource scaling will be performed on resources of ASMMP (depending on chosen PDP). On the other hand, lower use of resource partitioning mechanism, which is the case for alpha value 5 reduces this impact and results only in 1 percent decline in performance compared to the non-partition version.

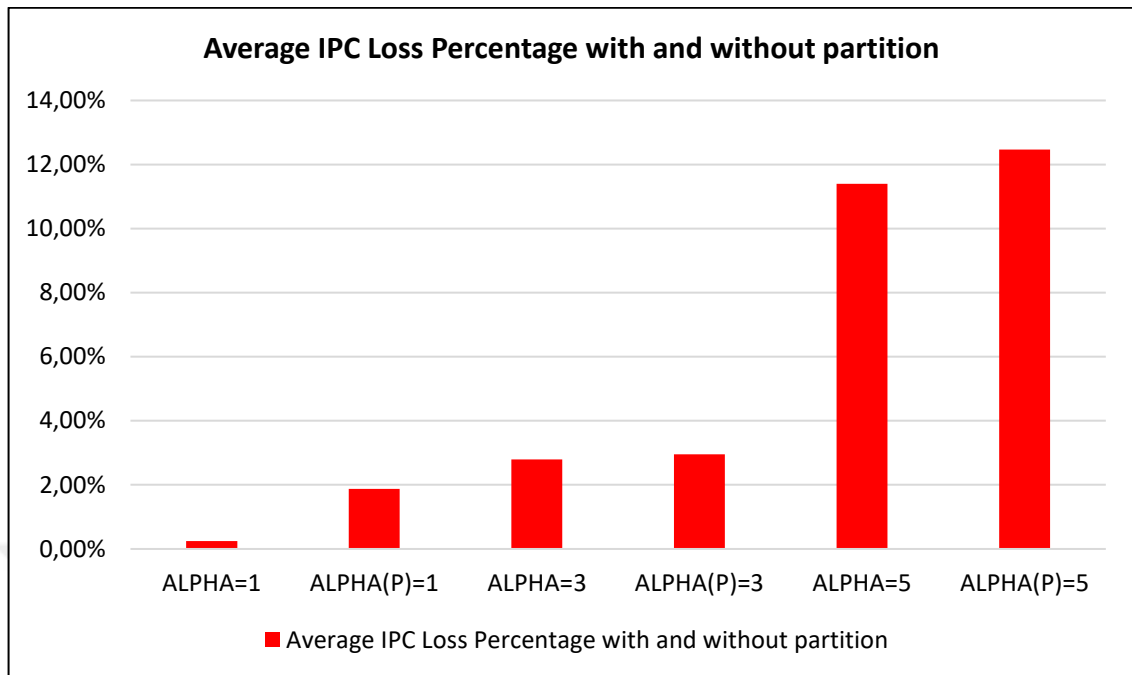


Figure 6.26. Percentage of performance drop of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds

The idle configuration again here is to choose alpha value one or three. Choosing threshold value three results only in a 0.10 percent performance drop when compared to the non-partition version. As we mentioned before, this threshold value helps gives opportunity to both mechanisms for functioning in balancing and proper way.

Finally, as shown in Figure 6.27 for alpha threshold values one and three, our ASMMP keeps In-Order execution mode activated for 18.5 percent and 24.57 percent, respectively, from the total runtime due to low application dispatch ratio. These runtime percentages allow saving more power by downsizing Out-of-Order structures to one partition and deactivate resource scaling. On the contrary, ASMMP activates only resource partitioning mechanism for 81 percent and 75 percent, respectively ( high dispatch ratio) from the total runtime while deactivating the switching mechanism to continue saving less power and avoid performance degradation.

#### 6.3.4. In-Order Runtime

The incorporation of resource partitioning into ASMMP maintained application performance close to the baseline case (full Out-of-Order no partition) and had no effect on

the percentage of In-Order runtime for different alpha threshold values because the switching decision mechanism is completely dependent on the application speed-up factor, which monitors instruction dispatch ratio, and instruction commit over fetch ratio. As illustrated in Figure 6.27, the difference in In-Order runtime percentage between the final ASMMP and the non-partitioned version of ASMMP is negligible.

