OPTIMIZING QOS AND THROUGHPUT ON 4-THREAD SMT PROCESSORS

by

Uğur Nezir

Submitted to Graduate School of Natural and Applied Sciences

in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Engineering

Yeditepe University

2023

OPTIMIZING QOS AND THROUGHPUT ON 4-THREAD SMT PROCESSORS

APPROVED BY:

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

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

Assist. Prof. Dr. İsa Ahmet Güney .....
(Fenerbahçe University)

DATE OF APPROVAL: ..... / ..... / 20.....

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 Uğur Nezir

Signature .....

ACKNOWLEDGEMENTS

I am immensely grateful to my advisor, Gürhan Küçük, who has been both an inspiration and an illumination on my path in academia with his immense support and guidance. The journey I started with him during my undergraduate years and came to this point was a total delight.

I thank Sercan Sarı, Yiğit Bilgin, Mehmet Erdem Çakır, and Onur Demir who have significant contributions to this research. Without them, this thesis wouldn't be as it is. Also, I thank my professors who all have taught me incredible topics and displayed once again the magnificence of computer engineering. I would like to thank my company, Yongatek Microelectronics as well, for allowing me to commit the most valuable time I required to commit to this thesis.

There aren't enough words to thank the people around me for how they were and are great sources of inspiration and motivation both regarding academia and life. Working full-time and studying in a master's program meanwhile trying my best to do a satisfactory job on both was occasionally a draining process. Yet, I thank the people around me: my older sister Ece, who was both a great confidant and a recent Ph.D. that I hugely respect take an example of, my beloved Duygu, who was always there to support and encourage me whenever I felt up or down and ensured that all this journey was always a happy one; Louché, who was my number one partner and the person probably who pushed me further and further through all my master's journey, my family whom I cannot thank enough for all the support they provided; Baran, my confidant who stood by me through the same stages of life for more than fifteen years, İpek, my dear friend who weathered the storm that is called COVID with us, making those critical days of my master's phase much more enjoyable, and many other distingué people that I cannot count. I deeply thank you all for everything. I am the most fortunate to have you people around me.

ABSTRACT

OPTIMIZING QOS AND THROUGHPUT ON 4-THREAD SMT PROCESSORS

Simultaneous multithreading (SMT) is one of the key performance features in modern processors. However, in data center applications where immense computing power is required, there is a certain aspect that is much more important than overall performance: Quality of service. SMT, though, meanwhile getting its strength from sharing resources between multiple threads in a core, actually hurts the throughput of singular workloads, and a significant amount of these workloads require an exorbitant degree of quality of service. Thus, SMT, which is one of the breakthroughs lying under the modern processor microarchitecture, actually hurts data center applications due to its sharing nature. This handicap even leads the data centers to the point of disabling SMT on their processors to preserve their required quality of service. This topic has been occasionally studied in the literature and several propositions were made to overcome this barrier and obtain higher performance and increase performance per Watt and performance per total cost of ownership dollars. One of the leading studies is QoSMT which aims to enable a quality of service satisfying SMT approach in data centers. It achieves so by implementing a statistic-based dynamic resource partitioning scheme. However, the original QoSMT paper implements a design for two-threaded SMT cores meanwhile the modern data center processors use four or eight threads. When there are only two threads in a core, there is only a single transaction pathway between threads. High priority thread (HPT) takes or gives resources from/to low priority thread. When the thread count increases though, a necessity for resource scheduling arises as well and the statistical accuracy of partition decisions naturally degrades. In this study, we take the baseline QoSMT study, implement it, extend it to four-threaded SMT cores, suggest and implement some overall design improvements and analyze its capabilities for three different scheduling algorithms. We observe that our proposed best designs provided up to 66.63% quality of service improvements compared to the baseline SMT with only a 7.09% loss of total performance in terms of IPC.

ÖZET

4-İŞ PARÇACIKLI SMT İŞLEMCİLERDE HİZMET KALİTESİ VE İŞ ÇIKTISI OPTİMİZASYONU

Eş zamanlı çoklu iş parçacıklı işlemciler, modern işlemci mimarisinin en önemli performans unsurlarından biridir. Buna karşın oldukça yüksek bir işlem gücüne sahip olan veri merkezleri için performanstan çok daha önemli bir unsur söz konusu: Hizmet kalitesi. Eş zamanlı çoklu iş parçacıklı işlemciler gücünü, çekirdek içerisindeki farklı iş parçacıkları arasında kaynak bölüşümünden aldığı içinse veri merkezlerinin hizmet kalitesi hedefleri için önemli bir soruna gebe olmakta. Bu sorun, hizmet kalitesine ihtiyaç duyan iş yüklerinin kaynakların paylaşımından ötürü hizmet kalitesi kaybı yaşamasıdır. Bu sorundan ötürü, modern veri merkezleri, işlemcilerindeki en önemli performans unsurlarından olan eş zamanlı çoklu iş parçacıklı çalışmayı devre dışı bırakmayı dahi tercih edebilmekte. Bu komplikasyon, literatürde belli bir ölçüde araştırılmış ve muhtelif çözümler önerilmiş bir durum. Bu alanda önde gelen çalışmalardan birisiyse QoSMT. QoSMT, çalışma zamanı esnasında bilumum istatistiksel ölçümlemelere dayalı bir dinamik kaynak paylaşımı organizasyonu önermekte ve bu sayede yüksek öncelikli iş yüklerinin hizmet kalitesi standartlarını koruyarak eş zamanlı çoklu iş parçacıklı işlem kabiliyetinin de önünü açmayı amaçlamakta. Lakin bu çalışma oldukça başarılı ve saygın olsa da önemli bir eksiği mevcut: Modern veri merkezi işlemcileri dört yahut sekiz iş parçacıklı işlemcilerden oluşurken QoSMT, iki parçacıklı işlemciler üzerinde gerçekleştirilmiş bir çalışma. Bir çekirdek üzerinde iki iş parçacığı olduğu takdirde ortadaki kaynak paylaşımı kararları ise bir hayli basit oluyor: Yüksek öncelikli iş parçacığının düşük öncelikli iş parçacığına kaynak vermesi yahut ondan kaynak alması. İş parçacığı sayısı ikiyi aştığında ise bir kaynak paylaşım planlaması gereksinimi ortaya çıkıyor ve istatistiksel hesaplardaki değişken sayısıyla orantılı olarak hatalar artıyor. Bu çalışmada; orijinal QoSMT çalışmasını imite edip, bunu dört iş parçacıklı işlemcilere uyarlayıp, bazı geliştirmeler uyguladıktan sonra üç farklı planlama algoritması üzerinde incelemesini gerçekleştiriyoruz. Sonuç olarak en başarılı tasarımlarımızın, standart bir dört iş parçacıklı çekirdeğe kıyasla hizmet kalitesinde %66.63'lük bir artış sunduğunu ve bunu, hizmet kalitesi odağına karşın yalnızca %7.09'a varan toplam performans kayıplarıyla başardığını gözlemliyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

SMT	Simultaneous Multithreading
HPT	High Priority Thread
VM	Virtual Machine
CPU	Central Processing Unit
IQ	Issue Queue
LQ	Load Queue
SQ	Store Queue
ROB	Re-order Buffer
IPC	Instruction per Cycle
QoS	Quality of Service
OoO	Out-of-Order
ILP	Instruction-Level Parallelism
WAW	Write After Write
WAR	Write After Read
ALU	Arithmetic Logic Unit
LSQ	Load/Store Queue
RAW	Read After Write
TCO	Total Cost of Ownership
CAM	Contention Analysis Module
PAM	Partition Application Module
LPT	Low Priority Thread
CDM	Contention Detection Module
Intel CAT	Intel Cache Allocation Technology
SINGLE	Same workload on all four threads
DOUBLE	One workload on HPT, three instances of another workload on all three LPTs
MULTI	Four different workloads on all four threads
SMT-BASE	Baseline SMT with equal threads
QoSMT-RR	QoSMT with Round-Robin Scheduling
QoSMT-UB	QoSMT with Utility-Based Scheduling

QoSMT-RUB	QoSMT with Reduced Utility-Based Scheduling
predictiveQoS	Predicted QoS within the design
epochCycles	Processor cycle count used for each decision interval
contentionCycles	Predictive delay cycle count based on delay register
actualQoS	QoS calculated offline based on single-thread performance
$IPC_{qosmtHPT}$	Throughput of HPT in terms of IPC
$IPC_{singlethread}$	Throughput of a single-thread in terms of IPC
$IPC_{qosmtTotal}$	Total throughput of QoSMT core in terms of IPC
$IPC_{smtTotal}$	Total throughput of baseline-SMT core in terms of IPC
IPC_{smtTN}	Throughput of Nth thread of baseline-SMT core in terms of IPC
$IPC_{qosmtTN}$	Throughput of Nth thread of QoSMT core in terms of IPC
MAE	Mean Absolute Error

1. INTRODUCTION

With computing application demand spreading through the globe more and more each day, the requirement for computational efficiency increases as well. The more computational demand results in more processing units to be manufactured and used that consume an immense amount of power globally. According to IEA's 2022 reports, data centers consume around 220-320 TWh of electricity, which constitute around 0.9 to 1.3 percent of the global electricity demand, excluding the cryptocurrency mining operations, which was 100-140 TWh in 2021 [1]. Besides, with the end of Moore's law coming and internet of things applications becoming more popular, this amount is expected to multiply by 4 by 2030 [2].

Intelligent monitoring applications and management systems that utilize big data analytics, artificial intelligence applications, high frequency trading, simulation tools, robotics and automation applications are the main trends that utilize data centers today, and these trends increase the demand from data centers as well [3].

In the light of these studies, it can be said that data centers are expected to be a more pivotal topic of study in terms of both energy sector and computing applications. There are several ways to study this topic such as reducing the maintenance requirements, enhancing the cooling systems, utilizing smart software applications and so on. In this study, we focus on improving data center processors in terms of microarchitecture.

In data centers, simultaneous multithreading (SMT) is highly utilized in terms of processing. Processors used there aim to utilize all kinds of structures within the core with highest efficiency. Besides, some operations are highly latency sensitive and providing a certain degree of quality of service to these kinds of workloads is another pivotal topic. Unfortunately, these two aims clash with each other quite often.

Utilizing several workloads using varying architectural structures within a core is an important aim for obtaining value from data center processors. However, this regularly causes issues like this: Let's assume we are a search engine provider such as Google and millions of people use our website to do some web search every single second. With the size of online content today, even with the most optimized indexing methodologies, there would be an immense amount of data to be looked up within our servers to return relevant results. We have milliseconds to

provide results since if we would make users wait a few seconds for every single search, we would lose quite a lot of users to the competition who does it faster.

We have a huge environment with tens or hundreds of virtual machines (VM), each storing a portion of our search index. At this point, let's assume that each VM has a 4-threaded SMT core and these cores have each of their threads working on several different workloads, since we desire utilization. Within our hypothetical one hundred VMs, ninety-nine of our VMs returned "false", indicating they couldn't find what they were looking for and waiting for the 100th VM. However, in this core, there is a resource contention between threads; thus, search operation slows down significantly and we cannot return an exact result until the last operation is finished. On the other hand, we may just give all the threads to our priority workload but then, what would be the point of using an SMT core? We lose significant utilization from different and not-so-latency-sensitive workloads. In other words, we have a double edged sword in our hands. We want to maintain a degree of quality of service for latency-sensitive workloads, meanwhile having some processing power utilization for batch operations, as called in the study of Margaritov et al., which addressed significant solutions this problem [4].

This study is primarily based on Jin et al.'s QoSMT paper, which proposes several control circuitries to be implemented in 2-threaded data center central processing units (CPU) to address this issue as well. These control circuitries' aim is to observe high priority and low priority thread behaviour through several shared resources which are issue queue (IQ), load queue (LQ), store queue (SQ), re-order buffer (ROB), and cache; and according to the behavioral observations, partition these resources dynamically between threads to provide higher throughput in terms of instructions per cycle (IPC) yet keep a certain degree of quality of service (QoS) for the low priority thread that shares its resources with the high performance thread [5].

Keeping a certain degree of a quality of service, though, is a great challenge considering the requirements of measuring or predicting real time performance. In QoSMT, the authors suggest a structure that is based on a dedicated standalone interference counter. This counter keeps track of interference occurrences regarding the issue queue, re-order buffer, load-store queue, L1 data cache, L1 instruction cache and unified L2 cache. Using this dedicated counter,

it determines a potential standalone throughput without the interferences and compares the result with its actual throughput in each predetermined epoch time. If the ratio of these two does not meet the predetermined quality of service percentage, then the system transfers resources from the low priority thread to the high priority thread. Otherwise, the resource transfer would be held from high priority to low priority thread. However, the exchanges occurring between only two threads make the decisions and the regarding implementation trivial since there is only one possible source of contention and the high priority thread resource transfers can only occur with that single source of contention.

In this study, we focus on implementing the approach not to two threaded but to four threaded SMT cores. The reasoning for this mainly is partitioning resources between two threads is a fairly straightforward approach: When you need resources on a specific thread, then you simply take resources from the other thread and it's done. Yet, modern data center microporcessors have 4, 8, sometimes even 16 threads per core; thus, when a high priority thread requires more resources, then there should be a decision-making on which thread or threads shall sacrifice their resources between the threads to provide resources to the high priority thread. This requires a more intricate resource partitioning scheme and this is where we primarily aim to contribute to the literature.

The difficulties in partitioning between four threads can be broken down into four main parts. Firstly, the hardware complexity increases since there are more resources being partitioned. This increases the complexity of resource allocation algorithms. Secondly, the contention increases because now, there are more threads requesting cache, memory, I/O device accesses. This contention can lead to performance degradation. Thirdly, dependencies increase. When the thread count increase, the possibility of a thread input depending on another thread's output increases. So, the requirement for ensuring each thread having adequate resources becomes more important. Lastly, limited resources would mean that increasing thread count comes with thinner slices of resources. Therefore, minor resource deallocations may cause severe performance overheads in singular-thread throughput. [6].

This study extends the original QoSMT paper in two main ways: Firstly, we implement resource partitioning algorithms to a four threaded SMT core. Secondly, we apply some improvements to the partitioning schemes to achieve better results. In the end, we compare

our results with baseline 4-threaded SMT core in terms of throughput, quality of service and fairness.

2. BACKGROUND & RELATED WORK

In this section, we overview some of the underlying technologies and preliminary work that hold the most importance regarding our study. These can be listed as:

- Out-of-Order Execution
- Simultaneous Multithreading
- Data Center Processing
- Quality of Service
- Resource Partitioning
- QoSMT : Supporting Precise Performance Control for Simultaneous Multithreading Architecture
- Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores

2.1. OUT-OF-ORDER EXECUTION

Out-of-order execution (OoO) is a method in computing that makes use of the instruction cycles that otherwise-would-be-unused. Some instructions take short execution times yet when executing some longer ones like floating-point division or memory operations, for example, the following instructions wait idly while they could be executed as if they weren't waiting in a queue.

Superscalar approach has been introduced to address this issue initially. The design was rather simple: An in-order executing core would have a U-shaped fetch and decode mechanism when it is superscalar; such as Intel's Pentium architecture [7]. This mechanism is capable of fetching and decoding two instructions in parallel from a u-pipe and v-pipe. Which could be thought as a pseudo out-of-order mechanism. This parallelism had an issue though: These instructions had to require independent data and some independent resources. However, most of the instruction pairs that were checked for this possible parallelism were dependent since consecutive instructions have a higher tendency to be related. Therefore, superscalar cores

were a massive breakthrough in terms of achieving instruction-level parallelism (ILP); yet, they were far from perfect on their own with the limitations imposed from in-order execution.

The out-of-order approach utilizes some additional microarchitectural components that keep track of out-of-orderly executed instructions to not mess up instruction ordering so that the core can execute the waiting instructions and utilize unit time. The first implementation of out-of-order execution was in CDC 6600 which utilized a scoreboard mechanism to keep track of instruction ordering. However, this design was prone to write-after-write (WAW) and write-after-read (WAR) conflicts and arithmetic logic units (ALU) could stall [8]. Two years later, IBM System/360 Model 91 was released which utilized Tomasulo's algorithm and brought register renaming concept, which let it not to suffer from the two conflicts that CDC 6600 had; therefore, enabled full out-of-order execution in a core. Register renaming today is still used in modern OoO microprocessors [9].

Since long before to today, most out-of-order cores utilize superscalar approach as well to achieve a greater performance, one of the milestones being Pentium II [10].

2.1.1. Issue Queue

The issue queue (IQ) is a data structure that keeps track of the instructions that have been fetched and are waiting to be executed. When an instruction is fetched, it is placed in the issue queue, along with any other instructions that it is dependent on [11].

The issue queue is responsible for managing the dependencies between instructions and ensuring that they are executed in appropriate instances. The issue queue checks for register dependencies and executes instructions as soon as registers are ready.

When an instruction is ready to be executed, it is removed from the issue queue and sent to the execution units. The execution units perform the actual computation once they are sent from the issue queue. The issue queue then updates the status of the instructions and checks to see if any dependent instructions can now be executed. In order to allow superscalar execution, issue queues may have more than a single read and write ports [12].

Overall, the issue queue is a critical component of out-of-order microprocessors, as it allows

instructions to be executed out of order while still ensuring correct program execution. By carefully managing the dependencies between instructions, the issue queue helps to maximize the performance of the microprocessor while minimizing the risk of errors or incorrect program behavior.

2.1.2. Re-order Buffer

The re-order buffer (ROB) is a data structure used in out-of-order execution microprocessors to keep track of instructions that have been issued but not yet completed. It is essentially a circular buffer that holds instructions in their original program order until they can be executed [13].

The ROB is designed to solve the problem of instruction dependencies that can occur in out-of-order execution. Because instructions can be executed out of order, it is possible for later instructions to depend on the results of earlier instructions that have not yet completed. The ROB helps to manage these dependencies by keeping track of the original program order of the instructions and ensuring that they are committed in the correct order.

When an instruction is fetched, it is first placed in the re-order buffer. The re-order buffer holds instructions in program order until they are ready to be issued for execution. When an instruction is issued, it is placed in the issue queue along with any data dependencies that it has.

The ROB keeps track of the instruction's program order, its status (e.g., issued, executed, committed), and any data dependencies. It also has pointers to the physical registers that hold the instruction's operands.

When an instruction is executed, its results are written to the physical registers and its status in the ROB is updated. If the instruction has no such dependencies, it is marked as ready for commit. When an instruction is committed, its result is written to the architectural register file and the ROB entry is freed.

The ROB provides several benefits to out-of-order execution microprocessors. It allows instructions to be executed out of order while still ensuring correct program execution, and it

provides a buffer between the processor's execution units and the architectural register file, allowing for efficient scheduling of instructions. By managing dependencies and ensuring correct program order, the ROB helps to maximize performance and minimize errors in out-of-order execution microprocessors.

2.1.3. Load Store Queue

Load store queue (LSQ) is another important structure in an out-of-order microprocessor. In issue queue, some dependencies are handled. However, identifying memory dependencies is harder since we require effective addresses for such dependency checks [14].

In a modern out-of-order microprocessor, the load store queue is designed to absorb bursts in cache accesses and to maintain the order of memory operations by keeping all in-flight memory instructions in program order. So, we can say that this is a specialized extension to support issue queue only for load and store operations, as its name suggests.

The load store queue calculates the effective addresses when registers become available and checks for read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazards in each processor cycle and decides on the available load and store instructions that can be issued.

As CPU core clock speeds increase, bus delays with the cache and memory get worse relatively. As a result of it, in-flight memory instructions in the pipeline would increase, increasing the load and store dependency pressure within the system. Besides, the issue queue sizes increasing to improve performance also increases load and store operations to be handled correctly as well. Therefore, load store queue is becoming more and more important handling these both in terms of reliability and performance [15].

2.2. SIMULTANEOUS MULTITHREADING

Simultaneous multithreading can be called a breakthrough as a technique that improved superscalar processors to a great extent. In order to explain it, we may use a figure. Firstly there was superscalar processing concept which can be seen in Figure 2.1 (a), in which a

processor tried to issue multiple cycles in a single program's execution time whenever it can. When it cannot, some of the issue slots weren't used.

Figure 2.1 (b) is an advancement on superscalar, which is the fine-grained multithreaded superscalar implementation. This approach granulates multiple instructions and alternate through different threads. The main advantage of this is that during execution of long latency operations there were cycles that don't issue anything in its predecessor. However alternating different threads, we significantly reduce unused cycle amount.

Figure 2.1 (c) part is the SMT approach where the processor just executes operations from any thread any time, dynamically scheduling them later and exploit instruction level parallelism (ILP). Thread workloads that already have high ILP wouldn't benefit from this approach much, while the workloads that have low ILP, SMT processors can just speed up execution incredibly. [16]

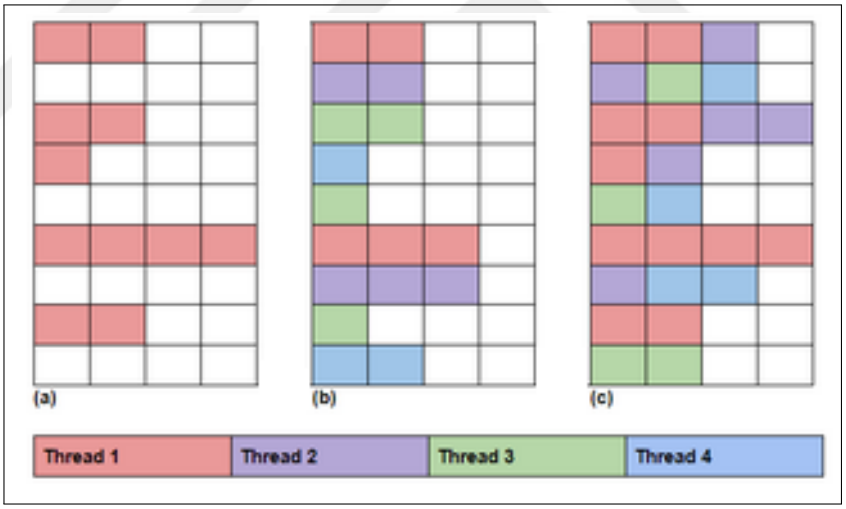

Figure 2.1. Instruction commit visualisation for (a) Typical superscalar, (b) Fine-grained multithreaded superscalar, (c) Simultaneous multithreading execution.