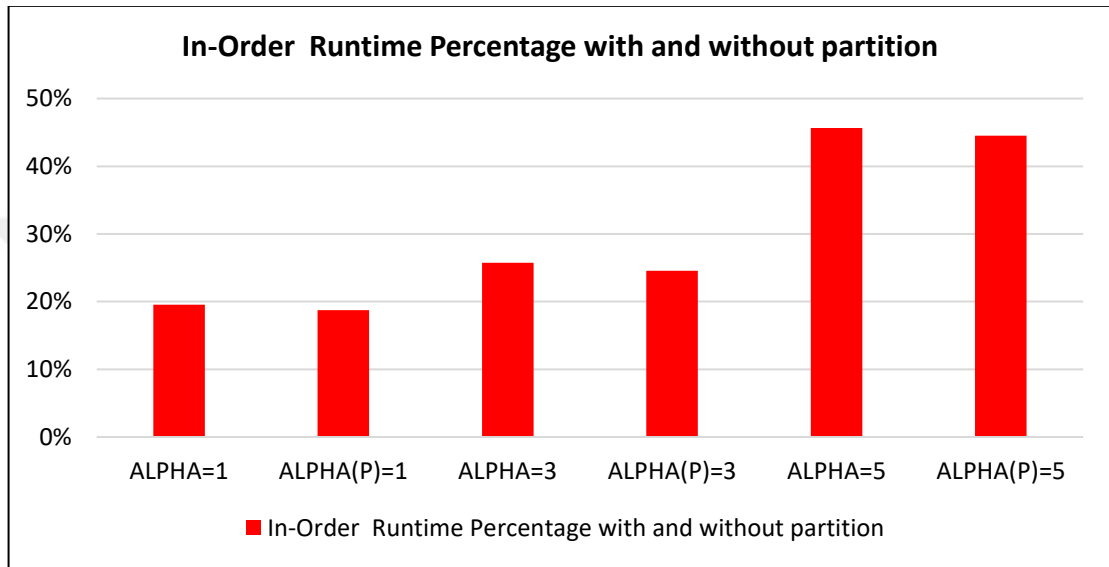


Figure 6.27. Runtime percentage of the In-Order mode in ASMMP for both partition and non-partition versions for various alpha thresholds

### 6.3.5. Power Savings

The percentage of total processor energy consumed by its resources is usually around 55 percent [10]. Figure 6.28 summarizes the average power saving percentage observed for various alpha threshold values of the Spec2006 benchmarks in the final ASSMP architecture.

In this case, an additional power saving estimation of more than 28 percent, 13 percent, and 8 percent can be achieved by using a resource scaling mechanism and total savings of 43 percent, 34 percent, and 42.58 percent respectively achieved in the final ASMMP design with keeping performance degradation below 5 percent for most Spec2006 benchmarks when alpha threshold values of one, and three are chosen, and performance loss of 12.5 percent is noticed when alpha value 5 is chosen.

Besides, in this Figure, we notice a high power saving of around 75 percent in three benchmarks (bwaves, leslie3d, and sjeng ). The reason for this saving relates to the mode decision mechanism in ASMMP, which identifies these slow benchmarks (low IDR, CFR) and, regardless of any alpha threshold value, In-Order execution mode is always chosen to run these benchmarks. Furthermore, we realize that 75 percent of the savings is because the In-Order execution mode consumes only 25 percent of the Out-of-Order mode power.

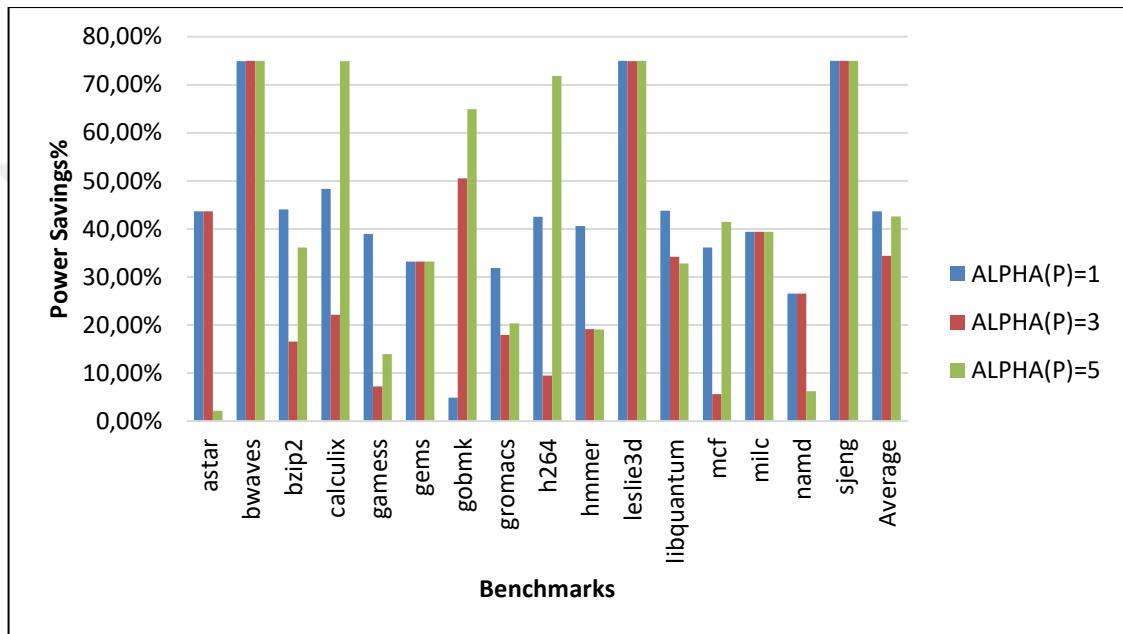


Figure 6.28. Percentage of power savings in final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds.