Simultaneous multithreading is achieved through several changes on a multithreaded superscalar processor. It requires register ports to be updated, instruction dependency checking circuity, forwarding logic, a revised instruction scheduling mechanism, and also a new cache structure to be implemented. However with its initial design, it results in speedups up to 4 times against single-threaded wide superscalar execution, and up to 2 times faster than fine-grained multithreading according to Tullsen et al [17][18].

2.3. DATA CENTER PROCESSING

In data centers, achieving maximum performance per Watt and per total cost of ownership (TCO) per dollar is an aim that is strived upon. One of the main approaches to achieve this goal is colocation of workloads within cores; mainly, running latency-sensitive and batch workloads together as Google does [19].

As some related studies have shown, the colocation causes significant drawbacks to the latency-sensitive, priority operations' quality of service [20]. This goes to the point of disabling simultaneous multithreading in those cores and wasting their potential immensely. Or, trying to utilize batch operations only when latency-sensitive workloads are idle.

These handicaps show that in order to increase performance per Watt and TCO per dollar, there is a great untapped potential within enabling colocation of workloads and simultaneous multithreading in data center microprocessors; if it could be achieved efficiently. This is our main motivation regarding our research.

2.4. QUALITY OF SERVICE

Quality of service is the measurement of overall system performance of a service, regarding how fast, efficient and reliable the service is provided. In today's world where cloud infrastructures are everywhere in our daily lives, quality of service mostly equates to quality of life as well; which makes this subject crucial for users in terms of life quality and for companies in terms of obtaining and maintaining a consumer base [21].

In an example, a web search with millions of result queries sorted in a relevance and content scoring manner is held only within 410 milliseconds as of writing this thesis. These results are quite accurate regarding the search query and intent as well. This can be described as a service that has a high quality of service; accurate and fast. However, these search results are obtained through tens of billions of web pages indexed from the internet and considering this, even the simplest of search queries require immense computational power behind them. However, if a company would be providing the same results within 3 seconds of time, let's say; that company wouldn't live for long considering the competition. This is

why quality of service is a matter of utmost importance and couldn't be discarded for some slight performance gains and this is the reason aforementioned data centers preferring the path to disable their microprocessors' simultaneous multithreading capability and causing more than 50% loss of performance within their cores just to uphold high quality of service standards [22].

2.5. RESOURCE PARTITIONING

In modern microprocessors, multithreaded architectures are being implemented within singular cores. This approach partitions several resources between the colocating threads in several manners. Intel, for example, mostly adopts static partitioning of resources thanks to its simplicity and avoiding resource starvation within cores [23].

On the other hand, there are several studies showing that dynamic partitioning of resources is actually a significant gain in some or all internal resources: Such as issue queue, re-order buffer, load store queue and caches [24][25].

These studies display the potential within resource partitioning within SMT cores and showcase different partitioning methodologies that could be implemented. These preliminary studies have an immense impact regarding creating the baseline of our approach when our goal is to achieve higher TCO values from data center microprocessors through utilizing SMT while pertaining acceptable quality of service values.

2.6. QOSMT: SUPPORTING PRECISE PERFORMANCE CONTROL FOR SIMULTANEOUS MULTITHREADING ARCHITECTURE

QoSMT is our baseline study regarding this research. This study implements a quality of service centric approach in order to enable simultaneous multithreading in a manner that would extract shrouded throughput and instruction-level parallelism from data center microprocessors.

Their design suggests a novel approach to achieve this: Utilizing real-time metrics to observe and record resource contentions through the execution of workloads, analyzing them

regularly and deciding on intra-core resource allocation between two colocating threads. As aforementioned, the mechanism checks resource contentions of six different resources and partition them dynamically. The resources that are partitioned are issue queue, re-order buffer, load store queue, instruction L1 cache, data L1 cache and unified L2 cache.

They extensively review the literature and evaluate their design and display that their proposed architecture meets the quality of service demands between 96 to 99 percent of the execution time while increasing throughput within almost all their tests compared to the previously suggested studies.

Our goal regarding QoSMT is to take this approach, implement and adapt it to support four threads to provide a baseline for scheduling concerns, address them and improve the proposed solution, and make it a more feasible and grounded approach to be applied into modern data center microprocessors.

2.7. STRETCH: BALANCING QOS AND THROUGHPUT FOR COLOCATED SERVER WORKLOADS ON SMT CORES

Stretch is another study which is fundamental regarding our research. This study suggests other microarchitectural structures to be implemented in data center processors so that enabling SMT and colocation of workloads within singular cores could be achievable without interrupting the quality of service requirements.

Their approach is based on running batch workloads in parallel with the latency sensitive workloads as well and they achieve it with a re-order buffer partitioning scheme. This partitioning scheme is asymmetrical where one of the partitions is kept much larger than the other. It utilizes two modes; B-mode for boosting batch processes when the latency-sensitive thread load is under low or moderate load, and Q-mode for boosting the quality of service when the latency-sensitive thread is under a high load. These decisions originate from the wisdom regarding the impact of reducing ROB size on performance after a certain point.

This study, like QoSMT, is based on two-threaded cores as well. It implements a lightweigt metric and scheduling mechanism which is mostly parallel to the processor's critical path and utilize Google's CPI2 software framework to pass software-based workload decisions into

the ROB partition mode switches in core-level.

3. DESIGN & IMPLEMENTATION

This study focuses on 4-threaded SMT microprocessors, which are highly popular in data center applications. Data center applications contain workloads that immensely utilize multithreading; such as graph searches, high frequency trading, big data operations and so on. These applications may even utilize thousands of threads concurrently, by using distributed computing approaches.

3.1. SMT IN DATA CENTER APPLICATIONS

Certainly, data centers run several different applications concurrently and the processes have differences as well. When different threads start running concurrent workloads, pipeline resources within a core start to be distributed among the threads. The resource distribution is handled in different ways. For example, Intel uses a static partitioning scheme in their cores [26]. This may be a fair approach; however, this approach often cause at least one, possibly all the threads to suffer from a lack of adequate resources. One application may require a huge amount of cache with quite few re-order buffer lines meanwhile another application would require the exact opposite. In such a scenario, these two applications both suffer from a static partitioning scheme because one wouldn't have enough cache and the other wouldn't have enough re-order buffer lines. In contrast, IBM, though, employs a partial partitioning scheme which allows workload prioritization and limits the instruction fetch ratio to provide the prioritization [27]. This allows a partial prioritization of workloads within a core. However, there are several architectural structures within a microprocessor, and SMT originally aims to utilize all these structures. So this approach is quite against its own design philosophy: Such a limitation would restrict what could be extracted from an SMT core.

Our first focus in this work was to apply the design philosophy of QoSMT into a 4-threaded core. In order to do it, we needed to implement two main design components: First, we need a contention analysis module (CAM) which would keep track of resource contentions occurring within the core between the threads. Then, we would need a partition application module (PAM) which would calculate resource partitioning amounts for the future according to the data it has in hand at the moment and apply resource partitioning within the core.

In our design, we assume that we have one high performance/priority thread (HPT) as well and the other three threads are low performance/priority threads (LPT). The reason for this design decision is that if there would be higher amount of HPTs, this would mean that quality of service that could be provided to an HPT would contend with the other, harming each other's quality of service caps significantly meanwhile leaving a much smaller amount of resources to utilize throughput from LPTs. Considering this, colocating multiple HPTs in a single core's threads would be infeasible.

3.2. CONTENTION ANALYSIS MODULE

In original QoSMT paper, there is a contention detection module (CDM) implementation which detects contentions from six different structures within the core:

- Issue Queue
- Re-Order Buffer
- Load-Store Queue
- L1 Instruction Cache
- L1 Data Cache
- L2 Unified Cache

All of these resources have their own dedicated contention tracker counters that are being used to calculate a delay prediction for the HPT, which would be used as the baseline source of information for the resource partitioning decisions.

The contentions need to be identified, though. When an instruction stalls because it couldn't find a suitable line of IQ, ROB or LSQ; it doesn't necessarily happen because of another thread's contention. It may be just because the structure is occupied by itself, such as a cache conflict. How we separate these is by using a shadow counter for each of these structures. This counter simply counts occupancy of our HPT within the structures. A contention is identified when an instruction of HPT stalls, yet, the shadow counter for that structure is smaller than the total size of the said structure. So, this simple logic allows us to identify real

contentions during the runtime with ease as shown in Figure 3.1.

However, we propose to use a shared delay register which is incremented with existence of any contention within an instruction cycle. Observing multiple contentions is certainly more informative; however, a contention in a cycle wouldn't get worse with multiple contentions since there already is an affected thread performance. Besides, this approach reduces the computational cost of resource allocation calculation. Therefore, it would require less circuitry, which means less area and power consumption. The main problem with this is, especially on the cache part, we cannot certainly determine the stall time caused by a cache miss. So, we just share the same delay amount with all the contentions, which could at least provide an overall idea regarding the situation.

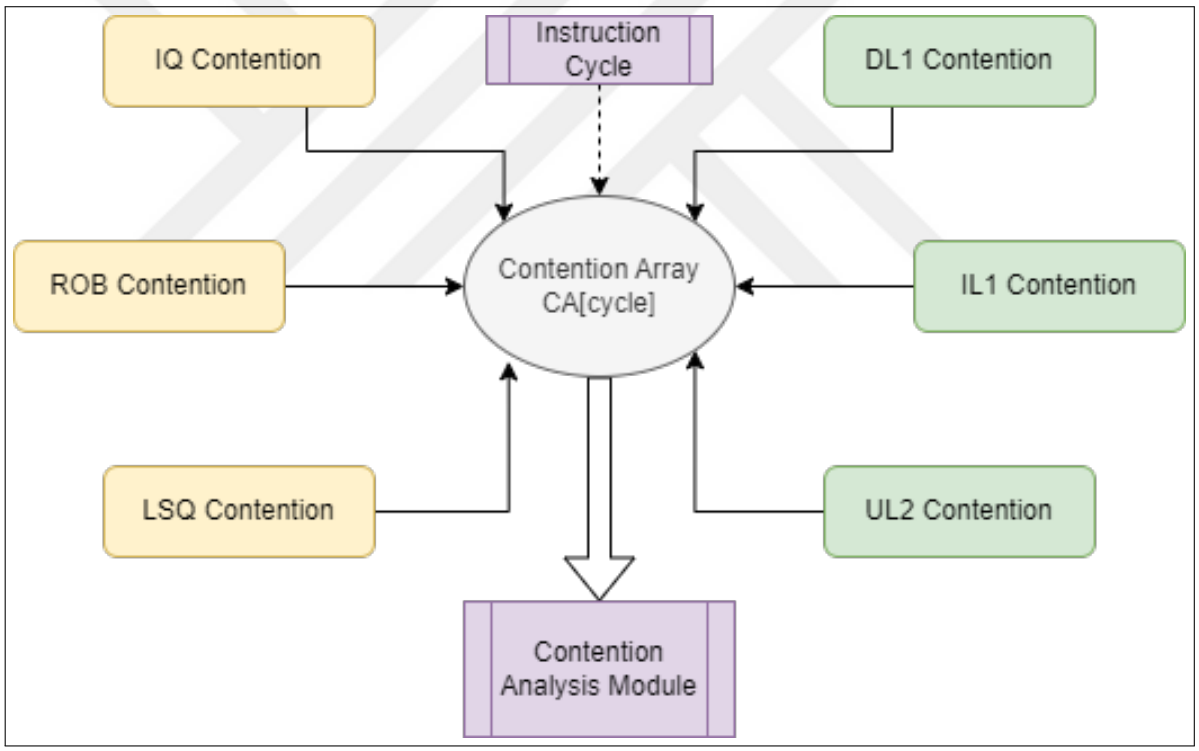

Figure 3.1. Contention Analysis Module illustration

3.3. CACHE CONTENTIONS

When we think about issue queue, re-order buffer or load-store queue; the contention predictions are easier to make. Yet, when the cache is the resource in hand, we need to be able to differentiate whether the reason of cache miss is the HPT itself or the LPT caused

the eviction. In order to keep track of it, we utilize auxiliary tag arrays in our cache structure. If a cache miss occurs when the relative tag array is occupied by another thread, only this means that the cache block in question was evicted by another thread. So, the aforementioned miss counts as a contention within our delay register. Figure 3.2 showcases the tag array implementation. Algorithm A.1, Algorithm A.2, and Algorithm A.3 can be further examined for implementation details in simulator.

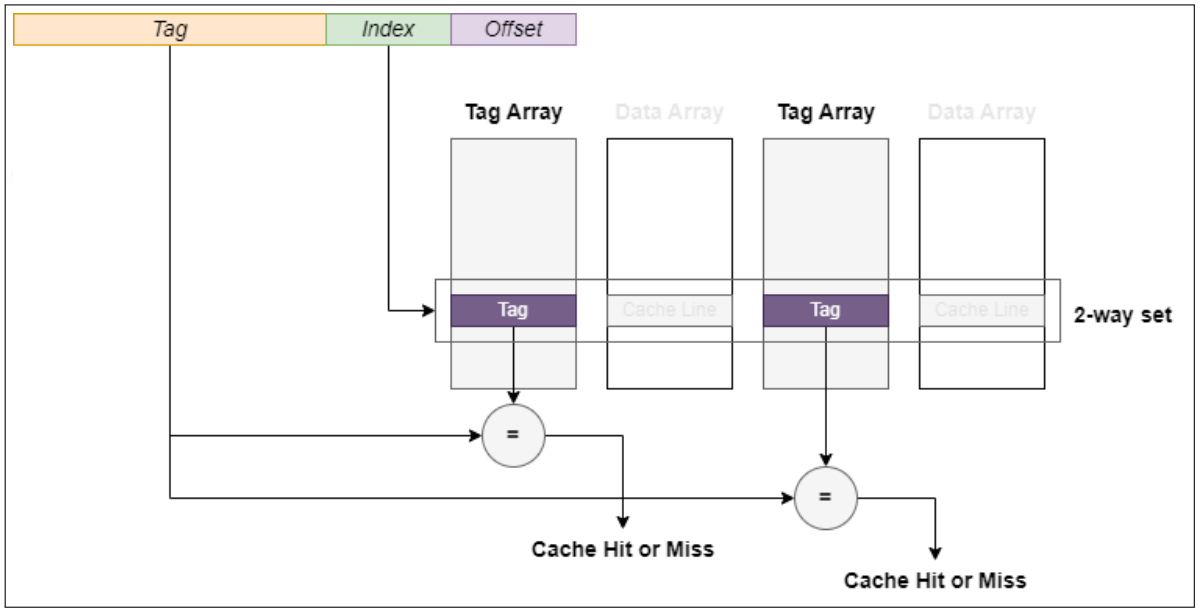

Figure 3.2. Tag Array Cache illustration

3.4. PARTITION APPLICATION MODULE

Once we have the metrics regarding the resource contention, we need to utilize them effectively to predict the future workload trend accurately. This is where our partition application module (PAM) kicks in and makes the decisions regarding the resource allocation for each thread to get the most throughput meanwhile maintaining a certain QoS threshold for our HPT.

Architectural resources listed before are not partitioned just with a flick of a finger, though. When partitioning resources, we need to consider switching power consumption and the duration of a thread letting that particular resource go to be used in the other thread; i.e., if there already is an issued instruction within an issue queue line, we cannot just give it to the other thread, since overwriting it with another thread's instructions would mean instruction

loss. Considering the power and reliability concerns, there should be a certain activation time between the allocation decisions. Besides, this would also let us have more data to address within each decision period and have more accurate predictions. In the end, we are just trying to foresee the future workload trend and adapt to it using data from the past. In such a scenario, mispredicting the future can highly degrade the runtime performance and the system would cause losses rather than gains.

In order to provide such a window, PAM runs its routine only once after each predetermined epoch time. This partitioning mechanism is quite a feather-light approach since it requires only very little amount of resources and it doesn't locate within the critical instruction path. This approach may not somehow provide the 100% of the uptime required for all the partition transfers to happen since no eviction of any resource is guaranteed to occur in a fixed time interval. However, we are aiming a best-effort approach here and the edge cases can only slightly affect the overall partitioning effectiveness. This parallel and lightweight nature of our proposed mechanism lets this approach work effectively without impacting the execution cycle time and costs only a negligible die area, which is explained in detail later in this thesis.

3.5. MSIM SYSTEM SIMULATOR

M-Sim, is a SimpleScalar 3.0d based architectural simulator [28]. The SimpleScalar is a toolset that provides a simulation environment of a computer system with CPU, cache and memory hierarchies [29]. Meanwhile SimpleScalar is a time-proven tool set that was used in many studies, it lacked some modern utilites. M-Sim extends the SimpleScalar with accurate pipeline modeling, explicit register renaming, and support for simultaneous multithreading. It also comes with Wattch model that SimpleScalar utilizes as well for power analysis. With all these capabilities, M-Sim is an ideal simulation environment that enables us to hold this study accurately.

3.6. UTILITY-BASED CACHE PARTITIONING

There are different approaches that could be taken for resource allocation for SMT cores. The resources could be all-shared, back-end static partitioned (Intel approach), or completely

static partitioned. These approaches were analyzed by Jin et al. and as it is observed, complete static partitioning is the superior approach of these three regarding the decreasing the performance variation.

Even though the completely static partitioning looks like the best option for reducing performance variation, it is far from perfect. The resource allocation is not always directly correlated with the performance, and some workloads may suffer considerably because of such an approach. For example, if we have a cache-sensitive workload in our high-priority thread while the low-priority threads do not require much cache, then we would be just taking the most important resources from our high-priority thread to not be utilized by the other threads. This is an immense performance loss scenario that nobody would prefer to have. In the end, data center processing units may be used for all kinds of workloads and such significant issues would restrict their utilization. Intel, for example utilizes Cache Allocation Technology (CAT) in their Xeon cores to partition L3 caches [30]. However, this software-based approach isn't applicable to L1 and L2 caches. So, there still is a large room for optimization regarding the L1 and L2 caches.

However, there are several approaches for partitioning resources according to utilization. Qureshi and Patt suggests a utility-based cache partitioning scheme that utilizes tag-array caches to achieve significant gains within SMT cores. Which is the method we adopt in our system regarding the partitioning of caches. The implementation details are provided in Algorithm B.1, Algorithm B.2 and Algorithm B.3.

3.7. DYNAMIC RESOURCE ALLOCATION DELTA

The contention-based resource partitioning is an already discussed approach: The resources within our core is allocated between high-priority and low-priority threads with keeping the quality of service target as the primary consideration and decides the allocations utilizing different partitioning schemes. However, there is a reallocation unit used for this partitioning scheme in the antecedent study. This allows a reasonable partitioning application speed and reliability which doesn't miss out the most optimized partitioning by a significant margin meanwhile allowing the partitioning to take place in a fast-enough manner.

However, as already discussed before, keeping things static may work well but it most likely isn't the most efficient way to them. We can say the same for a static reallocation unit, or, resource allocation delta. In our study, we also improved the Jin et al.'s base study in terms of this allocation delta.

The implementations required to achieve this actually are used the calculations we already were utilizing within the baseline QoSMT. In each epoch cycle, there are decisions made regarding the target quality of service threshold and our current estimated quality of service obtained from our calculations. These calculations allow us to predict whether we are providing enough quality of service to our high-priority thread or not and allocate resources between the threads accordingly. So, we already check our current estimated QoS and target QoS in our hands. This is the point where our dynamic delta kicks in: For each 10% of missing QoS, we multiply our resource allocation delta with that amount. In other words; if we are 0-10% below our designated QoS threshold, high-priority thread would be provided with our DELTA amount of resources. If we are 10-20% below our threshold, then the high-priority thread would be provided with two times our DELTA amount of resources. If there aren't enough resources on the LPT chosen for providing resources to HPT, the respective thread provides the maximum amount of resources it can provide, partially satisfying the multiplied delta amount. This base delta amount is a fine-grained unit and is never divided. This, also provides efficiency in hardware overhead.

This shared DELTA multiplier between the back-end resources (IQ, ROB, LSQ)) allows us to let our design adapt to changing workloads faster than the baseline QoSMT study with only a single parallel calculation held in each epoch cycle; thus, increasing the feasibility of our approach to the modern data center processors where a consumer would prefer as much adaptability as they can. The term "amount of resources" is used since each resource has its own set reallocation unit.

There is also another issue of oscillation with delta approach. Under certain circumstances, giving out resources from HPT may cause our calculated quality of service to fall down so drastically so that it would require resources immediately in the next decision cycle. When it takes the resources in the next decision cycle, though, the amount at hand would be excess in terms of quality of service and it would decide to give out resources again. This

causes oscillation near the critical resource amount thresholds. Dynamic delta, however, proposes to fine-grain such resource decisions and partition more resources when there is a huge overflow, and partition much less when there is a little overflow. This doesn't cut out oscillation completely, but provides a pseudo-protection against such oscillations and badly influencing decisions significantly.

Cache, however, does not implement this approach as its own partitioning mechanism is based on singular cache accesses.

3.8. SCHEDULING ALGORITHMS

3.8.1. Round Robin Resource Distribution Algorithm (RR)

As it is widely known, round robin resource distribution is a quite simple one. When the high-priority thread requires additional resources according to the calculations in CAM, these resources are taken away from all the low-priority threads in a round robin fashion one-by-one. It is most likely the fairest algorithm possible since it does not starve any thread for resources. The resource distributions are made in each specific epoch cycle. The pseudocode of this scheme can be examined from Algorithm 1 below:

Algorithm 1: Round Robin

HPT : High-priority thread;

LPT_i : Low-priority thread with the index i;

curQoS : Current Quality of Service of HPT;

expQoS : Expected Quality of Service;

LPTCT : Number of LPTs;

i : Index for selecting desired LPT;

DELTA : A dynamic number of resources going to change hands after each EPOCH time;

```
if curQoS < expQoS then
    i = 1;
    while i < LPTCT do
        if LPT_i resources > DELTA / LPTCT then
            increase HPT Resources by DELTA / LPTCT;
            decrease LPT_i Resources by DELTA / LPTCT;
        end
        i++;
    end
else
    i = 1;
    while i < LPTCT do
        if HPT Resources> DELTA/MAXLPT then
            increase LPT_i resources by DELTA / LPTCT;
            decrease HPT Resources by DELTA / LPTCT;
        end
        i++;
    end
end
```

3.8.2. Utility-Based Resource Distribution Algorithm (UB)

In a multithreaded system, we are presented with various workloads that differ in their resource requirements. Some workloads demand an excessive amount of specific resources while others require fewer resources overall. To optimize resource usage, we introduce the concept of thread utility. Thread utility measures the efficiency of a thread's resource usage, defined as the average instruction throughput per resource entry in an epoch. For instance, if two threads, A and B, have an equal amount of resources, and thread A commits more instructions in a given epoch than thread B, its utility value will be higher.

Our objective in the first utility-based algorithm is to prioritize low-priority threads with high utility and safeguard their resources. We begin by calculating the utility percentage of each low-priority thread and use this percentage value to determine the number of resources to allocate or withdraw from each thread. When the high-priority thread's observed QoS value falls below the expected QoS value, it takes resources from the low-priority threads. Conversely, if the high-priority thread's QoS value meets or exceeds the expected value, it returns resources to the low-priority threads. In order to achieve so, we utilize some additional internal registers such as utility percentage and reverse utility percentage.

The primary difference between this algorithm and round robin scheduling is the calculation of the variable amount of resource entries allocated or withdrawn from each low-priority thread. The amount of resources taken from low-priority threads is determined by a reduced utility percentage value. As a result, a low-priority thread with the highest utility gives the least amount of resources, whereas a low-priority thread with the lowest utility gives the most resources to the high-priority thread. Conversely, when the high-priority thread returns resources, the low-priority thread with the highest utility receives the most resources, while the low-priority thread with the lowest utility receives the least. In this second scenario, the resources allocated to each low-priority thread are determined by its utility percentage value. The pseudocode for this approach can be examined from Algorithm 2 below.

Algorithm 2: Utility-Based

HPT : High-priority thread;

LPT_i : Low-priority thread with the index i;

curQoS : Current Quality of Service of HPT;

expQoS : Expected Quality of Service;

LPTCT : Number of LPTs;

i : Index for selecting desired LPT;

DELTA : A dynamic number of resources going to change hands after each EPOCH time;

utility[i] : Throughput of LPT_i / resources of LPT_i;

utilityPercentage[i] : utility[i] / Sum of all Utilities;

reverseUtilityPercentage[i] : (1 - utilityPercentage[i]) / 2;

if *curQoS* < *expQoS* **then**
    i = 1;
    while *i* < *LPTCT* **do**
        if *LPT_i resources* > *DELTA* ∗ *reverseUtilityPercentage[i]* **then**
            increase HPT resources by DELTA ∗ reverseUtilityPercentage[i];
            decrease LPT_i resources by DELTA ∗ reverseUtilityPercentage[i];
        end
        i++;
    end
else
    i = 1;
    while *i* < *LPTCT* **do**
        if *HPT resources* > *DELTA* ∗ *utilityPercentage[i]* **then**
            increase LPT_i resources by DELTA ∗ utilityPercentage[i];
            decrease HPT resources by DELTA ∗ utilityPercentage[i];
        end
        i++;
    end
end

3.8.3. Reduced Utility-Based Resource Distribution Algorithm 2 (RUB)

While the utility-based distribution algorithm is effective in safeguarding low-priority threads with high resource utility from losing resources, it comes with additional hardware and computation time costs associated with calculating utility, utility percentage, and reduced utility percentage values. To address these challenges, we introduce a reduced utility-based algorithm that simplifies the first algorithm and reduces both its hardware and latency costs. In this approach, we eliminate the reduced utility percentage structure and its serialized calculation latency, while retaining the remaining logic intact. The resulting algorithm fetches resources from the most utilized thread instead of the least utilized thread, protecting the threads in the worst-already conditions.

Utility-based and reduced utility-based approaches are almost identical besides a single subtraction operation. However, this simple difference actually presents a huge change in design philosophy. If we were to think with common sense only, removing resources from a thread that is highly utilizing its provided resources would sound like a pretty bad idea since we would hurt throughput significantly. That is the primary aim lying behind the utility-based partitioning model. On the other hand, if we were to protect those highly utilized threads, we may end up severely starving the threads with lower utilization by always fetching resources from them, sticking the boot in. This may end up severely sacrificing lowly utilized workloads' throughputs. Besides, since we don't have a substitution mechanism for HPT fetches, HPT may end up not being able to obtain enough resources from the lowly utilized threads, causing a significant QoS loss. Therefore, these two approaches are completely in two opposite directions only with that simple difference. The final results are later examined regarding the efficacy of these two approaches later in the thesis. The pseudocode provided in Algorithm 3 can be examined for details.

Algorithm 3: Reduced Utility-Based

HPT : High-priority thread;

LPT_i : Low-priority thread with the index i;

curQoS : Current Quality of Service of HPT;

expQoS : Expected Quality of Service;

LPTCT : Number of LPTs;

i : Index for selecting desired LPT;

DELTA : A dynamic number of resources going to change hands after each EPOCH time;

utility[i] : Throughput of LPT_i/ resources of LPT_i ;

utilityPercentage[i] : utility[i] / Sum of all Utilities ;

```
if curQoS < expQoS then
    i = 1;
    while i < LPTCT do
        if LPT_i resources > DELTA * utilityPercentage[i] then
            increase HPT resources by DELTA * utilityPercentage[i];
            decrease LPT_i resources by DELTA * utilityPercentage[i];
        end
        i++;
    end
else
    i = 1;
    while i < LPTCT do
        if HPT resources > DELTA * utilityPercentage[i] then
            increase LPT_i resources by DELTA * utilityPercentage[i];
            decrease HPT resources by DELTA * utilityPercentage[i];
        end
        i++;
    end
end
```

4. RESULTS & ANALYSIS

4.1. EXPERIMENTAL METHODOLOGY

We held our tests using MSim as our system simulator. The simulator configurations used for this study are listed in Table 4.1 below:

Table 4.1. MSim Simulator Configuration

Fast Forwarded Inst. Count	1,000,000
Max. Inst. Count	1,000,000
Inst. Commit Width	8
Re-order Buffer Size	224
Issue Queue Size	96
Load-Store Queue Size	128
L1 Data Cache Set Count	256
L1 Data Cache Block Size	32
L1 Data Cache Associativity	8
L1 Inst. Cache Set Count	256
L1 Inst. Cache Block Size	32
L1 Inst. Cache Associativity	8
L2 Cache Set Count	4096
L2 Cache Block Size	64
L2 Cache Associativity	8

SPEC2006 benchmarks are used on the configuration above to evaluate the proposed designs [31]. A summary of the used benchmarks are provided below [32]:

- bwaves: High pressure and density explosion calculations in several cubic areas. Used on computational fluid dynamics.
- bzip2: Data compression and decompression. Completely executed on memory.
- cactusADM: Einstein evolution equations.
- dealII: An error estimation library for C++.
- gcc: Generates code based on gcc 3.2.
- gobmk: Playing Go and analyzing potential moves. Used in artificial intelligence.
- gromacs: Newtonian equations between millions of particles such as proteins and lipids. Used in molecular dynamics.
- h264: Encodes h264avc video streams.
- hmmer: Sensitive database search in DNA sequences.
- lbm: Lattice Boltzmann Method used for simulating incompressible fluid behavior. Used in computational fluid dynamics.
- mcf: A simulation for optimizing vehicle scheduling in mass public transportation.
- milc: Simulations of four dimensional lattice gauge theory on MIMD parallel machines.
- namd: A NAMD parallel program for simulation of large biomolecular systems.
- omnetpp: Discrete event simulation of a huge Ethernet network.
- povray: A ray tracing benchmark for rendering light sources and illuminated objects.
- sjeng: A chess algorithm which generates trees of viable moves. Used in artificial intelligence.
- specrand: A random number generation algorithm.

- sphinx3: Speech recognition system.
- zeusmp: Computational fluid dynamics code.

In order to obtain a comprehensible test base for our analysis, we held three main types of tests using our proposed designs:

- HPT and all three LPTs running an instance of the same workload, each (SINGLE)
- One workload on HPT, three instances of another workload on all three LPTs (DOUBLE)
- Four different workloads on all four threads (MULTI)

In order to obtain as comparable results as possible with the original QoSMT study, our DOUBLE test set contains the tests held within that paper with the four-core adaptation. For example, bwaves + mcf test in the original paper was held as bwaves in HPT and mcf in all three low-priority threads. Since we had some benchmark issues though, not all the tests could be the same. On the other hand, we used several other SPEC2006 benchmarks to fill in the gap and expand the data set.

Evaluation is held for four different setups:

- A baseline 4-threaded SMT where all threads have equally and statically partitioned resources (SMT-BASE)
- Our proposed QoSMT applied with round-robin scheduling (QoSMT-RR)
- Our proposed QoSMT applied with utility-based scheduling (QoSMT-UB)
- Our proposed QoSMT applied with reduced utility-based scheduling (QoSMT-RUB)

Using the most optimized design implementations for each scheduler scheme, We evaluate our schedulers in five main aspects:

- Quality of Service
- Performance
- Fairness

- Prediction Accuracy
- Hardware Overhead

4.1.1. Quality of Service

The main aspect of our study is quality of service. In our proposed design, we set certain quality of service targets and the system allocates resources to the threads with the aim of achieving the set target. As it will be later observed, we are not always satisfying our QoS targets. The reasons for this behavior are mainly because of our QoS calculations are statistically predicted and cannot be completely accurate, which is a point we will discuss later. On the other hand, we initialize our threads with equal resources in our test conditions. Hence, if the QoS is extremely low in initial epoch cycles, even though the steps would be taken, the average QoS obtained at the end of test workloads may be lower than expected since they are simulated for a limited amount of 1 million instructions, which will have further addressing later.

The quality of service prediction is provided in Equation 4.1:

$$predictiveQoS = \frac{epochCycles - contentionCycles}{epochCycles} \tag{4.1}$$

Where epochCycles is the total cycles passing through each calculation cycle, and contentionCycles is the contention-caused delay prediction value obtained from the delay register. The quality of service target for the implementations discussed below are 90% in terms of predictions if written otherwise.

The actual quality of service calculated offline is given in Equation 4.2 below:

$$actualQoS = \frac{IPC_{qosmtHPT}}{IPC_{singlethread}} \tag{4.2}$$

Where $IPC_{qosmtHPT}$ is the throughput of HPT in QoSMT mode in terms of IPC, and $IPC_{singlethread}$ equals to the IPC of a single thread with all the four QoSMT threads' resources (ROB, IQ, LSQ, cache).

4.1.2. Performance

What this study primarily aims for is to achieve a high degree of quality of service meanwhile enabling the utilization of SMT and core capabilities. A single threaded approach can certainly provide a 100% quality of service. Yet, achieving a respectable degree of quality of service and throughput side by side is the main challenge.

At this point, our approach is quite suitable for achieving such a goal. The reason for this is that dynamically partitioning resources between threads based on utilization unlocks a certain degree of performance improvement. Such utility based resource partitioning schemes in the literature proved this concept successfully as well; as studies of Sharkey et al., Li et al. and many others have shown [33,34].

In order to assess the performances of the proposed designs, we use relative speedup of the proposed designs compared to the baseline-SMT core, which provides the highest performances due to its performance-focused nature through the Equation 4.3 given below.

$$Speedup = \frac{IPC_{qosmtTotal}}{IPC_{smtTotal}} \tag{4.3}$$

4.1.3. Fairness

In this study, we treat fairness as all threads sharing the same resources have an equal amount of resources for each and perform in the same manner. In our proposed designs, we dynamically allocate resources between threads, so, for the calculation of the fairness metric for our proposed configurations, we use the formula given in Equation 4.4 below:

$$Fairness = \frac{4}{\frac{IPC_{smtT0}}{IPC_{qosmtT0}} + \frac{IPC_{smtT1}}{IPC_{qosmtT1}} + \frac{IPC_{smtT2}}{IPC_{qosmtT2}} + \frac{IPC_{smtT3}}{IPC_{qosmtT3}}} \tag{4.4}$$

Which calculates the fairness based on equal resource partitioned SMT threads' performance and the QoSMT threads' performance. Actual fairness could be also taken as the performance compared to a single-threaded design's throughput. Yet, we can consider that a baseline SMT has already a certain sacrifice of fairness that is accepted in its implementation and therefore,

we make our comparisons based on the relative fairness losses and gains with the baseline SMT rather than baseline single-thread implementation.

4.1.4. Prediction Accuracy

As mentioned in Section 4.1.1, our runtime QoS calculations are predictive and may be inaccurate. In order to calculate their accuracy and reliability, we shall assess their error ratios as well. In order to do so, we use mean absolute error calculations onto predicted and offline-calculated actual QoS values through the formula provided in Equation 4.5.

$$PredictionError = \frac{1}{N}\sum_{i=1}^{N} |actualQoS_i - predictiveQoS_i| \tag{4.5}$$

Where N is the number of test cases, actualQoS_i is the actual average QoS for the given test scenario, and predictiveQoS_i is the predicted average QoS for the given test scenario.

4.1.5. Hardware Overhead

One of the most crucial factors to consider meanwhile implementing microarchitectural optimizations for a core is the hardware overhead. Even though a proposed implementation improves a design, it most likely requires some changes on circuitry compared to the base design and this may cause the transistor count to increase. Maybe, an optimization would require such a huge amount of additional transistors that simply adding SRAM to the core worth that amount of transistors could improve performance more than the proposed design. Therefore, we shall analyse the hardware cost of our proposed design as well.

In order to obtain transistor cost of our proposed designs, we utilized YoSys 0.9.0 and UC Berkeley's ABC: System for Sequential Logic Synthesis and Formal Verification using CMOS technology [35][36].

4.2. PRE-EVALUATION DESIGN OPTIMIZATION

During the previous sections, we have discussed different implementation details that may have varying impact on our tested designs. In order to obtain the best implementation for each of the proposed schedulers, we made some tests with varying parameters. These tests are:

- Dynamic/Static Delta Evaluation
- Delta Parameter Variation Evaluation
- Decision Time Parameter Variation Evaluation

In order to check the design optimizations, we choose two random tests from each of the test groups mentioned above (2x SINGLE, 2x DOUBLE, 2x MULTI) and made our configuration evaluations based on these results. These preliminary evaluations were made based on QoS, speedup, and fairness.

When there is only a single benchmark name in the following figures, it means that all four workloads on the threads are instances of the workload provided below (bzip2 -> bzip2_bzip2_bzip2_bzip2). When there are two benchmark names, on the other hand, means that the HPT is running the first given workload and all three LPTs are running instances of the second benchmark (bzip2_sjeng -> bzip2_sjeng_sjeng_sjeng). Also note that there are some almost non-existent bar values within the figures provided below. They are not zero, but almost-zero values that face issues when reduced in size to fit into the page format.

4.2.1. Dynamic/Static Delta Evaluation

Dynamic delta is one of the primary implementation changes we proposed to the Jin et al.'s original study. However, since we are working with four threads and not two, we shall evaluate this idea's efficacy first before any testing.

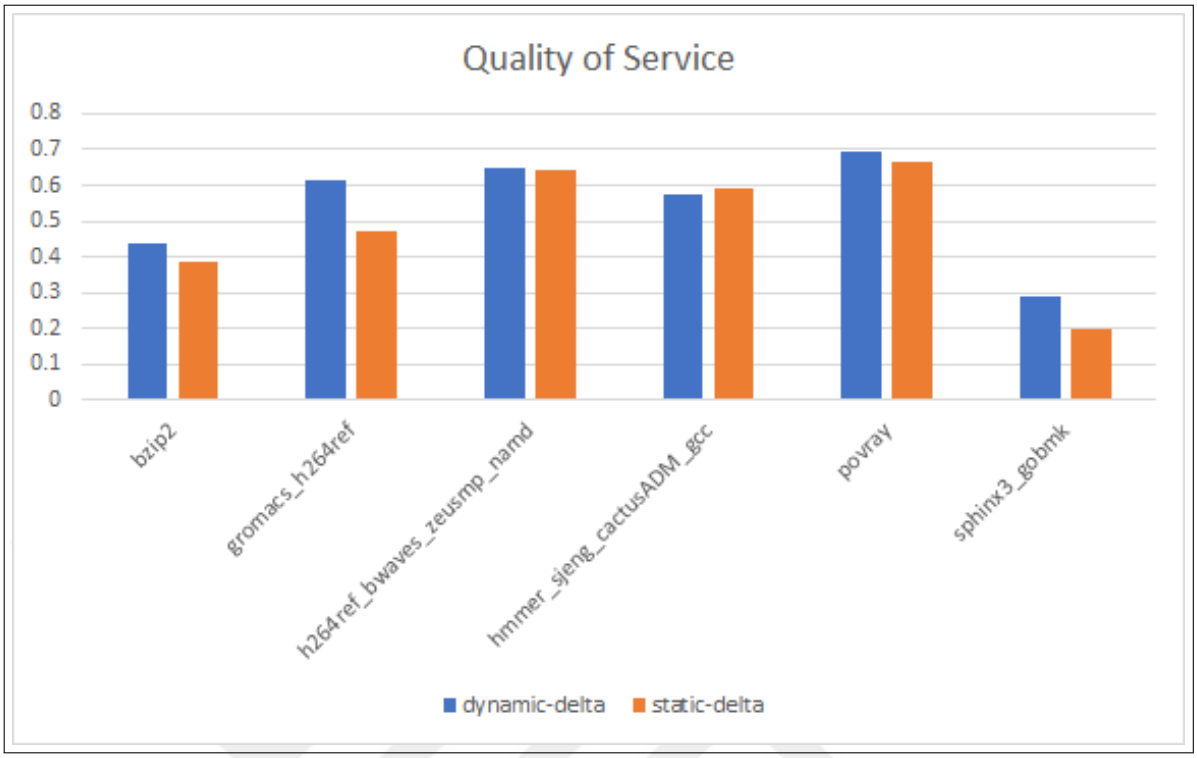

Figure 4.1. Delta configuration offline QoS comparison based on single-thread throughput (RR)

Based on Figure 4.1, we can observe that dynamic delta provides a better quality of service in five of the six tested scenarios. Which makes it far more superior to the static delta used in the original study for the round-robin scheduler in terms of our primary goal.

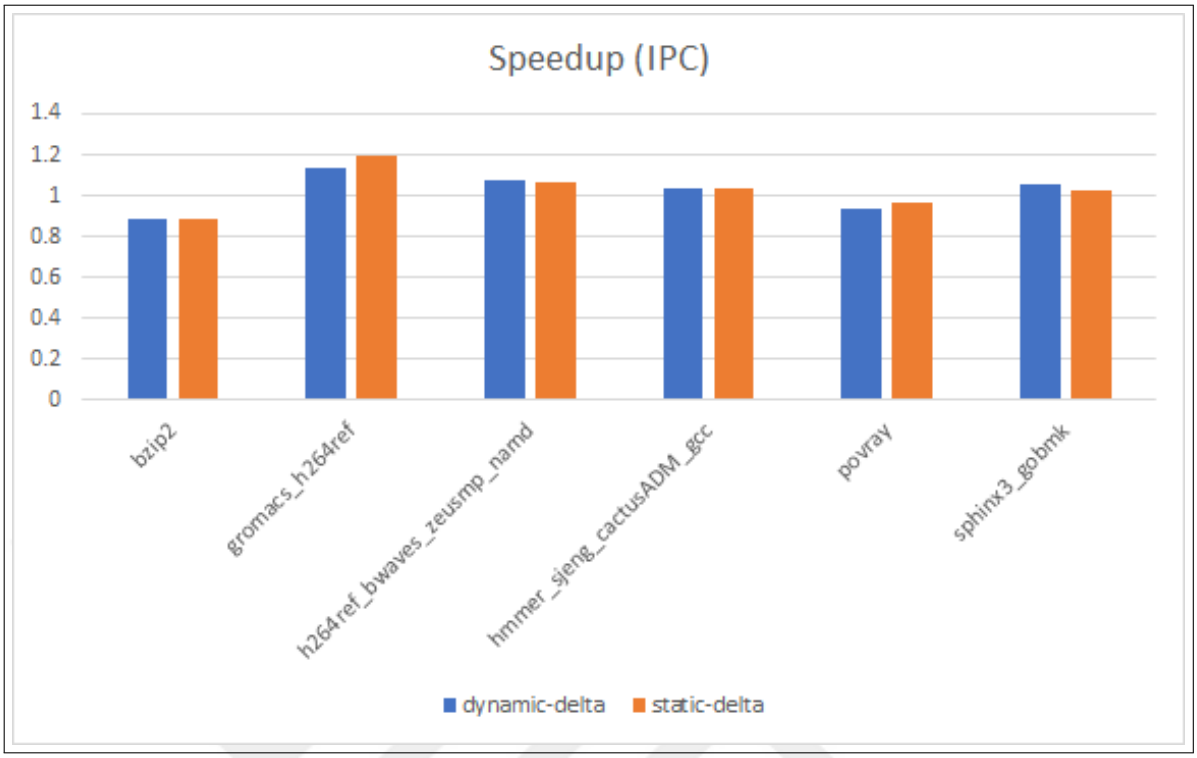

Figure 4.2. Delta configuration offline speedup comparison based on baseline-SMT performance (RR)

When we examine Figure 4.2, we observe that in four of the six scenarios, performance in terms of IPC is increased with the dynamic delta, which further incentivizes the use of dynamic delta compared to the static delta.