As illustrated in Figure 6.29, the final ASMMP design achieved greater power savings for different alpha threshold values. The Figure clearly illustrates the beneficial effect of resource scaling on ASMMP power consumption (by turning off unused entries during Out-of-Order execution mode) compared to the non-scaling version.

Finally, the least amount of power was saved by resource scaling mechanisms when a high alpha value was used (5). A higher alpha value indicates more In-Order periods and less resource partition mechanism activation. Thus, the execution mode decision mechanism results in additional power savings. For the above reasons, we could not achieve additional power savings through resource scaling, and the resulting savings remained consistent with the non-partition version of ASMMP.

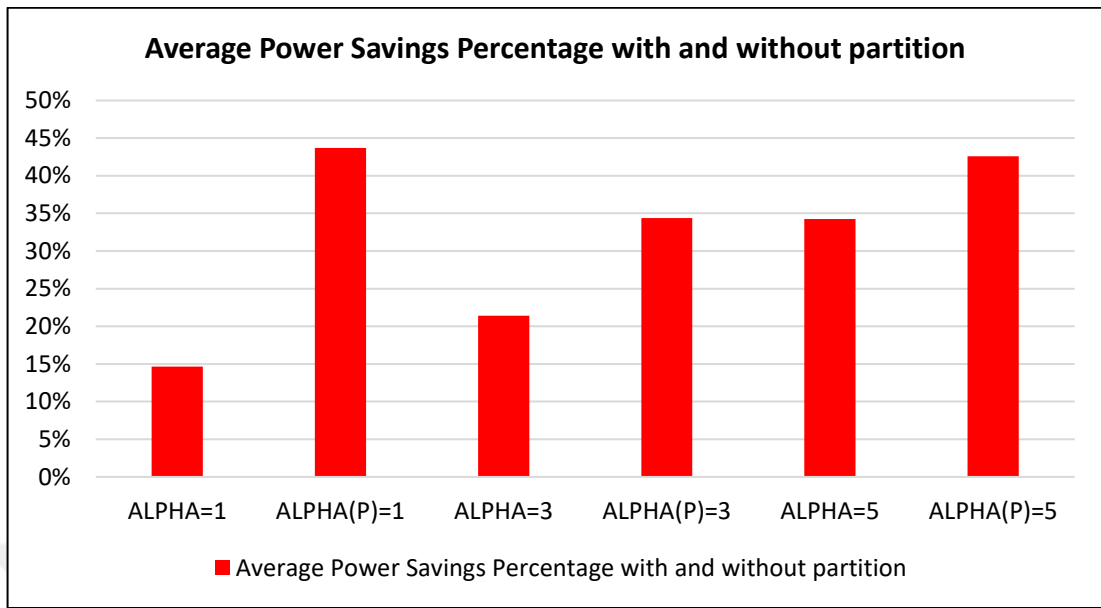


Figure 6.29. Percentage of power savings of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds

### 6.3.6. Energy-Delay product and Energy-Delay<sup>2</sup> product

To measure system efficiency, the power-delay product metric is used. The power-delay-product is calculated by multiplying the power consumption of the processor by time or delay. The delay represents the duration required to perform a computation. Moreover, lower power-delay-product means more energy-efficient systems. On the other hand, a system with a low power-delay-product may do calculations exceedingly slowly [11].

One of the important metrics used popularly to reflect system efficiency rather than power-delay-product accurately is the energy-delay product. Equation 6.4 is used for calculating EDP to measure the efficiency of the ASMMP design. When we compare the EDP value of the baseline processor with itself, this value will be one. This is due to the power expenditure on the baseline processor (Out-of-Order processor) will be 100 percent of the delay value, which is equal to 1.0 (base IPC/new IPC). Thus, the energy-delay product for the baseline processor will be equal to 1.