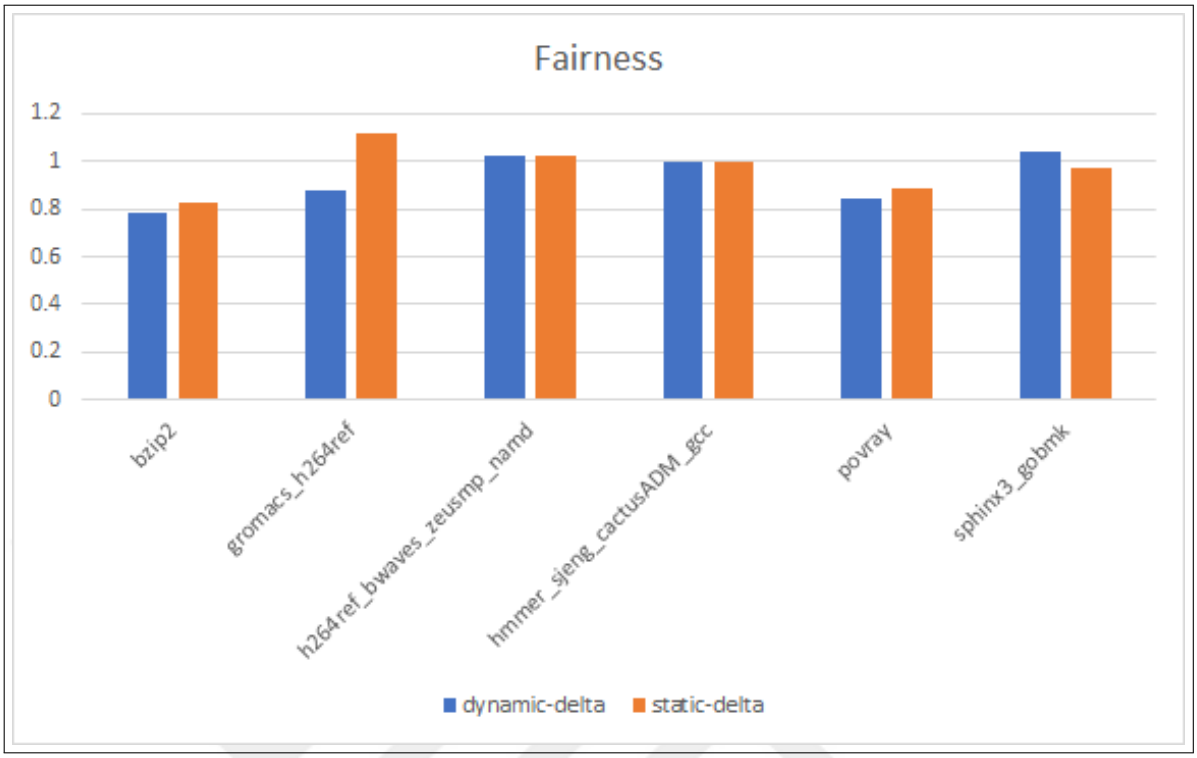

Figure 4.3. Delta configuration offline fairness comparison based on baseline-SMT performance (RR)

In terms of fairness, we observe that only two of the test scenarios showcase a better fairness value when a dynamic delta value is used, as shown in Figure 4.3. Our proposed design aims to provide priority to our HPT; therefore, the nature of our implementation is unfair. However, losing only 5% fairness compared to the static delta, and pertaining a 92.99% average fairness value in this design still showcases a strong performance and fairness trend.

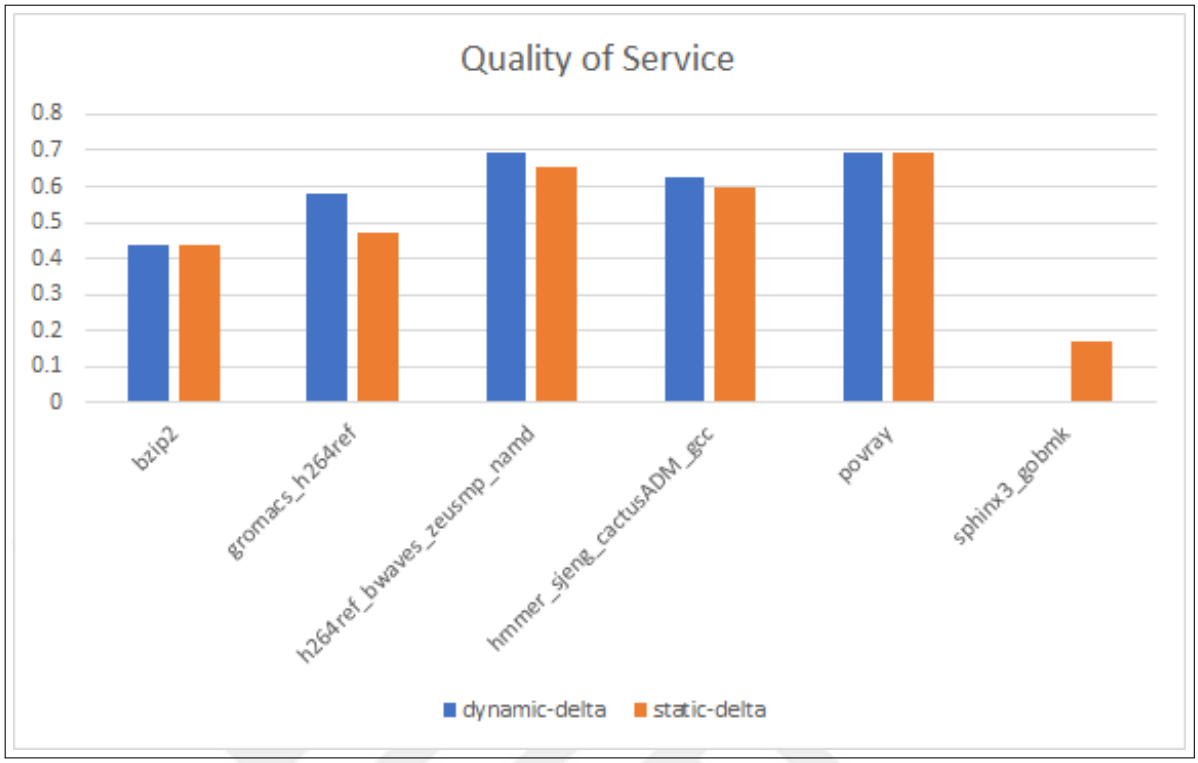

Figure 4.4. Delta configuration offline QoS comparison based on single-thread throughput (UB)

According to Figure 4.4, the quality of service displays a similar trend with its round-robin counterpart, scoring higher quality of service percentages in five out of six test scenarios.

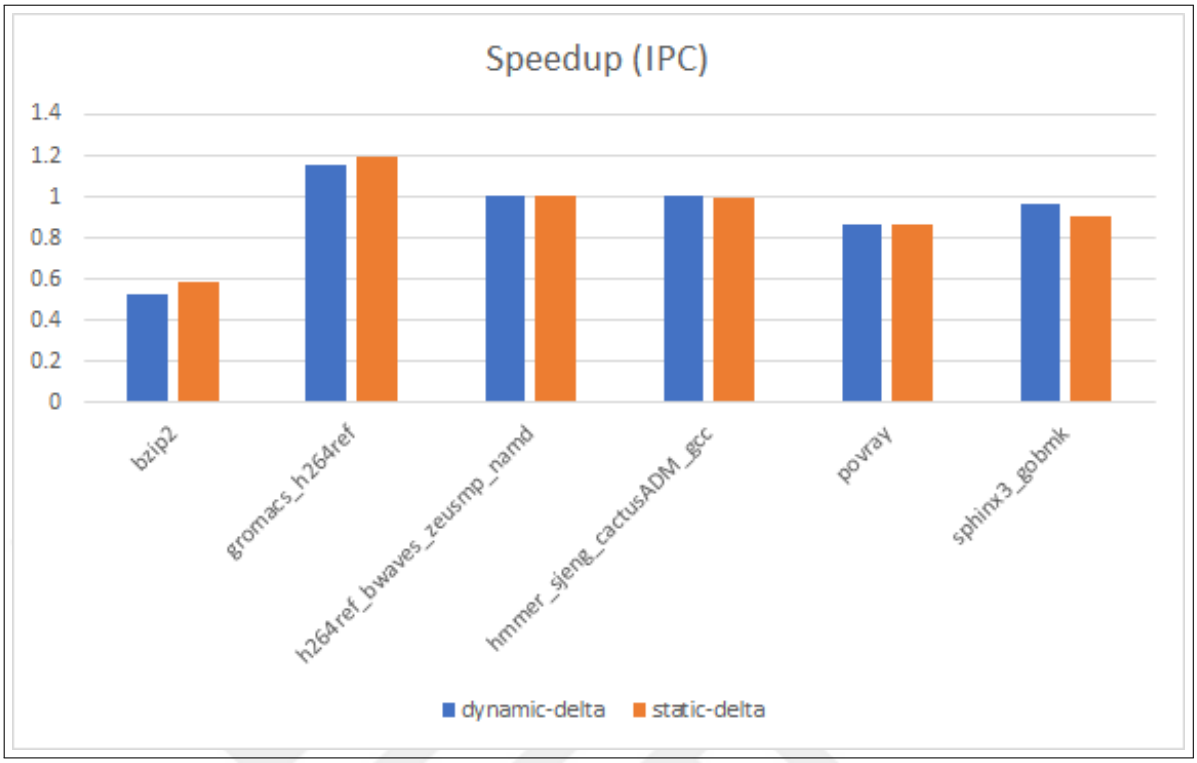

Figure 4.5. Delta configuration offline speedup comparison based on baseline-SMT performance (UB)

Based on Figure 4.5, speedup has quite a similar trend with its round-robin counterpart as well, scoring higher IPC-based speedups in four our of six test scenarios.

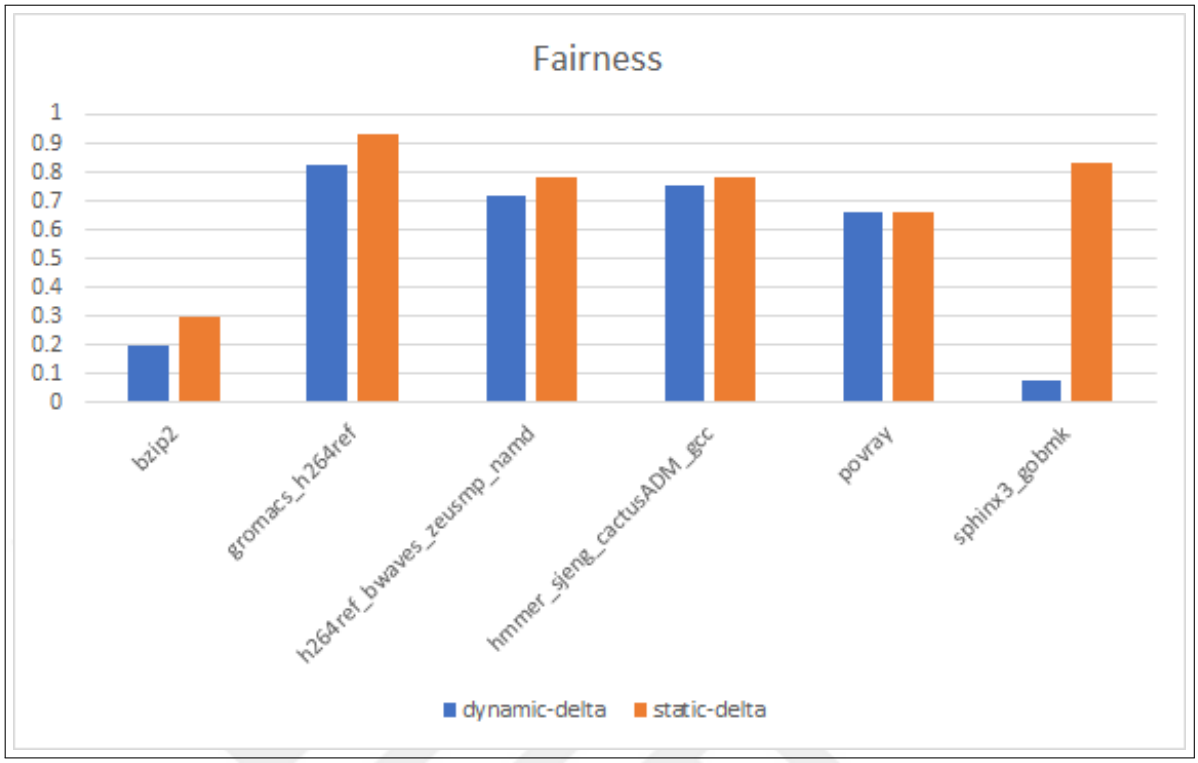

Figure 4.6. Delta configuration offline fairness comparison based on baseline-SMT performance (UB)

When we examine Figure 4.6, we observe that fairness is reduces much more significantly than its round-robin counterpart. This is most likely due to the HPT-prioritizing utility based nature of the Utility-Based scheduler, which may have a more drastic effect on fairness to provide HPT-focused priority.

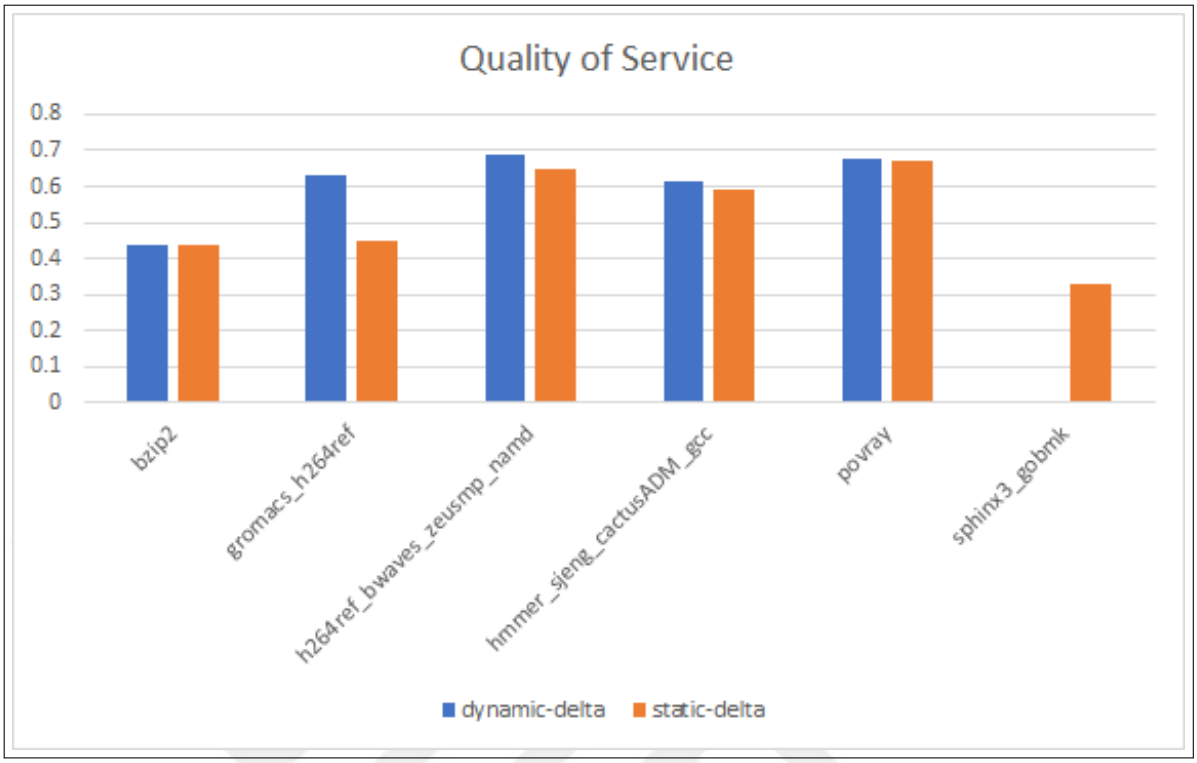

Figure 4.7. Delta configuration offline QoS comparison based on single-thread throughput (RUB)

Figure 4.7 completes the picture in terms of QoS that dynamic delta is far more superior compared to the static delta with a five out of six advantage. With this picture, we can argue that the last test case, sphinx3 on the HPT and gobmk on the three LPTs display anomalous results, which is most likely due to gobmk's cache-intensive extreme resource demand.

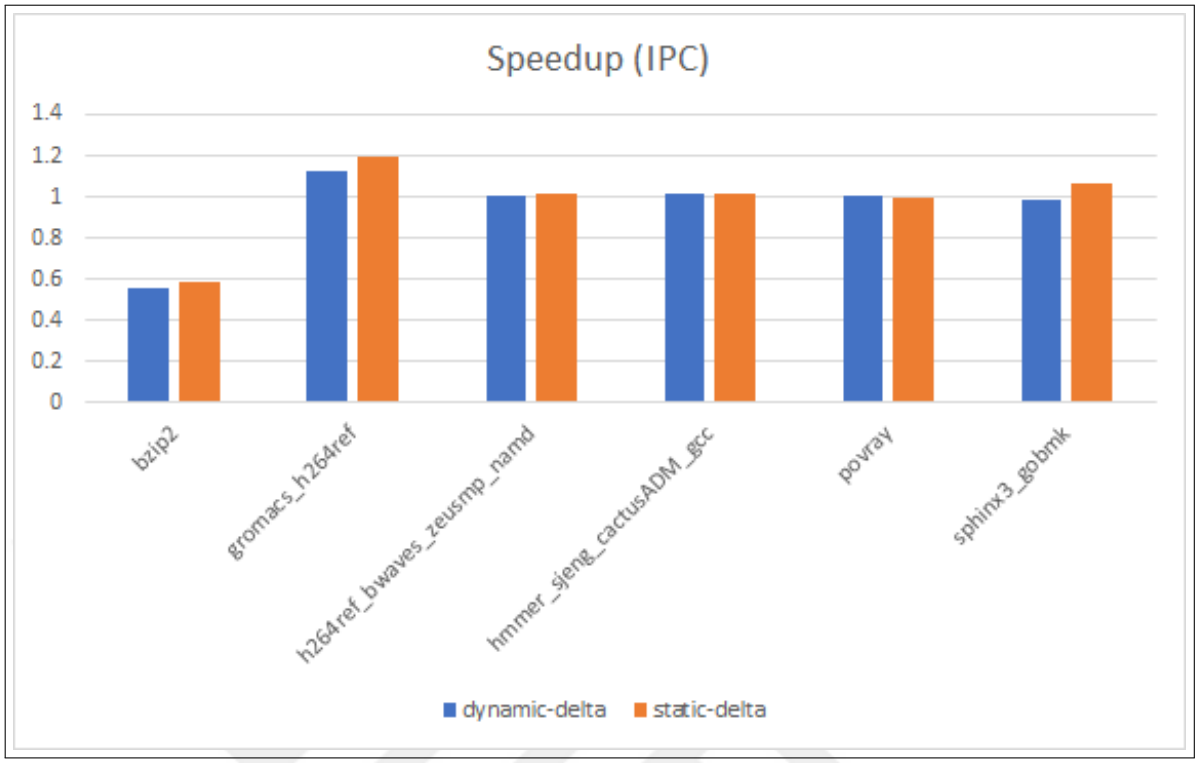

Figure 4.8. Delta configuration offline speedup comparison based on baseline-SMT performance (RUB)

In terms of throughput, for Reduced Utility-Based scheduler falls prone to more speedup loss compared to its RR and UB counterparts with only two out of six performance advantage as it can be observed from Figure 4.8. However, the average speedup reducing from 97.76% to 94.80% is not much significant for us to sacrifice the quality of service we gain through dynamic delta.

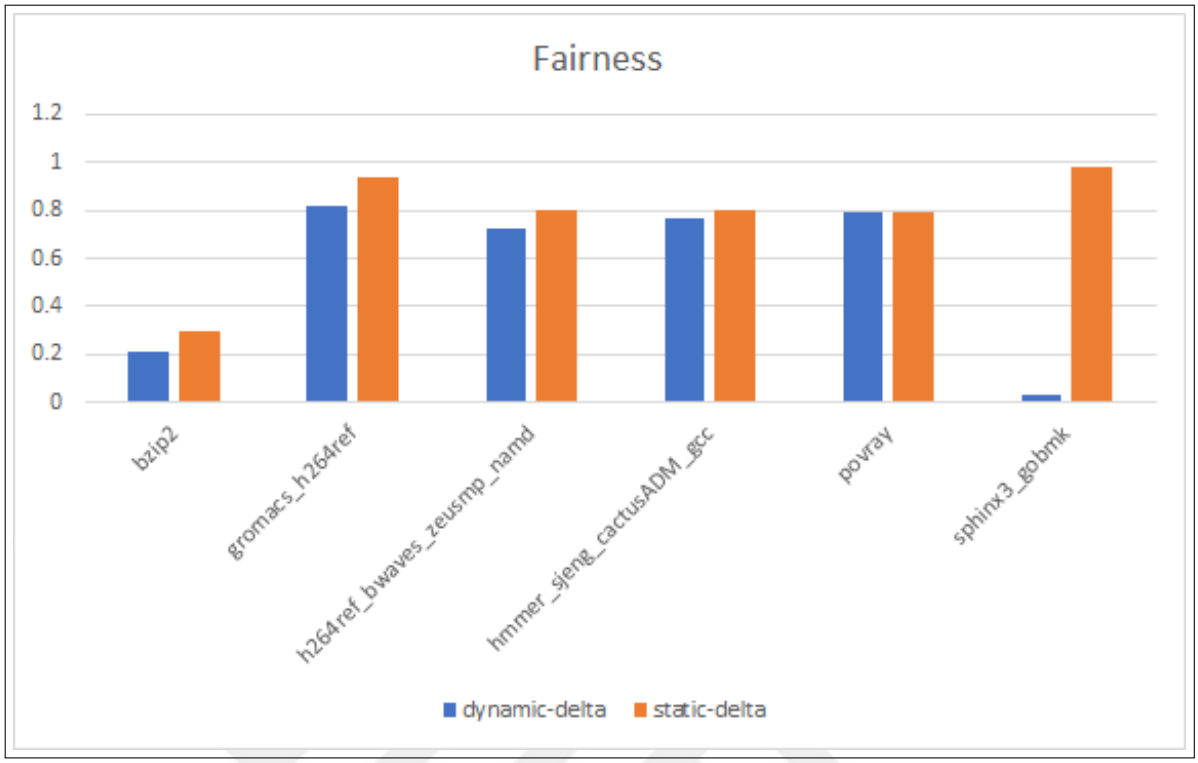

Figure 4.9. Delta configuration offline fairness comparison based on baseline-SMT performance (RUB)

We observe that dynamic delta causes a significant fairness loss such as RR and UB schedulers as well, as it can be seen from Figure 4.9.