To compare ASMMP's efficiency to the Out-of-Order processor as a baseline, we examine Figure 6.30, which displays the obtained (100 percent-EDP) values for the sepc2006 benchmarks. More EDP values close to zero indicate that our design is efficient, as we

consumed less energy and had a shorter delay time while performing a computation. The percentage of EDP savings in ASMMP is 42 percent, 32 percent, and 35 percent for alpha values of 1,3, and 5 in the final ASMMP version, respectively. ASMMP's integrated resource partitioning mechanism ensured efficient use of processor resources for alpha value one, achieving savings of more than 28 percent over the non-partitioning version with a performance loss of less than 1,9 percent. Additionally, the final ASMMP appears to be more efficient than the non-partitioned version depicted in Figure 6.31, not just for alpha value 1, but also for alpha values 3 and 5. The final ASMMP's EDP savings exceed 13 percent with a performance loss of less than 2,96 percent and 11 percent with a performance loss of 13 percent, respectively. Also, parallel to EDP results obtained for the energy-delay square product of final ASMMP is shown in Figure 6.33. Lastly, Figure 6.32 shows that the final ASMMP for alpha threshold values one and three is more efficient in terms of energy-delay square product than the final ASMMP for alpha threshold value five.

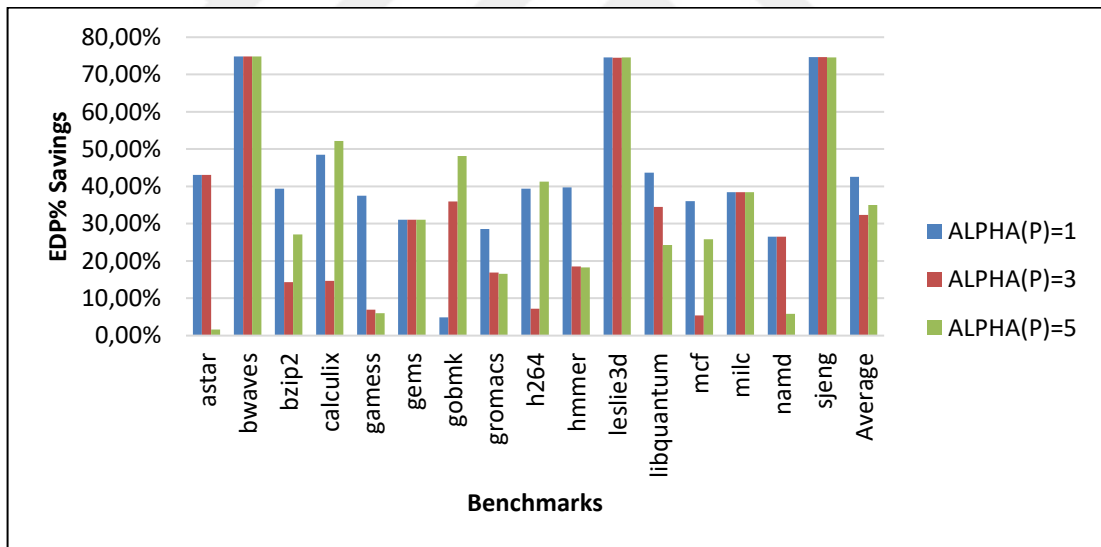


Figure 6.30. Percentage of EDP savings of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds

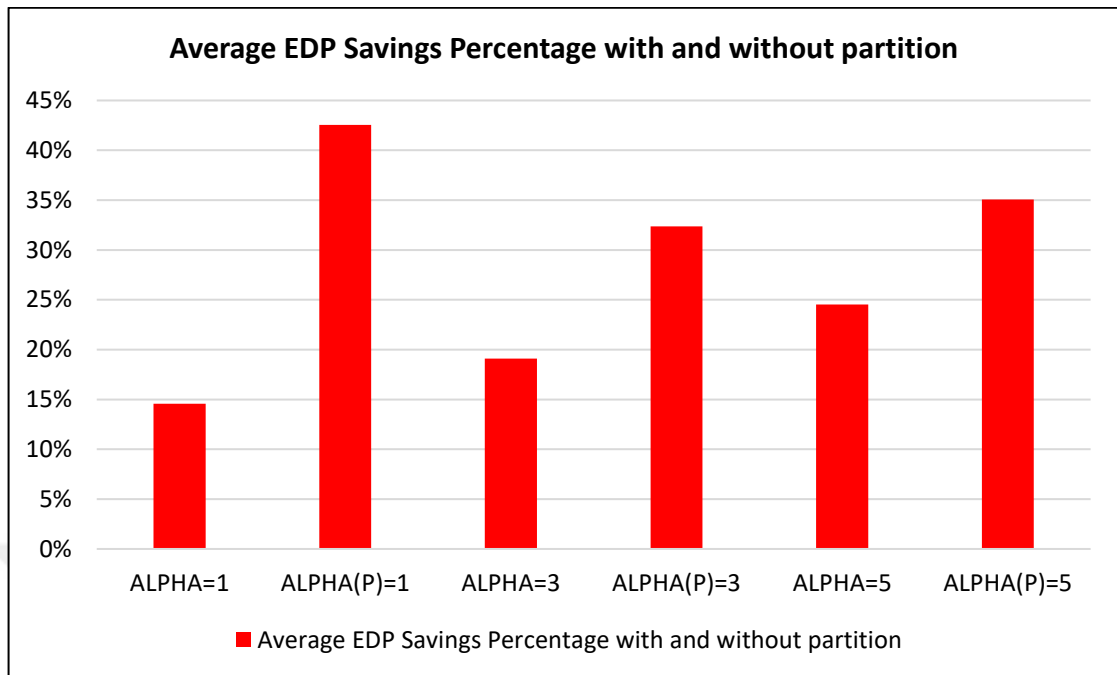


Figure 6.31. Percentage of EDP savings of ASMMP for both partition and non-partition version compared to baseline Out-of-Order processor for various alpha thresholds

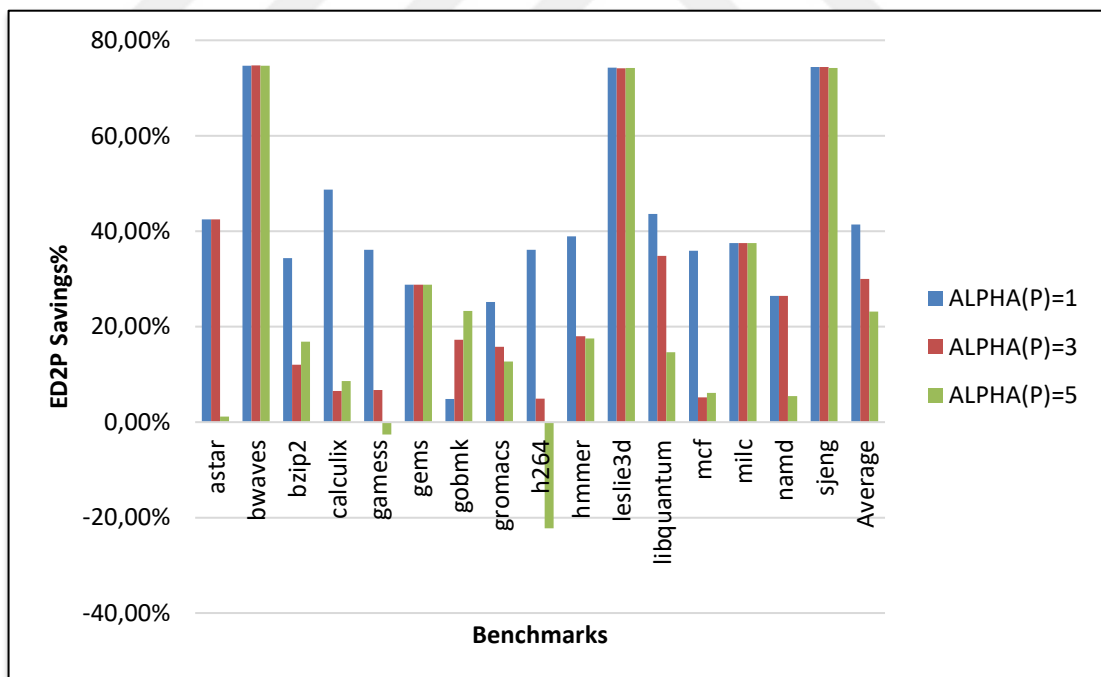


Figure 6.32. Percentage of ED<sup>2</sup>P savings of final ASMMP compared to baseline Out-of-Order processor for various alpha thresholds

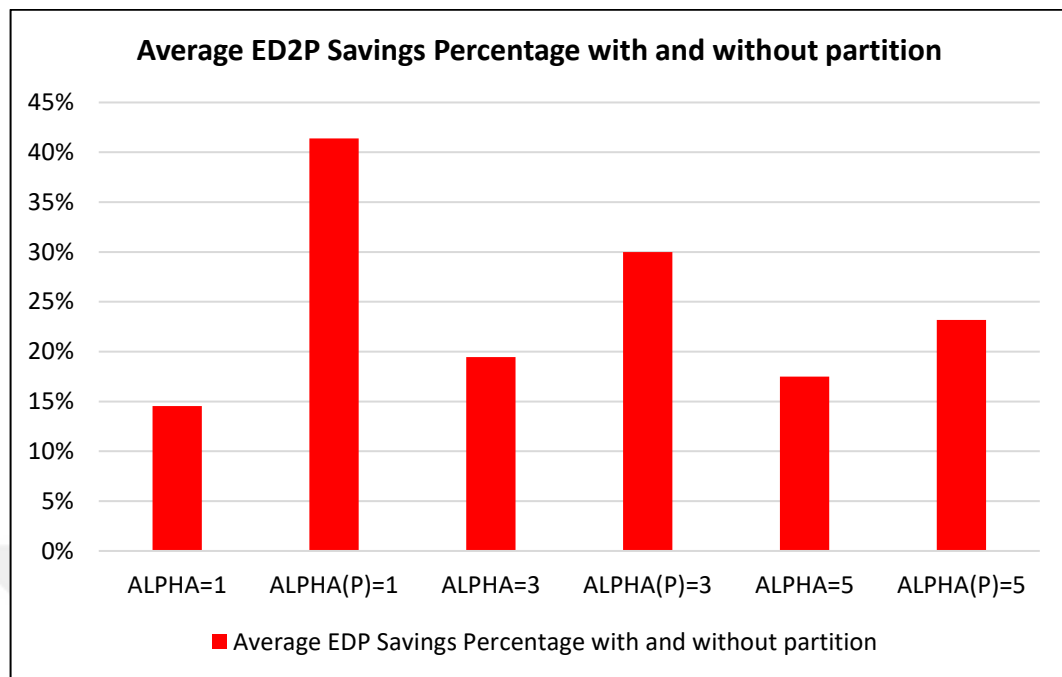


Figure 6.33. Percentage of ED<sup>2</sup>P savings of ASMMP for partition and non-partition versions compared to baseline Out-of-Order processor for various alpha thresholds



## 7. CONCLUSION AND FUTURE WORK

ASMMP, a single Out-of-Order superscalar processor capable of automatically scaling its resources in response to the needs of running applications and switching between In-Order and Out-of-Order execution modes, has been designed and tested in this thesis using Spec2006 benchmarks. ASMMP's primary objective is to achieve significant power savings in exchange for an average performance loss of 5 percent. Finally, Figure 7.1 depicts the complete implementation of our proposed ASMMP.

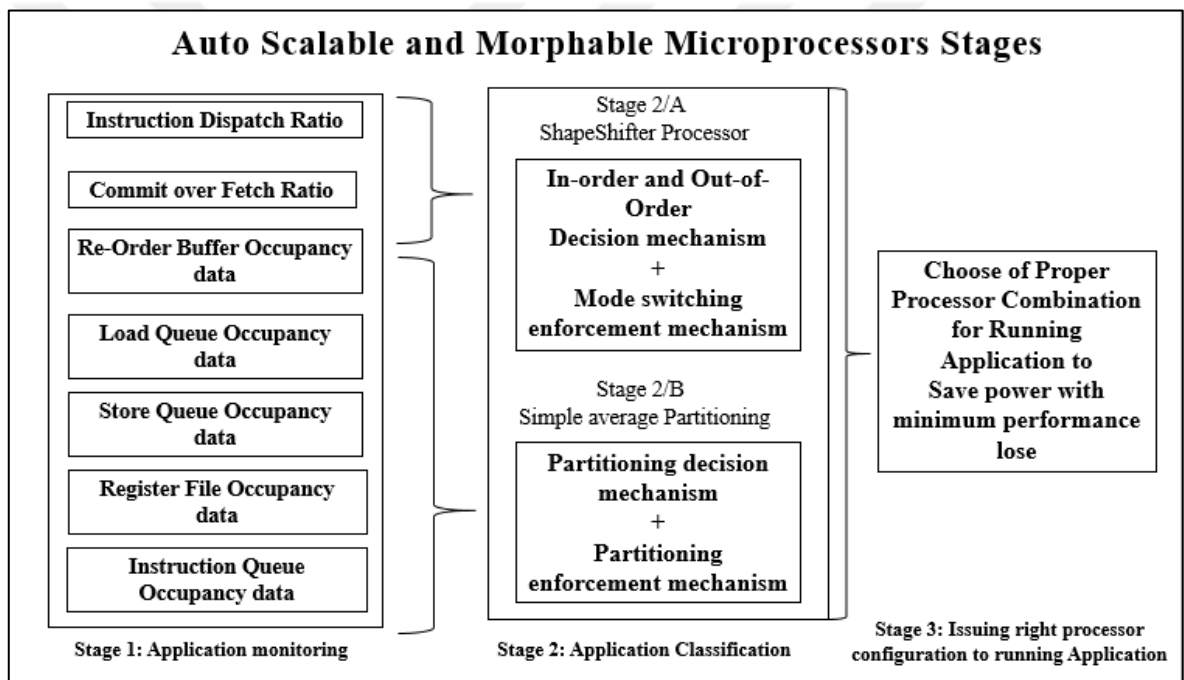


Figure 7.1. ASMMP implemented stages

We began this thesis by collecting and monitoring appropriate different processor parameters of the running application periodically. As a result, these parameters are typically collected using a sampling method that minimizes the proposed design's hardware complexity. The primary parameters that are dynamically collected are the utilization of different processor resources, the number of fetched instructions, the number of committed instructions, and the number of ready instructions in the issue queue for running applications.