In the light of all these observations, we decide to use dynamic delta for all three scheduler implementations: RR, UB and RUB.

4.2.2. Delta Parameter Variation Evaluation

Throughout the design examination, the delta amount of resource switches are discussed. However, this base delta amount certainly have an impact on the design efficiency as well. In order to assess the optimal delta resource amount of the schedulers, we evaluate three different base delta setups with the dynamic delta. These setups are:

- 4 ROB, 2 LSQ, 1 IQ (half-delta)
- 8 ROB, 4 LSQ, 2 IQ (base-delta)

- 16 ROB, 8 LSQ, 4 IQ (double-delta)

The naming convention being half, base, and double is because of the preliminary work on this thesis, which is a paper submitted and under review utilizes base-delta amount as its optimal setup. However, the newly proposed implementations in this thesis may have an affect on the optimal setup. Therefore, we reassess our base-delta amount to be used with dynamic delta in this thesis.

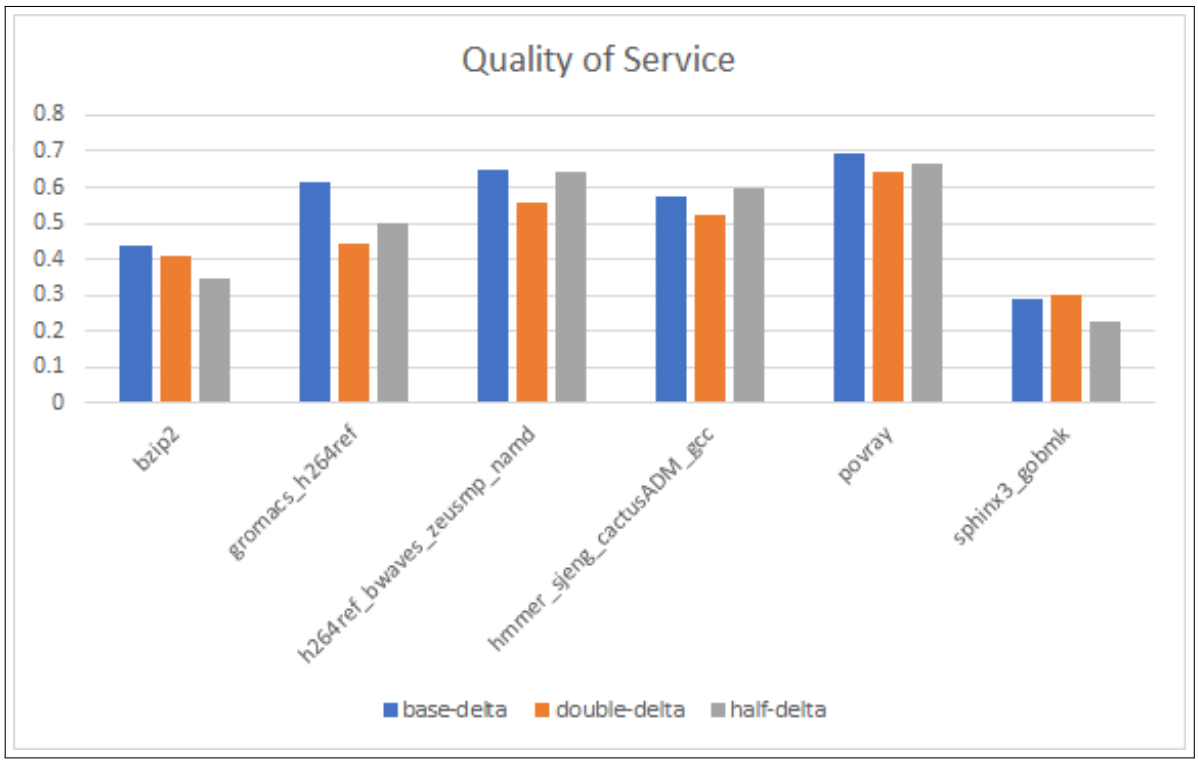

Figure 4.10. Delta multiplier offline QoS comparison based on single-thread throughput (RR)

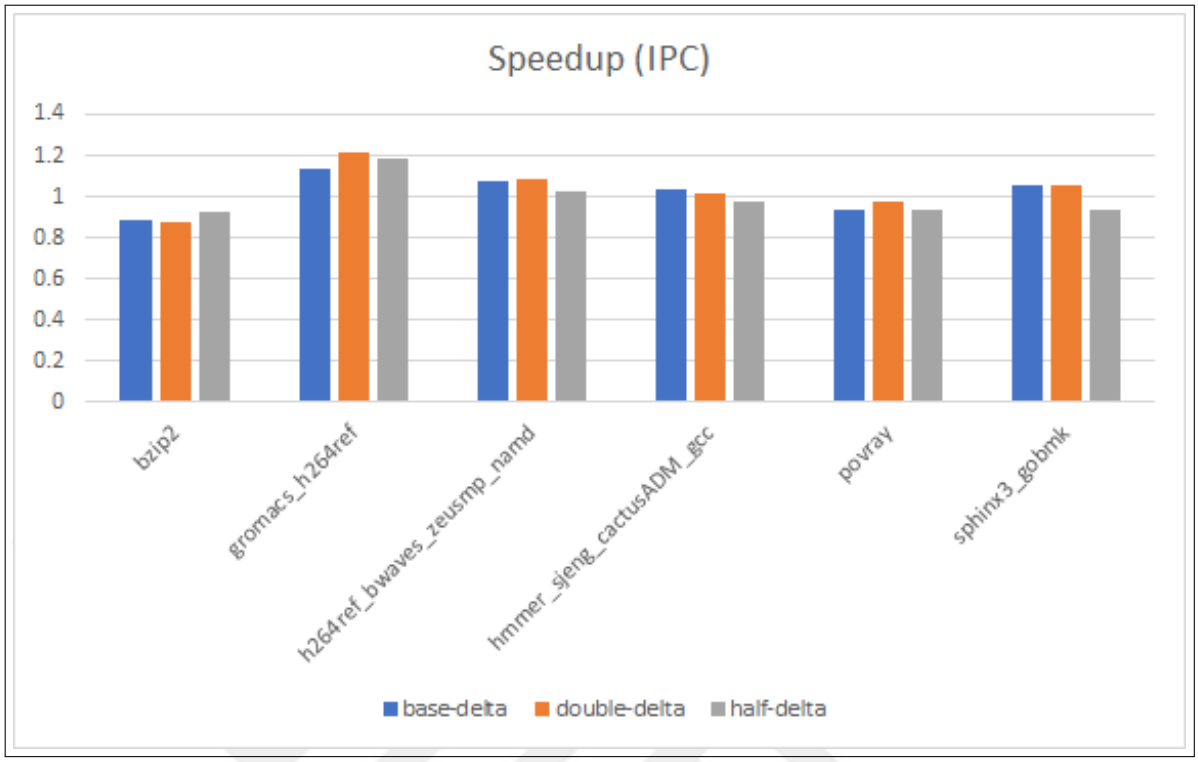

Figure 4.11. Delta multiplier offline speedup comparison based on baseline-SMT performance (RR)

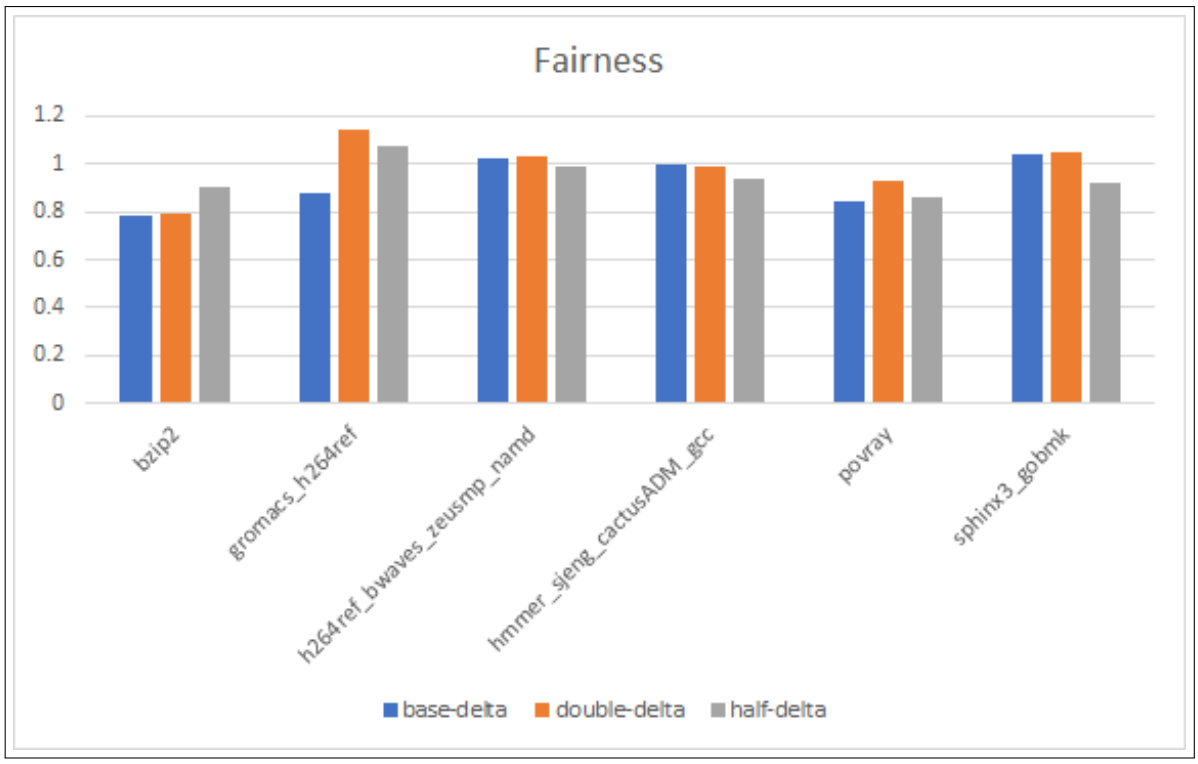

Figure 4.12. Delta multiplier offline fairness comparison based on baseline-SMT performance (RR)

Based on the Figures 4.10, 4.11, and 4.12; base-delta provides the best QoS values with four best, two-second best QoS values between the three multipliers meanwhile providing relatively strong speedup and fairness percentages. These results lead us to use base-delta amount for the Round-Robin scheduler.

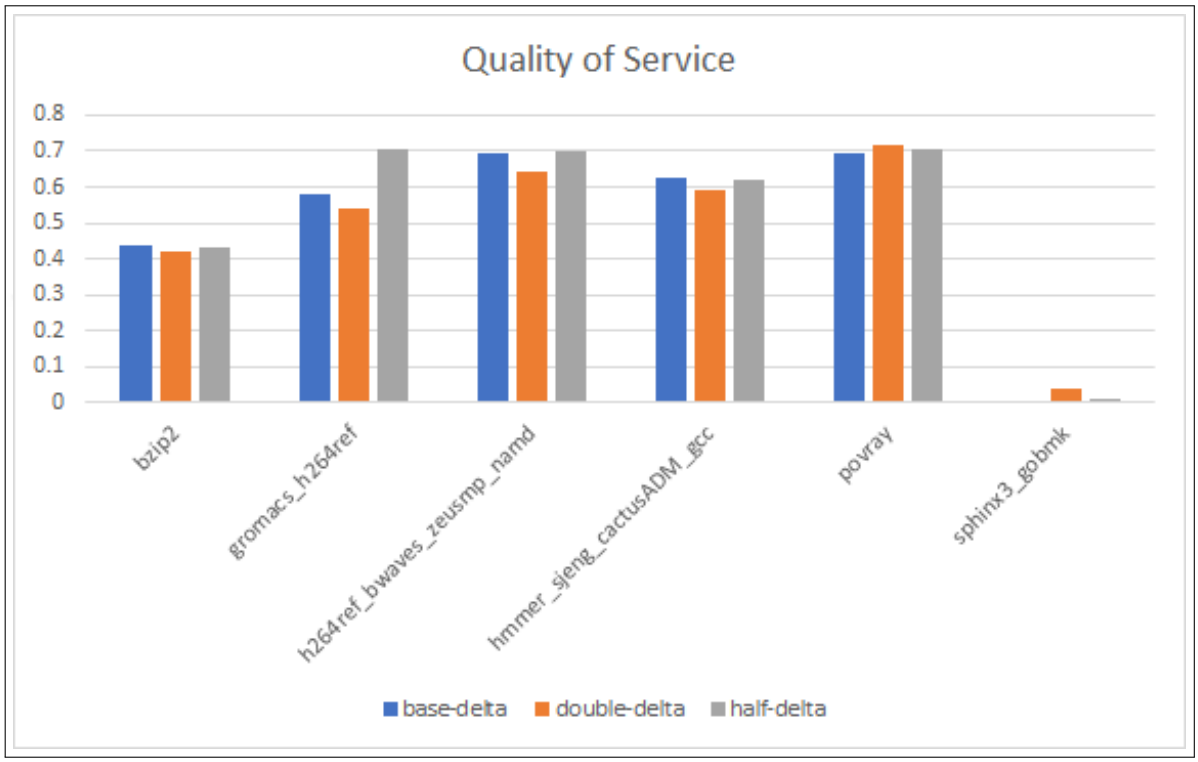

Figure 4.13. Delta multiplier offline QoS comparison based on single-thread throughput (UB)

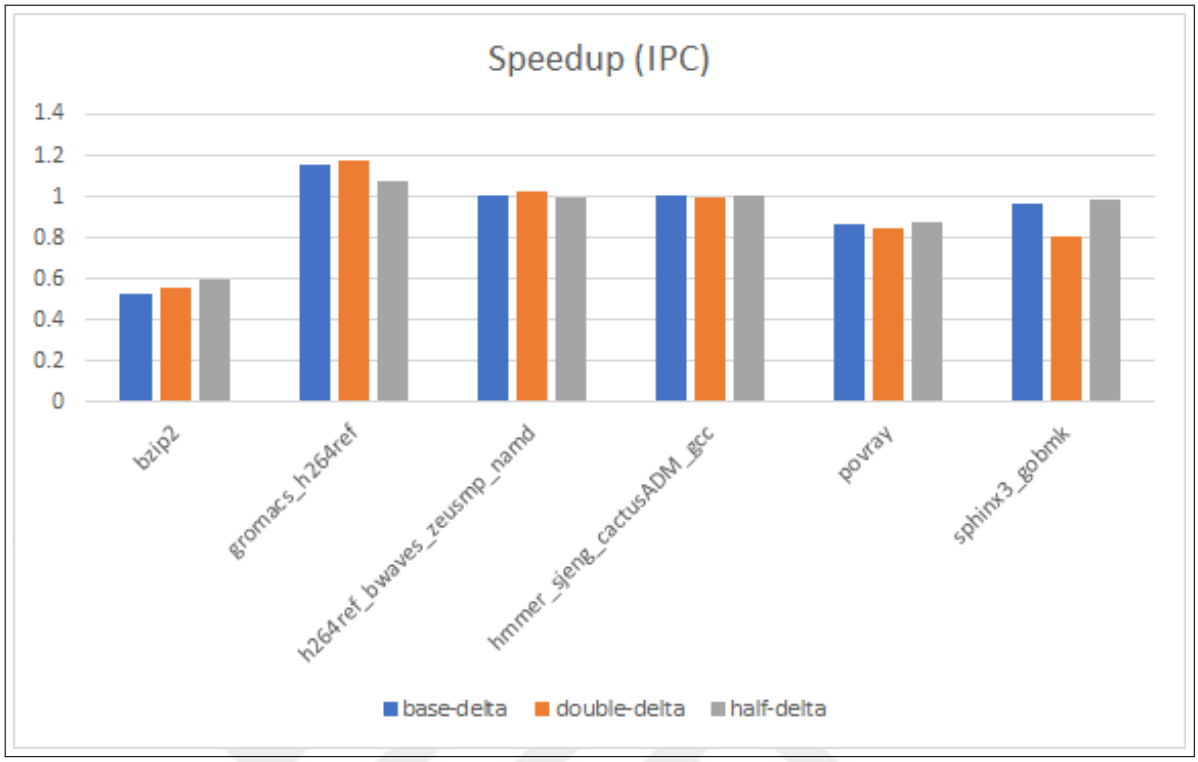

Figure 4.14. Delta multiplier offline speedup comparison based on baseline-SMT performance (UB)

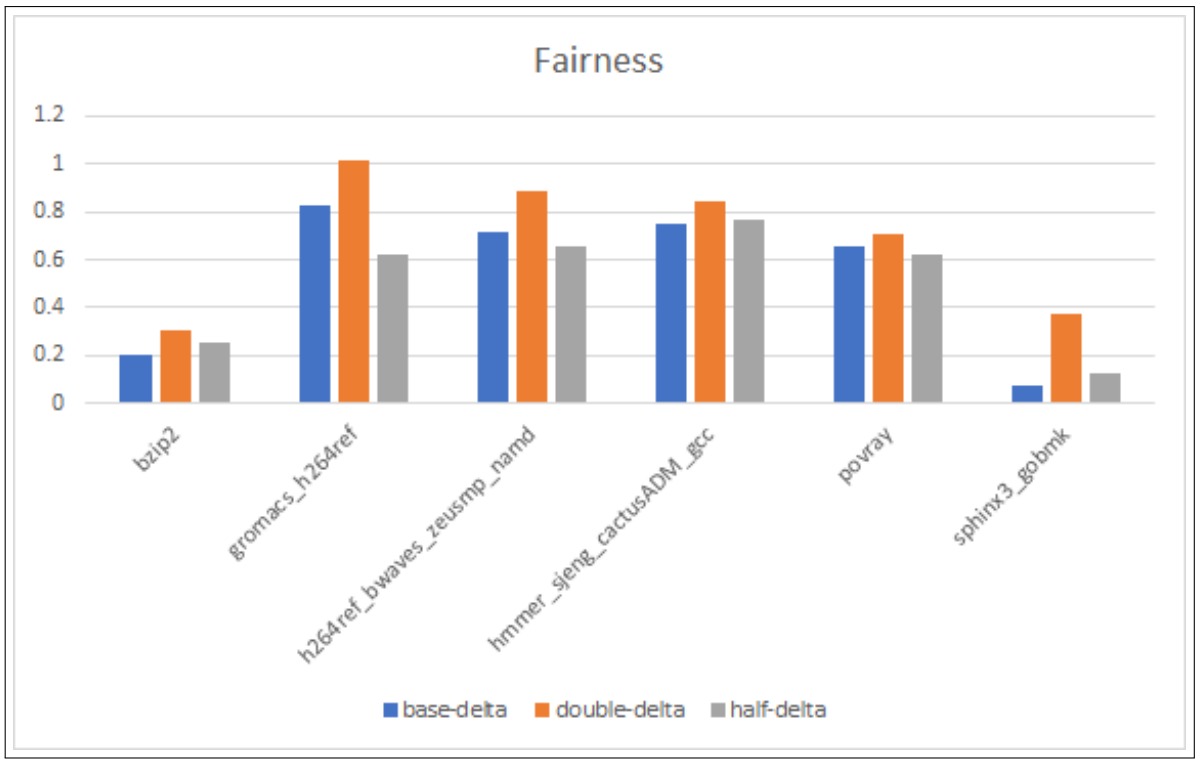

Figure 4.15. Delta multiplier offline fairness comparison based on baseline-SMT performance (UB)

Considering the Figures 4.13, 4.14, and 4.15; the quality of service values are closer between three delta multipliers. However, half-delta amount provides the highest average QoS percentage of 52.94% between the three. It also provides the best average speedup value of 92.18% meanwhile lacking in terms of fairness. However, due to its advantages in terms of QoS and speedup, we decided to use half-delta for the Utility-Based scheduler. The reason UB scheduler using half-delta unlike the RR scheduler is most likely due to utility-based schedulers tend to starve threads easier than the RR counterpart, which means that faster starvation may cause more performance loss to the overall core and less resources provided to the HPT.

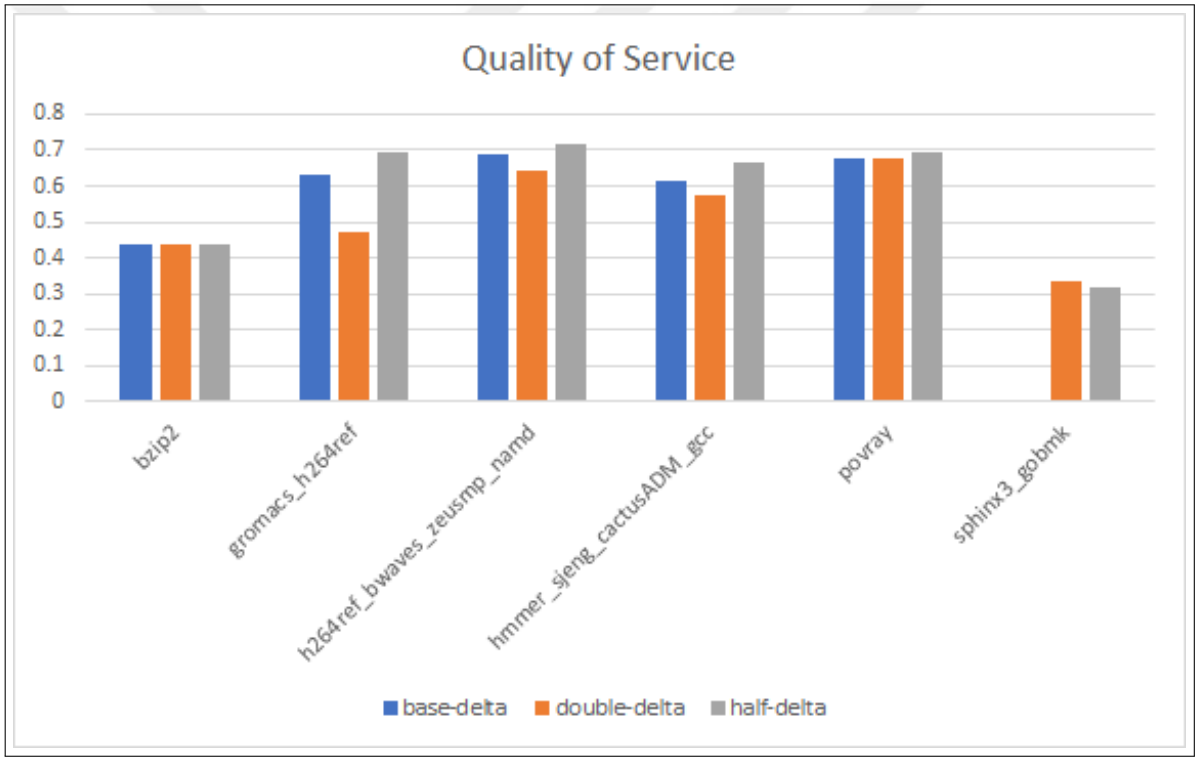

Figure 4.16. Delta multiplier offline QoS comparison based on single-thread throughput (RUB)

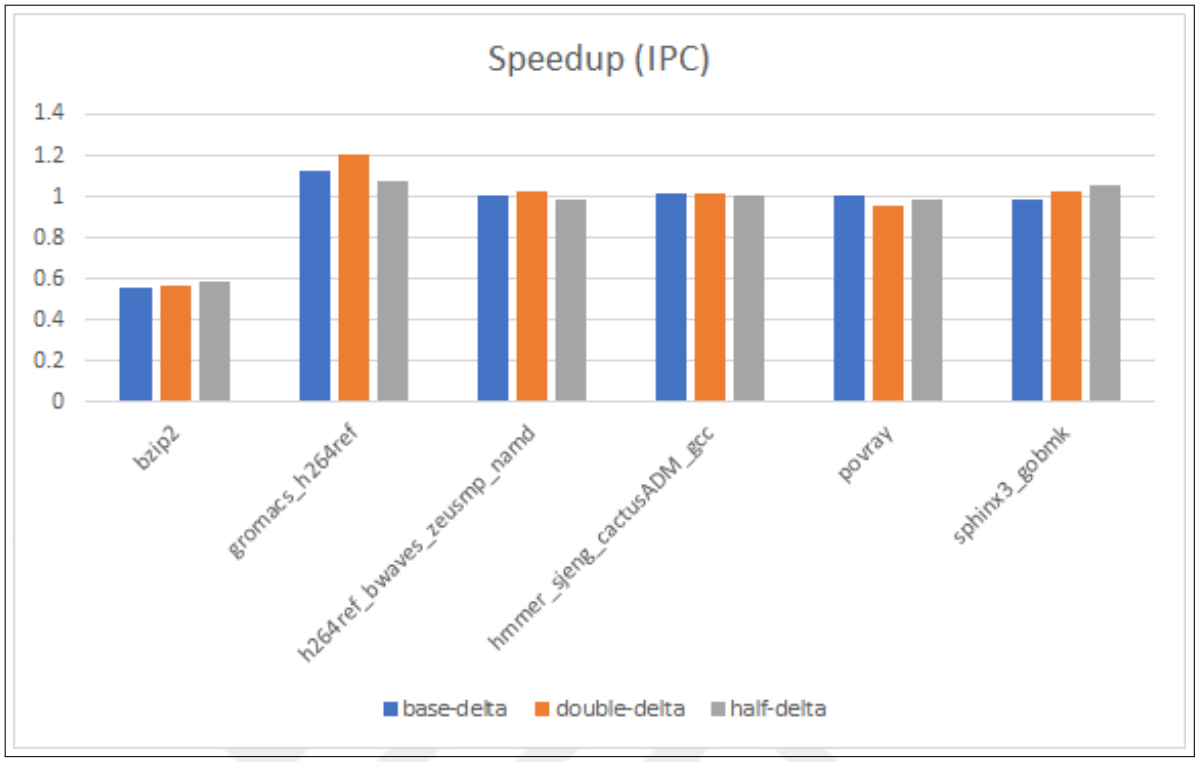

Figure 4.17. Delta multiplier offline speedup comparison based on baseline-SMT performance (RUB)

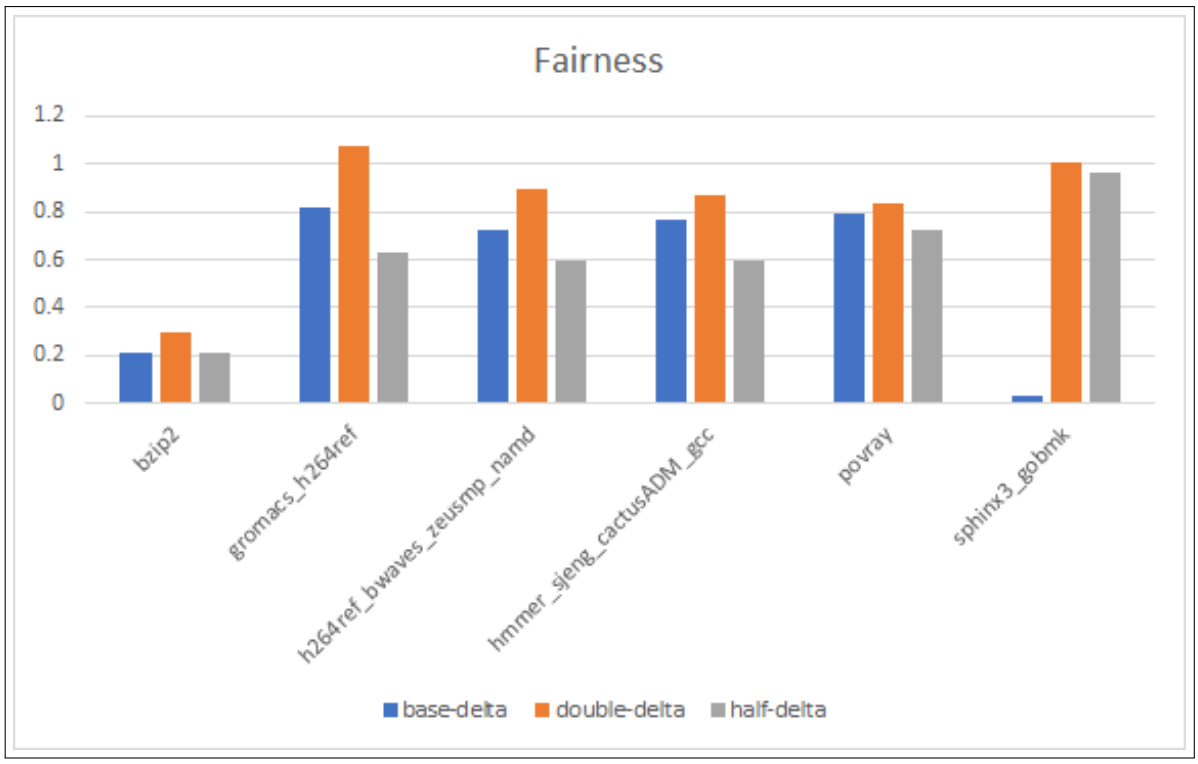

Figure 4.18. Delta multiplier offline fairness comparison based on baseline-SMT performance (RUB)

The trends of the Reduced Utility-Based scheduler regarding the delta amount is quite similar to the Utility-Based scheduler as it can be observed from Figures 4.16, 4.17, and 4.18. The difference from the round-robin towards the half-delta tendency is shared as well due to utilization focus' tendencies. So, we decided to use half-delta as the delta multiplier for the RUB scheduler as well.

4.2.3. Decision Time Parameter Variation Evaluation

Decision time, previously discussed as epoch cycle through the paper is another metric that has a significant impact towards the overall efficacy of the implementations. The decision time also affects the hardware overhead of the design because of the delay vectors hold in the baseline study. However, in our linearized approach which converts delay vector into a delay register disregards this change and decision time parameter variation affects only the implementation success. We evaluate three different decision time cycles as well, such as the delta parameters, which are given below:

- 10,000 cycles (half-epoch)
- 20,000 cycles (base-epoch)
- 40,000 cycles (double-epoch)

The reason we name these configurations as half, base, and double is the same as the delta parameter configuration naming as well, the origin of this study which utilizes 20,000 cycles of epoch time.

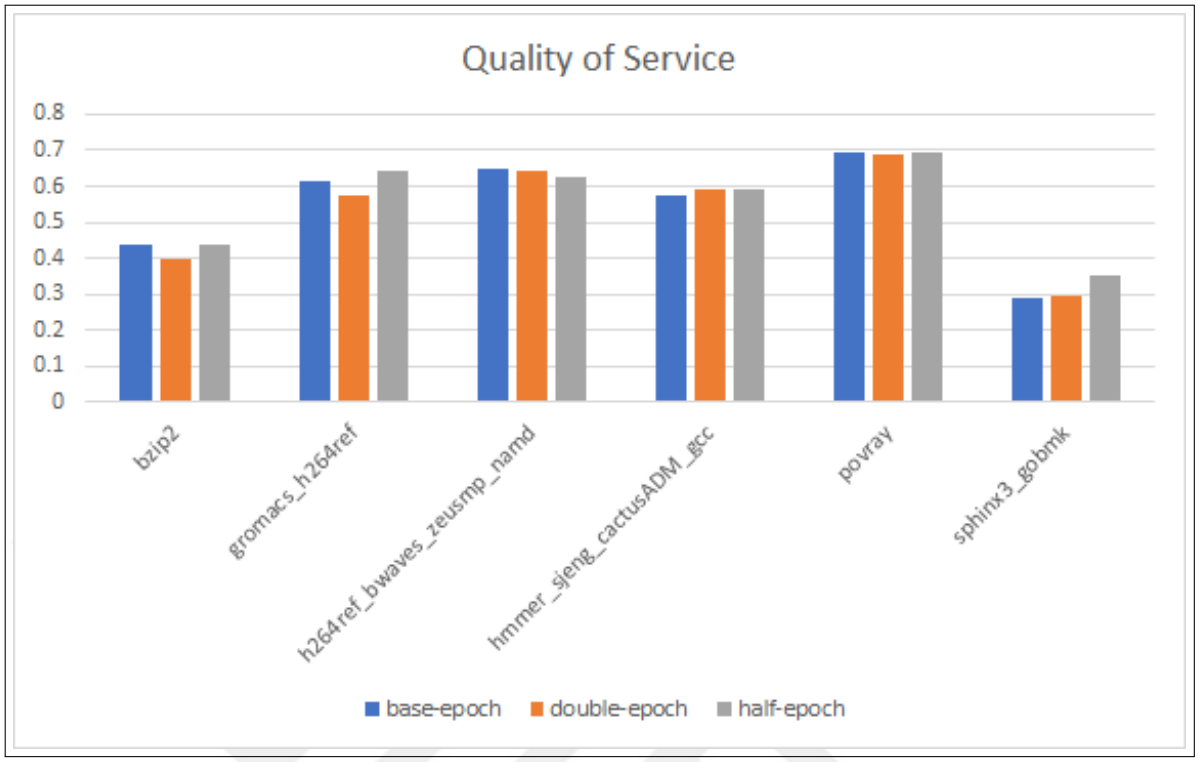

Figure 4.19. Epoch multiplier offline QoS comparison based on single-thread throughput (RR)

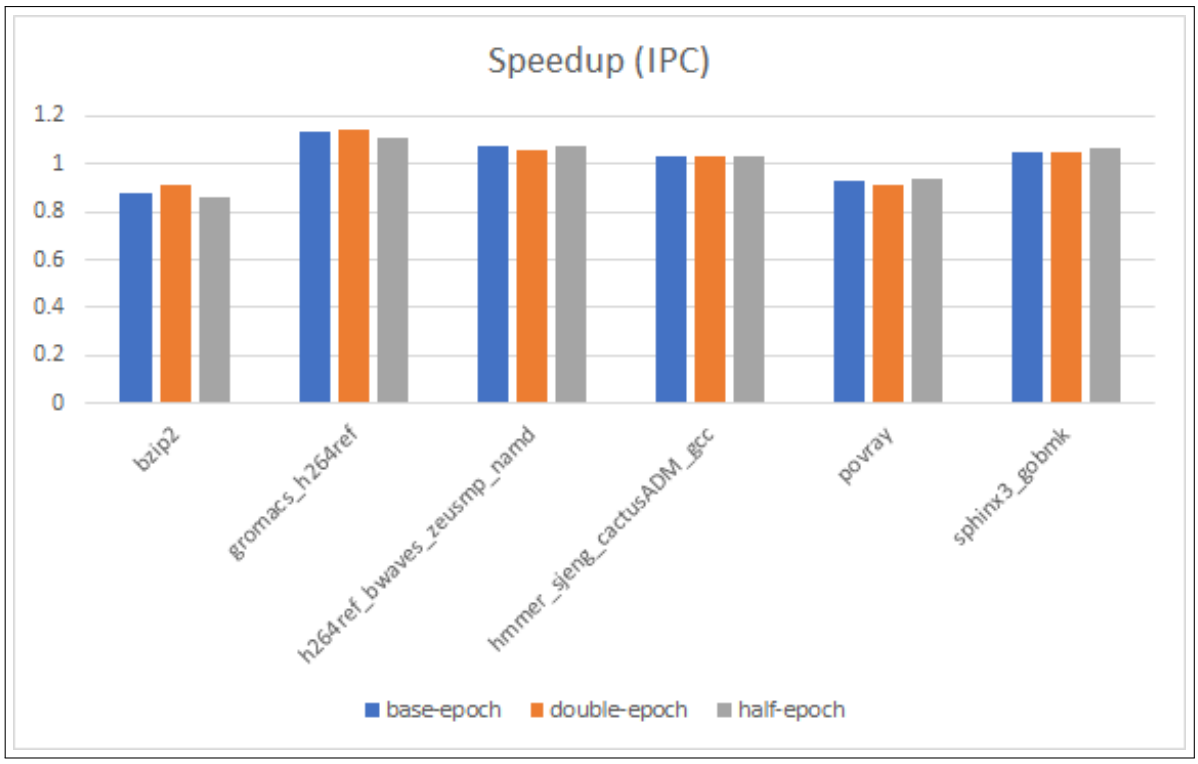

Figure 4.20. Epoch multiplier offline speedup comparison based on baseline-SMT performance (RR)

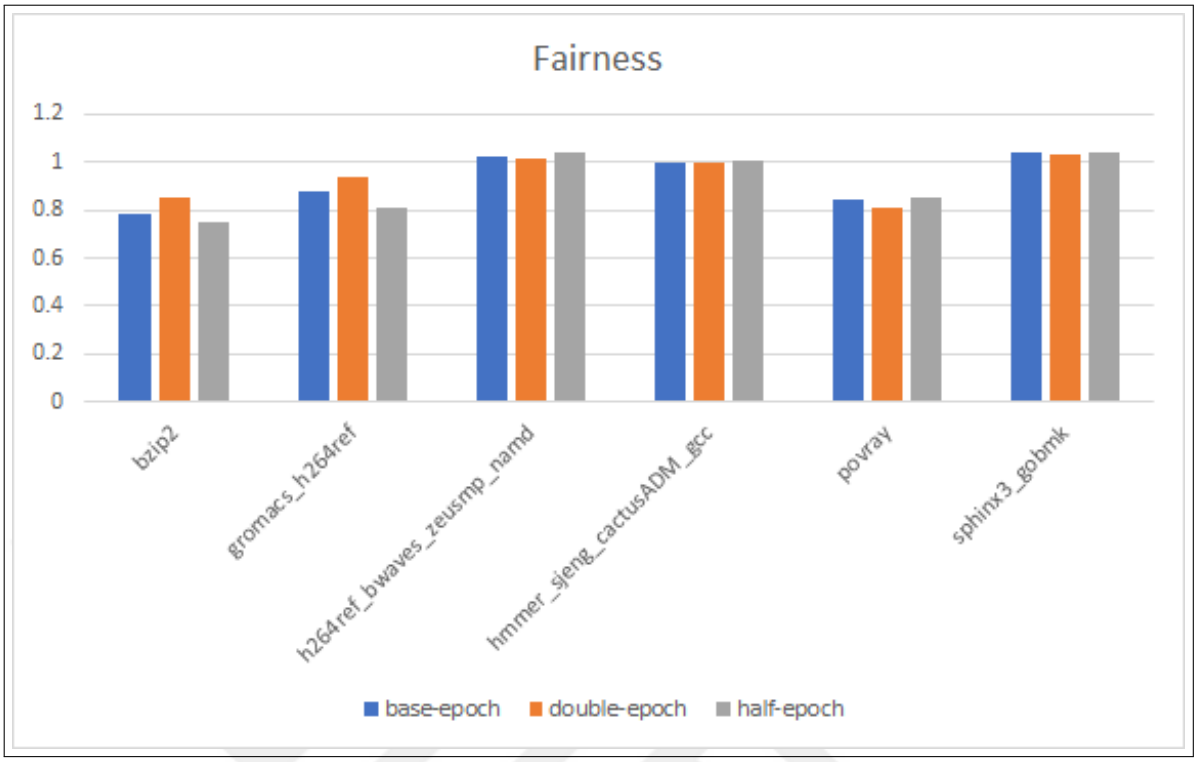

Figure 4.21. Epoch multiplier offline fairness comparison based on baseline-SMT performance (RR)

Based on the Figures 4.19, 4.20, and 4.21; the highest performing epoch time interval is the half-epoch, resulting in 55.61% average QoS percentage meanwhile providing an almost identical average speedup of 1.41% IPC increase with the other epoch multipliers but provide the worst fairness percentage of the three with 91.70%. Considering these results, we decide to use the half-epoch configuration as our decision interval.

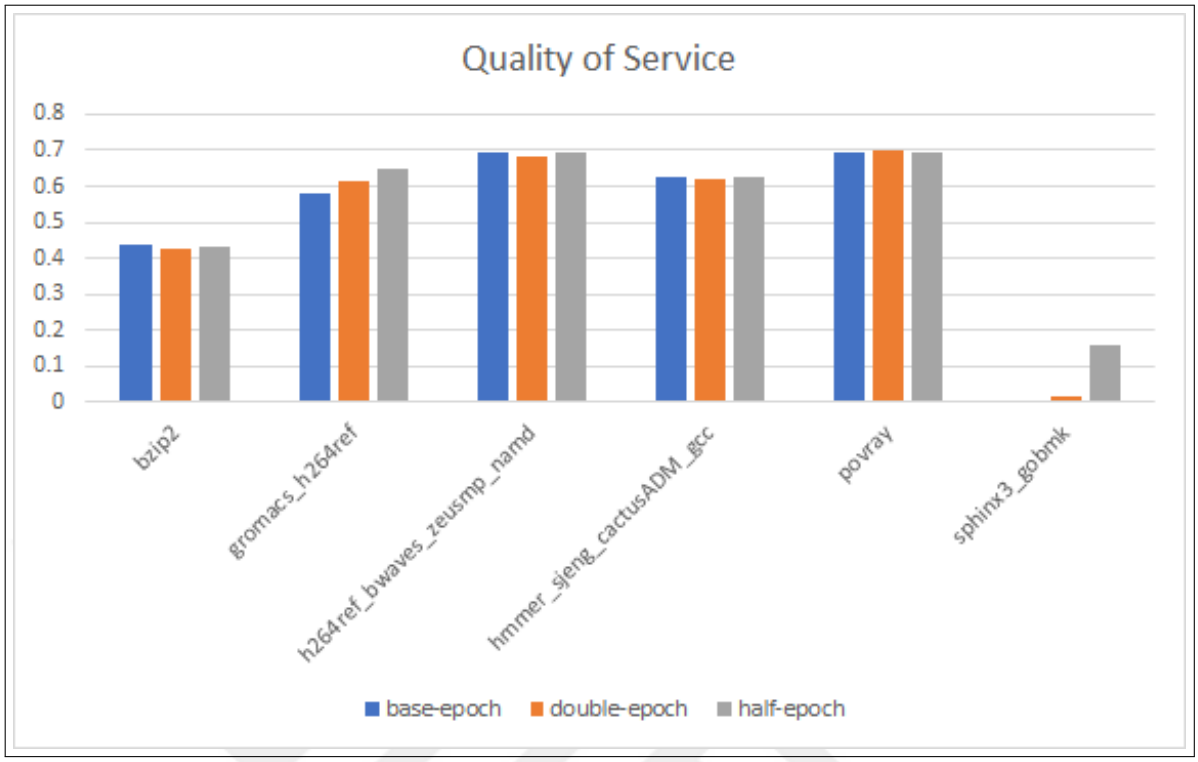

Figure 4.22. Epoch multiplier offline QoS comparison based on single-thread throughput (UB)