As a result, collecting 250 samples per decision period (2M) for occupancy parameters and 100 samples per decision period (10K) for various instruction statistics was sufficient to

reflect the application's requirements. Besides, it provides useful information to assist ASMMP mechanisms in performing their duties properly.

We focus on two distinct mechanisms in the second stage: a simple threshold-based resource partitioning mechanism and a mode switching mechanism. First, these mechanisms were implemented and tested independently. They were then combined in the final stage to create the final ASMMP design. Our resource partitioning mechanism comprises two components: a decision mechanism and a mechanism for enforcing resource partitioning. We compare the average collected resource occupancy samples with the threshold value in the partition decision mechanism. Then, a resource expansion or contraction decision is made based on the result of the comparison. Furthermore, the partition enforcement mechanism ensures that new partitions are assigned securely, without erasing any critical data about the running application's instructions.

Moreover, two distinct execution modes (large core Out-of-Order for high-performance applications and small core IO for low-performance applications) are rather utilized as in the case with heterogeneous architecture [8]. Our ASMMP, a single-core, attempts to predict application performance when run in different execution modes by implementing an execution mode switching mechanism. The proposed mechanism consists of a mode switching decision mechanism and a mode switching enforcement mechanism with the resource partitioning mechanism.

Two parameters influence the decision-making mechanism. The first is the instruction dispatch ratio (IDR), which is the ratio of ready instructions in the instruction queue for both execution modes. The second parameter specifies the number of over-fetched instructions that have been committed (CFR). These parameters enable us to compare the performance of the Out-of-Order mode to that of the In-Order mode. When the speed-up value, the product of the two previously mentioned parameters, exceeds the specified alpha threshold, an Out-of-Order mode is selected. Otherwise, an In-Order mode is selected. Finally, if certain conditions are met, the mode switching enforcement mechanism will perform mode switching and assign the running application to the newly selected mode.

The final stage of this thesis is to integrate both mechanisms for them to work concurrently and assign the most appropriate ASMMP configuration to the running application. The proposed ASMMP architecture is a single-core processor capable of scaling its resources and switching between IO and Out-of-Order execution modes in response to application requirements. As a result, the Gem5 simulator and the Spec2006 benchmarks were used to evaluate the ASMMP architecture. Finally, the average performance loss of the ASMMP architecture is kept below 5 percent compared to the Out-of-Order baseline processor for alpha values one and three. Additionally, ASMMP saved more than 43 percent and 34 percent of total processor power, respectively, and 42 percent and 32 percent in terms of energy-delay product across all simulated Spec2006 benchmarks when alpha values were one and three.

We intend to perform resource partitioning on level two and level three caches in the future because these structures consume a disproportionate amount of power compared to other on-chip structures. Also, rather than using a fixed alpha threshold value for the mode switching decision mechanism, we can use a dynamic method to determine the alpha threshold value based on application behavior and which execution mode is unquestionably superior.

## REFERENCES

- [1] Sparsh M. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.* 2016;48(3):38.
- [2] Binkert N, et al., The gem5 simulator, *SIGARCH comput, Archit. News.* 2011;39:1-7.
- [3] Henning JL. SPEC CPU2006 benchmark descriptions, *SIGARCH Comput, Archit. News.* 2006;34:1-17.
- [4] Bahar RI and Manne S. Power and energy reduction via pipeline balancing. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001: Annual.
- [5] Albonesi DH, et al., Dynamically tuning processor resources with adaptive processing, *Computer.* 2003;36:49-58.
- [6] Manne S, Klauser A and Grunwald D. Pipeline gating: speculation control for energy reduction. *Proceedings of the 25th Annual International Symposium on Computer Architecture*; 1998: Annual.
- [7] Hubner M, et al., Dynamic processor reconfiguration, *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, 2011.
- [8] Lukefahr A, et al., Composite cores: pushing heterogeneity into a core. *Proceedings of the 2012 45th Annual 37 IEEE/ACM International Symposium on Microarchitecture*; 2012: IEEE/ACM.
- [9] Lukefahr A, et al., Exploring Fine-Grained Heterogeneity with Composite Cores, *IEEE Transactions on Computers*, 2016;65:535-547.