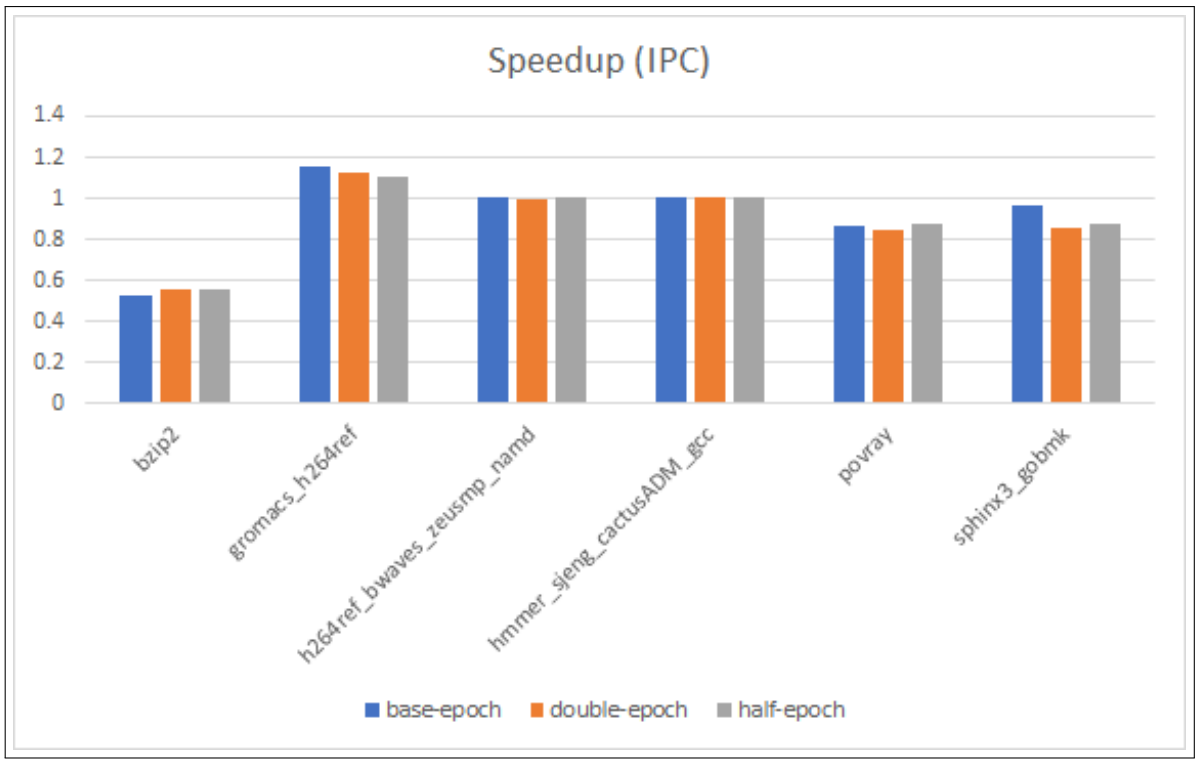

Figure 4.23. Epoch multiplier offline speedup comparison based on baseline-SMT performance (UB)

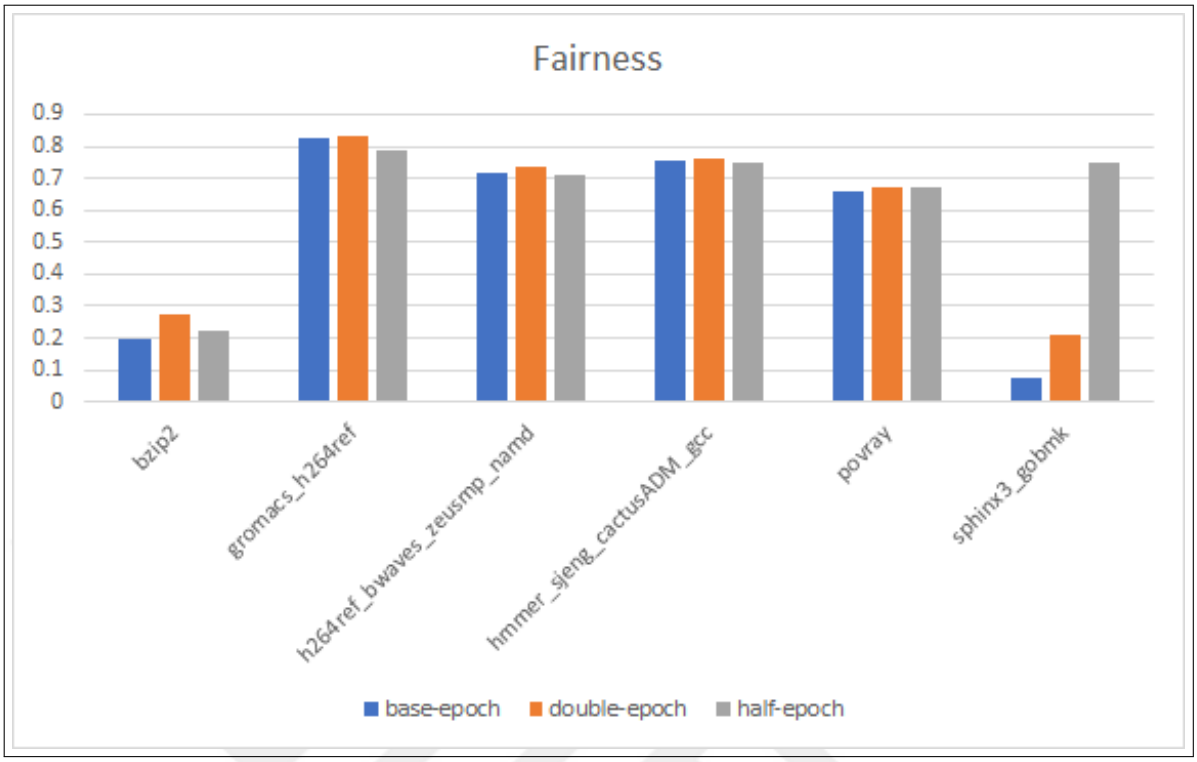

Figure 4.24. Epoch multiplier offline fairness comparison based on baseline-SMT performance (UB)

According to the results observed from Figures 4.22, 4.23, and 4.24; the smaller decision cycles benefit Utility-Based scheduler as well as the Round-Robin scheduler with not only the highest average QoS percentage of 54.16% and a close speedup, but also a quite significant fairness as well. These results lead us to use half-epoch decision cycle for the UB scheduler as the RR scheduler.

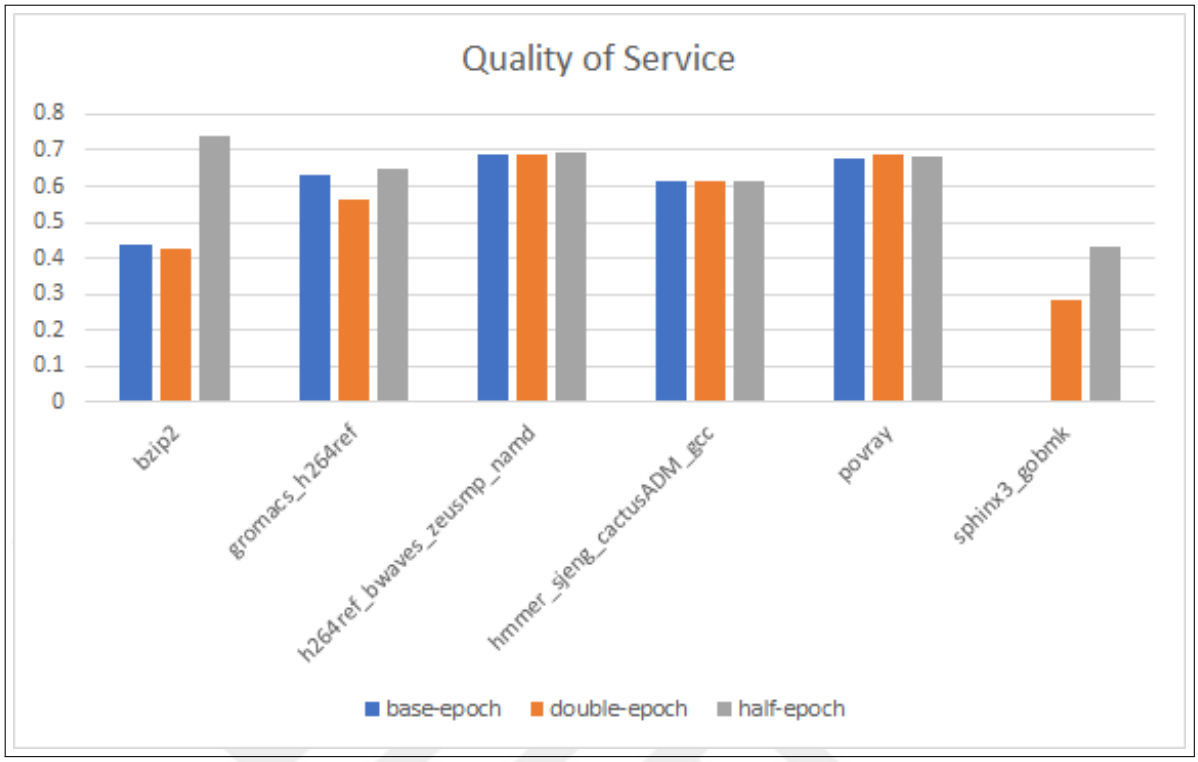

Figure 4.25. Epoch multiplier offline QoS comparison based on single-thread throughput (RUB)

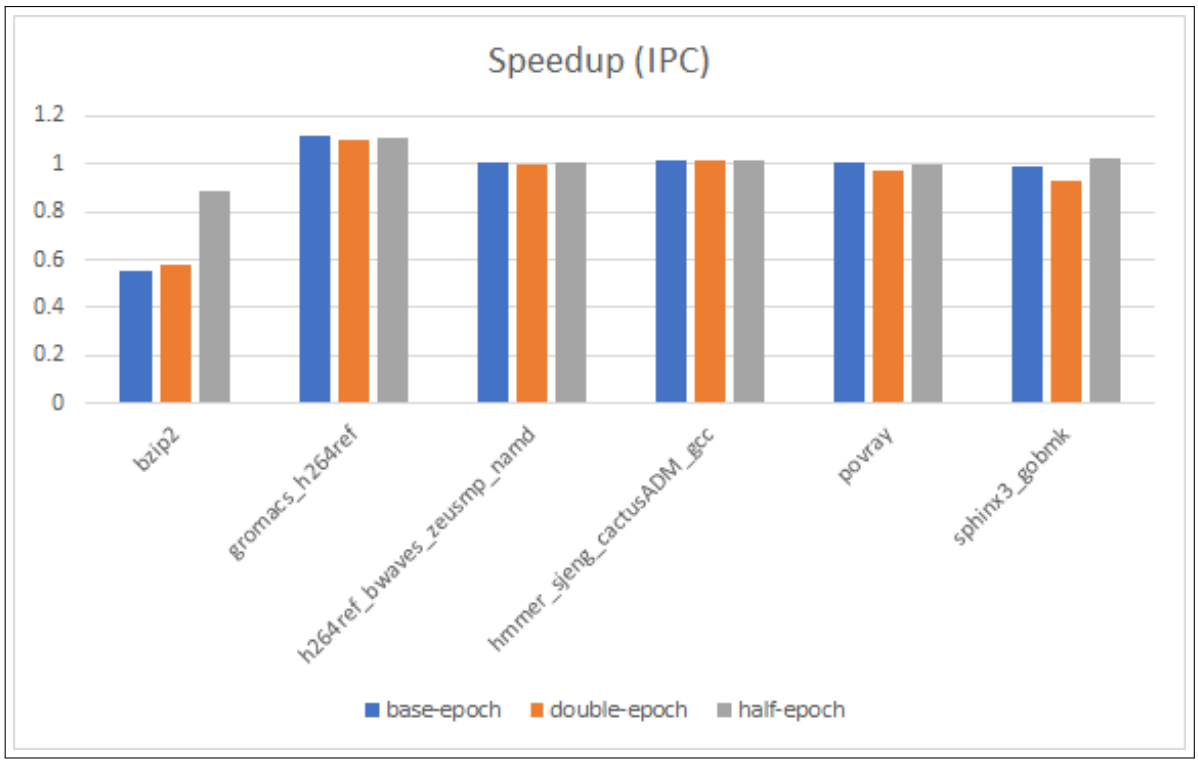

Figure 4.26. Epoch multiplier offline speedup comparison based on baseline-SMT performance (RUB)

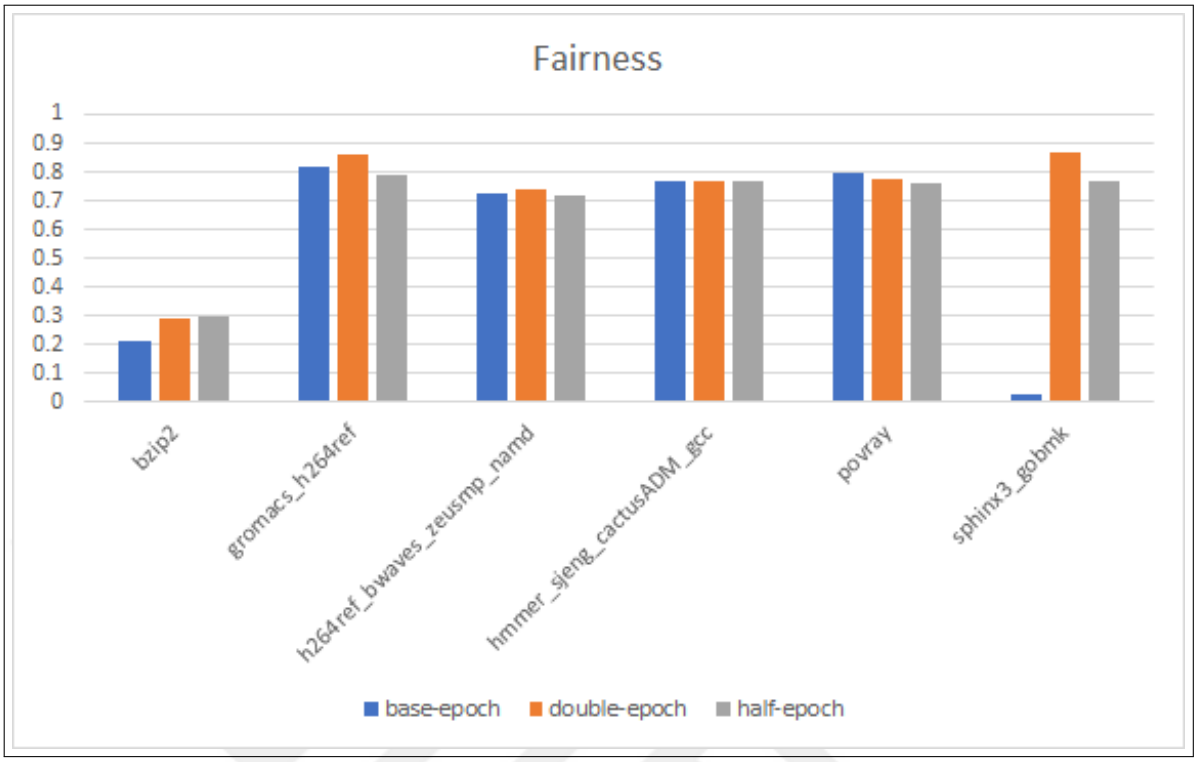

Figure 4.27. Epoch multiplier offline fairness comparison based on baseline-SMT performance (RUB)

In the light of the results observed from 4.25, 4.26, and 4.27; the highest average QoS percentage of 63.58% and highest throughput with the only positive speedup of 0.81% are obtained with the half-epoch decision cycle time. This interval also provides a decent fairness percentage as well.

As a result of the evaluations, we decided that half-epoch decision cycle is the best fit for all three scheduling schemes with different priorities. This shows that granularity of the decision cycles increase the prediction accuracy. However, this tendency of higher granularity wouldn't be forever considering that resource allocations require the allocated resources to turn non-busy in order to be actually provided to another thread, so that they would require time to actually execute, and really low intervals would cause overfitting on data, which would make the processor make decisions regarding only a tiny part of the larger picture, acting in a premature and badly optimized manner.

4.3. SCHEDULER EVALUATION

Using all the methodologies explained above, we finally evaluated three proposed scheduler schemes between each other. Our "the most optimal" setups are summed up below for recap:

- RR scheduler (dynamic-delta, base-delta, half-epoch)
- UB scheduler (dynamic-delta, half-delta, half-epoch)
- RUB scheduler (dynamic-delta, half-delta, half-epoch)

Between these evaluations, we do not consider the hardware overhead, which will be discussed later in this thesis.

4.3.1. SINGLE Workload Test Set

SINGLE workloads were the workloads where all four threads were made to execute the same benchmark executables. This test group would be the group where the expected resource contention would be the most and possibly provide the worst case results regarding our scheduler implementations.

This test group makes appropriate comparisons between the RR, UB, RUB scheduler-based QoSMT implementations and a baseline-SMT implementation as a reference.

Figure 4.28. Scheduler offline QoS comparison based on single-thread throughput (SINGLE)

The offline QoS (Actual QoS) metric is the primary focus of our study. Even though we have our predictors within the implementations for the design, those predictions may be inaccurate and in order to quantitatively analyze our proposed design's efficacy, we should always consider the actual QoS metric.

As it can be easily observed from 4.28, even though our QoS target is set to 90%, the actual QoS values calculated aren't reaching to that point. This shows that our prediction model isn't as accurate as we would like it to be, which will be further examined later. However, in terms of the results we obtained, the average QoS for the RR scheduler is 50.62%, the UB scheduler is 55.99%, and the RUB scheduler is 55.32%. The baseline-SMT performs considerably worse from all three proposed schedulers in terms of QoS with only 30.55%.

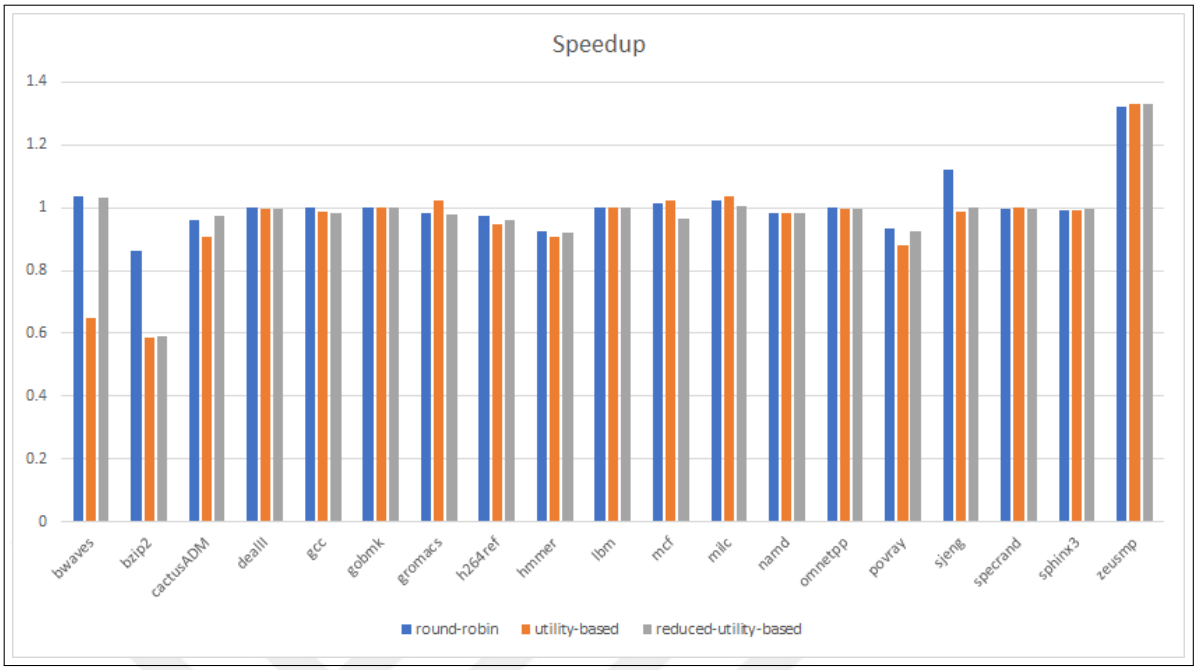

Figure 4.29. Scheduler offline speedup comparison based on baseline-SMT performance (SINGLE)

Even though the proposed designs embrace sacrificing some portion of the performance in order to obtain quality of service to the HPT, we observe that the proposed designs even manage to obtain positive speedup compared to a baseline-SMT which supposedly provide the peak performance from Figure 4.29. On average, we observe that the average speedup of RR scheduler is 1.01, UB scheduler is 0.96, RUB scheduler is 0.98. Considering that our proposed design is expected to sacrifice performance, losing only up to 4% of performance at worst in order to gain varying quality of service increases up to 83.27% for HPT already a quite feasible starting point for the evaluation.

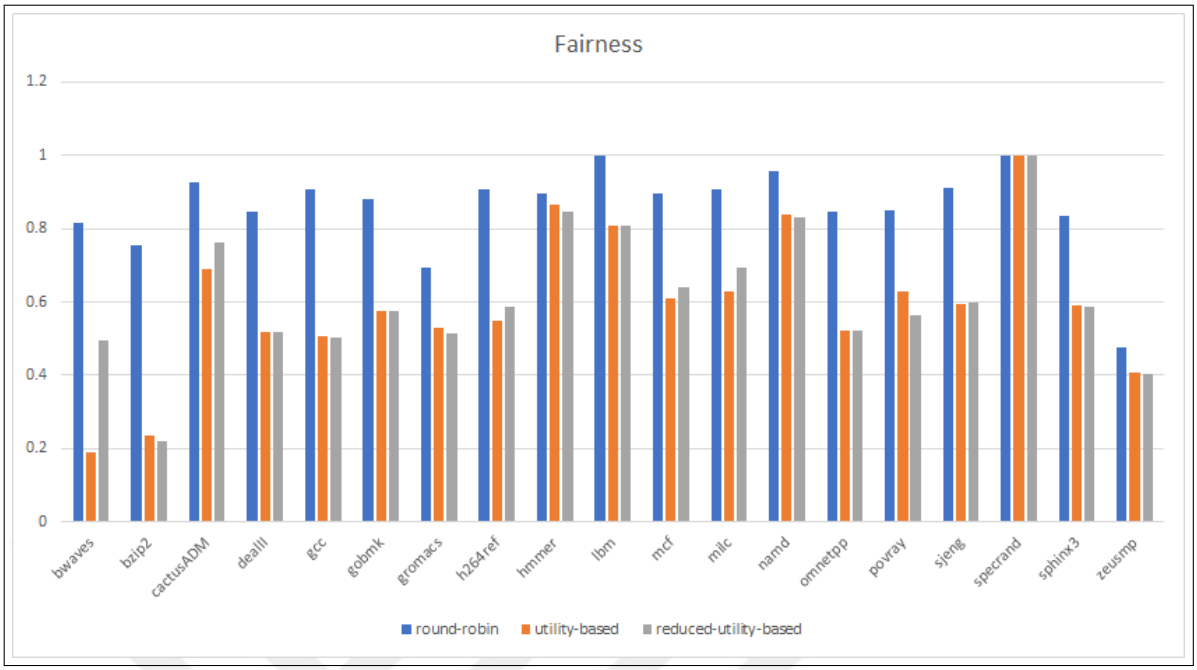

Figure 4.30. Scheduler offline fairness comparison based on baseline-SMT performance (SINGLE)

Fairness is where our proposed designs lack the most, fitting the nature of their implementations as it can be observed from Figure 4.30. The RR scheduler provides a relatively decent fairness of 85.83% between threads. On the other hand, UB and RUB schedulers provide only 59.39% and 61.43% average fairness values, respectively.