- [10] Folegnani D, Gonzalez A. Energy-effective issue logic, *Proceedings of the 28th Annual IEEE/ACM International Symposium on Computer Architecture*; 2001: IEEE/ACM.
- [11] Steinbach B, et al., *Recent progress in the boolean domain*. Cambridge: Cambridge Scholars Publishing.
- [12] Ghiasi S, Casmira J and Grunwald D. Using IPC variation in workloads with externally specified rates to reduce power consumption, *Workshop on Complexity Effective Design*, 2000.
- [13] Morancho E, Llaberia JM and Olive A. On reducing energy-consumption by late-inserting instructions into the issue queue. *Proceedings of the 2007 international symposium on Low power electronics and design*, 2007.
- [14] Homayoun H, Kontorinis V, Shayan A, Lin T and Tullsen DM. Dynamically heterogeneous cores through 3D resource pooling. *IEEE International Symposium on High-Performance Comp Architecture*; 2012: IEEE.
- [15] Afram F and Ghose K. FlexCore: A Reconfigurable Processor Supporting Flexible, Dynamic Morphing. *13 Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*; 2015: IEEE.
- [16] Khubaib M, Suleman A, Hashemi M, Wilkerso, C, Patt YN. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. *45th Annual IEEE/ACM International Symposium on Microarchitecture*; 2012: IEEE/ACM.
- [17] Sembrant A, et al., Long term parking (LTP): criticality-aware resource allocation in OUT-OF-ORDER processors. *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

- [18] Sleiman FM and Wenisch TF. Efficiently scaling Out-of-Order cores for simultaneous multithreading, *SIGARCH comput, Archit. News*. 2016;44:431-443.
- [19] Lebeck AR, Koppanalil J, Li T, Patwardhan J and Rotenberg E. A large, fast instruction window for tolerating cache misses. *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002.
- [20] Carlson TE, Heirman W, Allam O, Kaxiras S and Eeckhout L. The load slice core microarchitecture. *42nd Annual International Symposium on Computer Architecture (ISCA)*; 2015: IEEE/ACM.
- [21] Srinivasan ST, Rajwar R, Akkary H, Gandhi A and Upton M. Continual flow pipelines. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [22] Shioya R, Goshima M and Ando H. A front-end execution architecture for high energy efficiency. *47th Annual IEEE/ACM International Symposium on Microarchitecture*; 2014: IEEE/ACM.
- [23] Kumar R, Tullsen DM, Ranganathan P, Jouppi NP and Farkas KI, Single-ISA heterogeneous multicore architectures for multithreaded workload performance. *Proceedings. 31st Annual International Symposium on Computer Architecture*, 2004.
- [24] Kumar R, Farkas KI, Jouppi NP, Ranganathan P, Tullsen DM. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*; 2014: IEEE/ACM.
- [25] Annavaram M, Grochowski E and Shen J. Mitigating Amdahl's law through EPI throttling, *SIGARCH computer architecture. News*, 2005;33:298-309.

- [26] Kumar R, Jouppi NP and Tullsen DM. Conjoined-core chip multiprocessing. *Proceedings of the 37<sup>th</sup> Annual IEEE/ACM International Symposium on Microarchitecture*; 2014: IEEE/AMC.
- [27] Kim C, et al., Composable lightweight processors. *40th Annual IEEE/ACM International Symposium on Microarchitecture*; 2007: IEEE/AMC.
- [28] Ipek E, Kirman M, Kirman N and Martinez JF. Core fusion: Accommodating software diversity in chip multiprocessors, *SIGARCH comput archit. News*, 2007;35:186-197.
- [29] Savas M., Guney IA, Tokatlı NN, Kisinbay B and Kucuk G. iMODE (interactive MOod Detection Engine) processor. *4th International Conference on Computer Science and Engineering*; 2019: UBMK.
- [30] Greenhalgh P. Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7: Improving energy efficiency in high-performance mobile platforms, white paper. *ARM* 2001.
- [31] Tokatlı N, Güney IA, Sarı S, Güney Y, Nezir MU, and Küçük G. ShapeShifter: A morphable microprocessor for low power. *Turkish Journal of Electrical Engineering & Computer Sciences*, 2021;29(4):1964-1977. doi:10.3906/elk-2005-180.
- [32] Vidal J. 'Tsunami of data' could consume one fifth of global electricity by 2025 [cited 2017 12 December]. Available from: <https://www.climatechangenews.com/2017/12/11/-tsunami-data-consume-one-fifth-global-electricity-2025/>.
- [33] Mm µm 12. A history of microprocessor transistor count 1971 to 2013 [Internet]. *Wagnercg.com*. [cited 2021 Nov 30]. Available from: <http://www.wagnercg.com/Portals/0/FunStuff/AHistoryofMicroprocessorTransistorCount.pdf>.