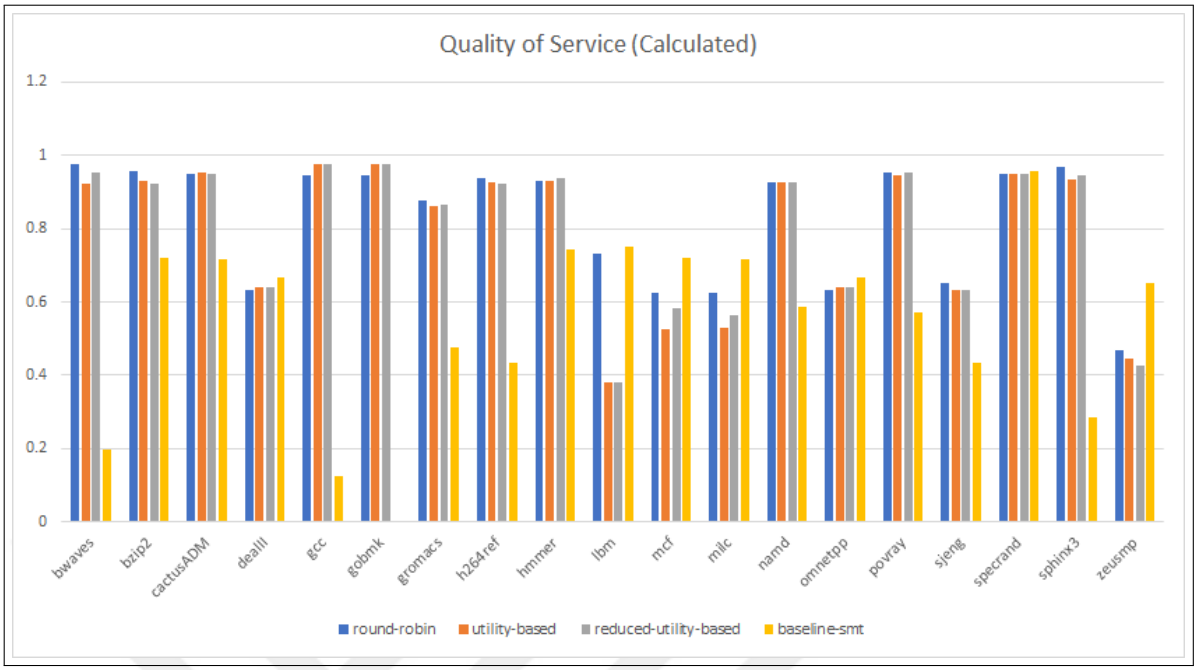

Figure 4.31. Scheduler runtime QoS comparison based on predictions (SINGLE)

Predicted QoS (calculated QoS) is the metric which the design keeps track of in order to achieve quality of service on the HPT. It has already been observed that actual QoS doesn't match our targets. On the other hand, as it can be observed from Figure 4.31, our proposed designs mostly achieve to satisfy their quality of service targets set internally. This shows that our designs satisfy the internal requirements, but inaccurate regarding the actual output. The average mean absolute errors of the RR, UB, and RUB schedulers are 0.36, 0.31, and 0.30; respectively.

4.3.2. DOUBLE Workload Test Set

These workloads are the ones that imitate the original QoSMT study the most with the original study using one workload on the HPT and another workload on the LPT. In this test set, since we have four, not two threads, we assign a workload to the HPT and a separate workload is assigned to all three LPTs to provide a pseudo-duplex-workload.

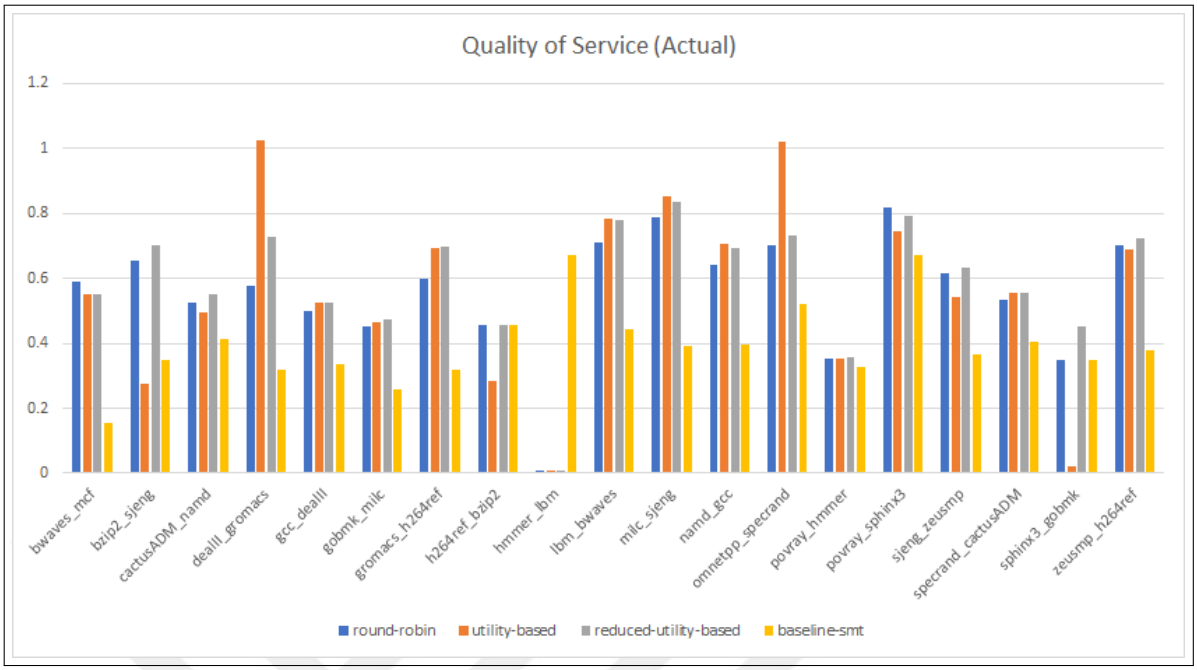

Figure 4.32. Scheduler offline QoS comparison based on single-thread throughput (DOUBLE)

According to the results from Figure 4.32, there is an overall improvement in terms of quality of service from the SINGLE test set. This, is due to the less contentious nature of the workload distribution between the HPT and LPTs. On average, RR, UB, and RUB schedulers provide 55.65%, 55.80%, and 59.17% actual QoS values, respectively. They continue to significantly outperform baseline-SMT which only provides 39.60% actual QoS by marigins up to 49.42%.

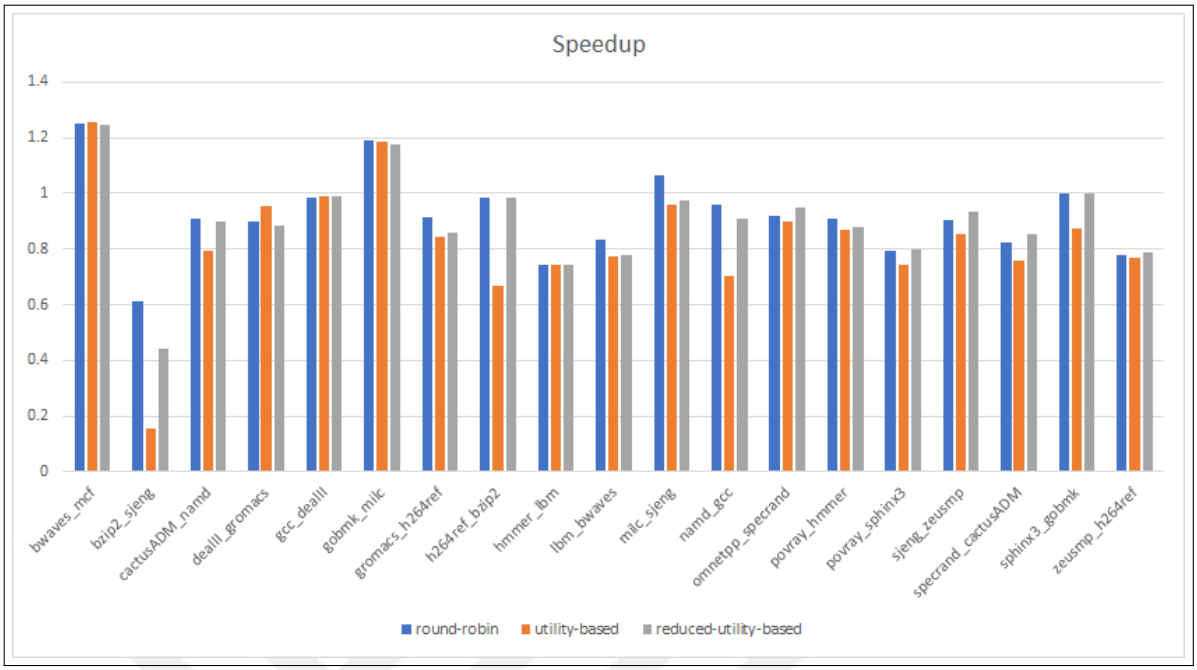

Figure 4.33. Scheduler offline speedup comparison based on baseline-SMT performance (DOUBLE)

With the contentions being reduced between the HPT and LPT workloads, speedup obtained from dynamic resource partitioning got slightly worse. Based on Figure 4.33, in this test set, none of the schedulers outperform the baseline-SMT. The average speedups from RR, UB, and RUB schedulers are 0.92, 0.83, and 0.90, respectively.

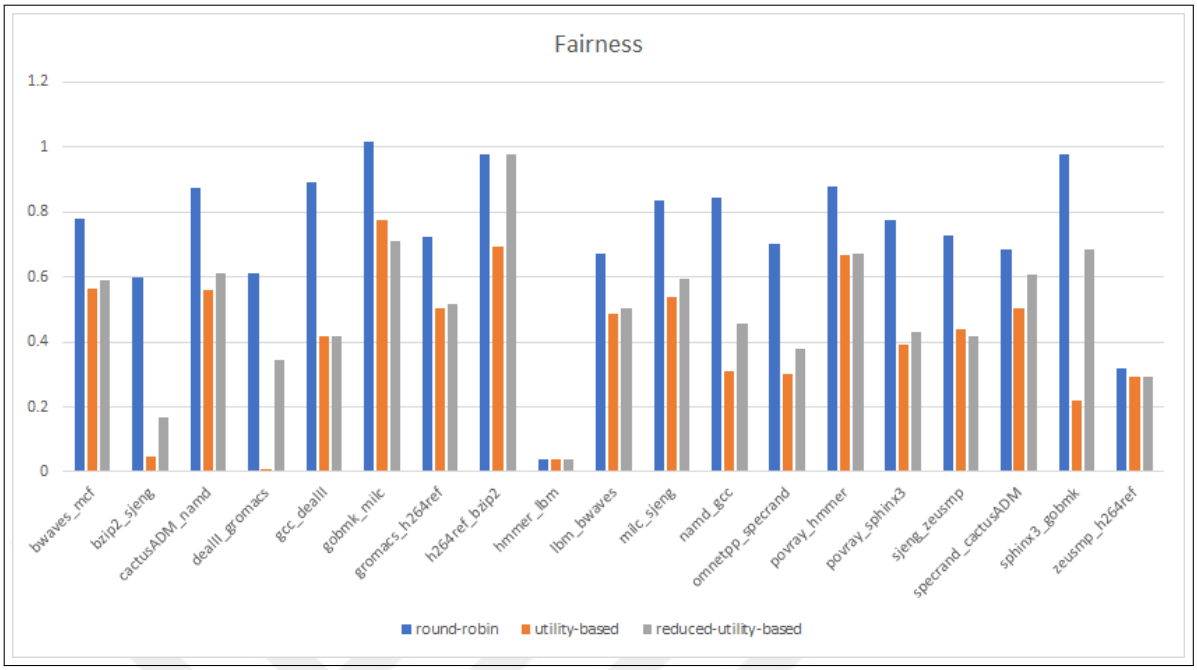

Figure 4.34. Scheduler offline fairness comparison based on baseline-SMT performance (DOUBLE)

With the workload variation increase, fairness decrease as well. However, as it can be observed from Figure 4.34, RR scheduler still provides a rather decent fairness value of 73.29%. On the other hand, UB and RUB schedulers pertain fairness values of mere 40.84% and 49.51%, respectively.

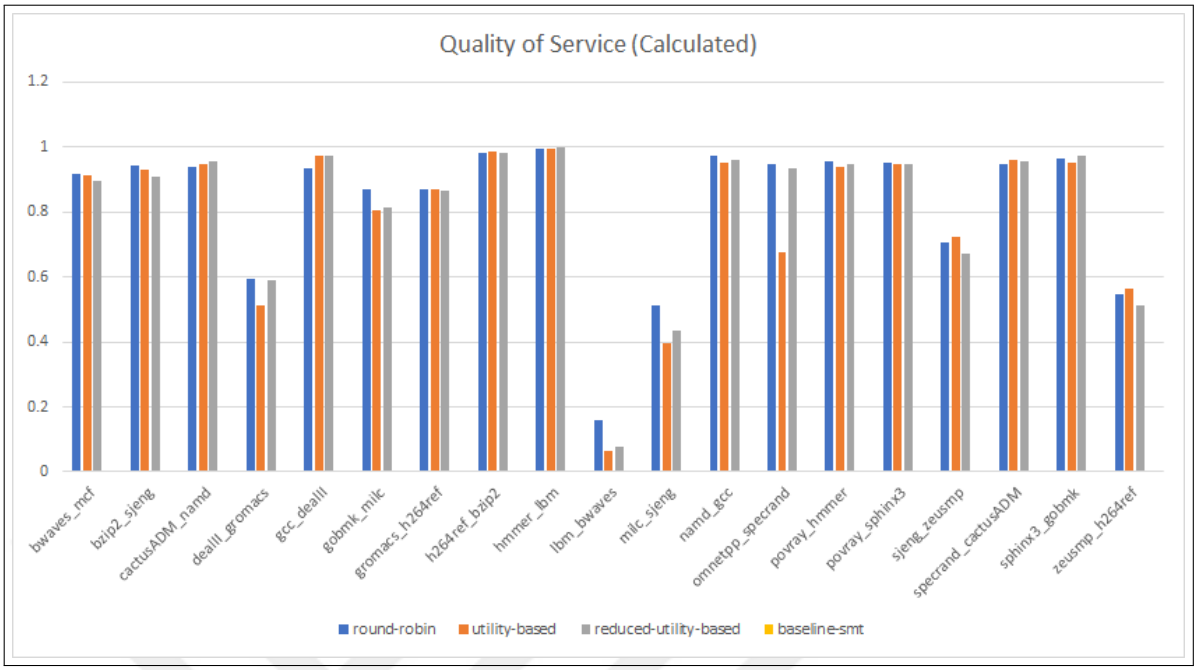

Figure 4.35. Scheduler runtime QoS comparison based on predictions (DOUBLE)

The proposed designs continue to mostly satisfy the internal predictive requirements as well as it can be observed from Figure 4.35. But, the predictions are worse in this test set with the actual quality of service values being better. Therefore, the accuracy is actually worse here with mean absolute errors of RR, UB, and RUB schedulers being 0.37, 0.47, and 0.37, respectively.

4.3.3. MULTI Workload Test Set

This test set assigns a different benchmark executable as a workload to each thread on the core. This scenario is the closest one to the real-life conditions and also the best for getting the most out of our proposed designs.

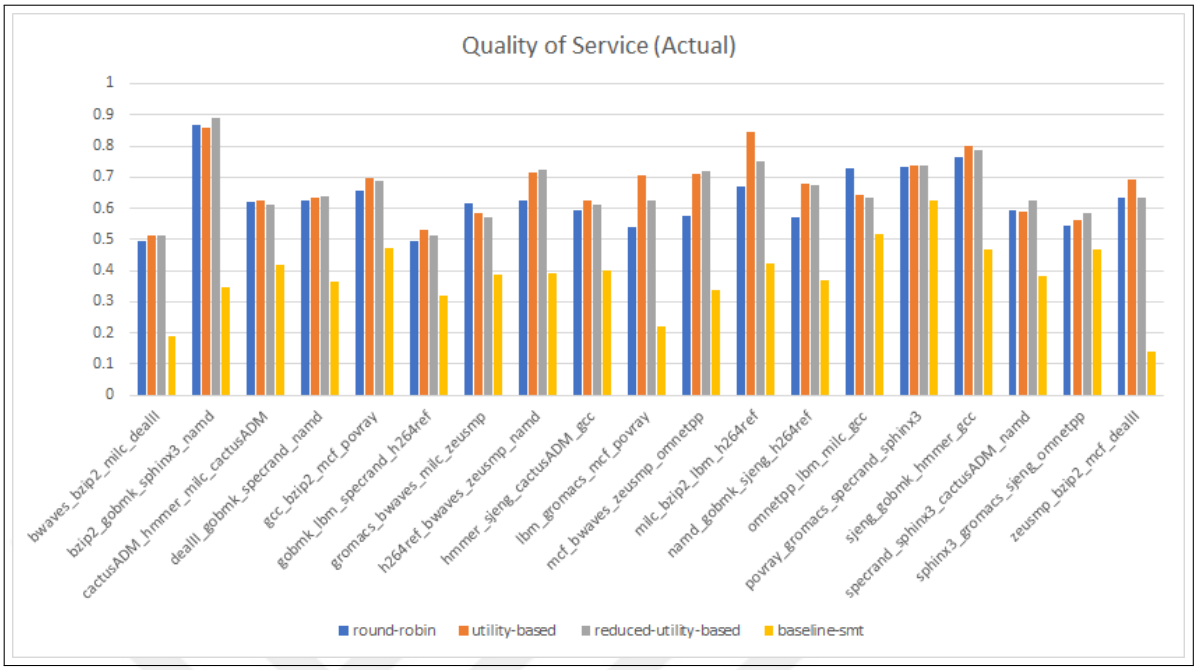

Figure 4.36. Scheduler offline QoS comparison based on single-thread throughput (MULTI)

Based on the results displayed in Figure 4.35, the actual QoS values are the highest of the three test set's results with RR, UB, and RUB schedulers showcasing 62.86%, 67.13%, and 65.94%, respectively. Compared with the 38.13% QoS of the baseline-SMT, the proposed implementations provide up to 76.06% relative QoS improvement.

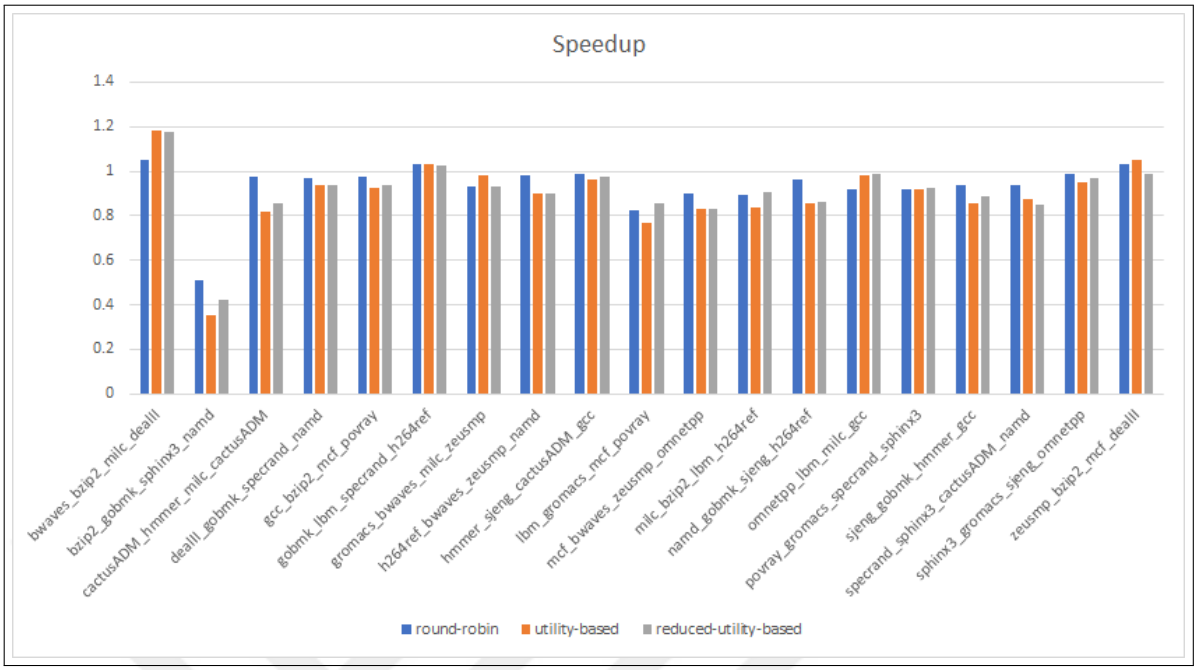

Figure 4.37. Scheduler offline speedup comparison based on baseline-SMT performance (MULTI)

The speedup values for this test set are in the middle ground of the SINGLE and DOUBLE test sets. According to Figure 4.37, the speedup values of RR, UB, and RUB schedulers are 0.93, 0.90, and 0.91, respectively.

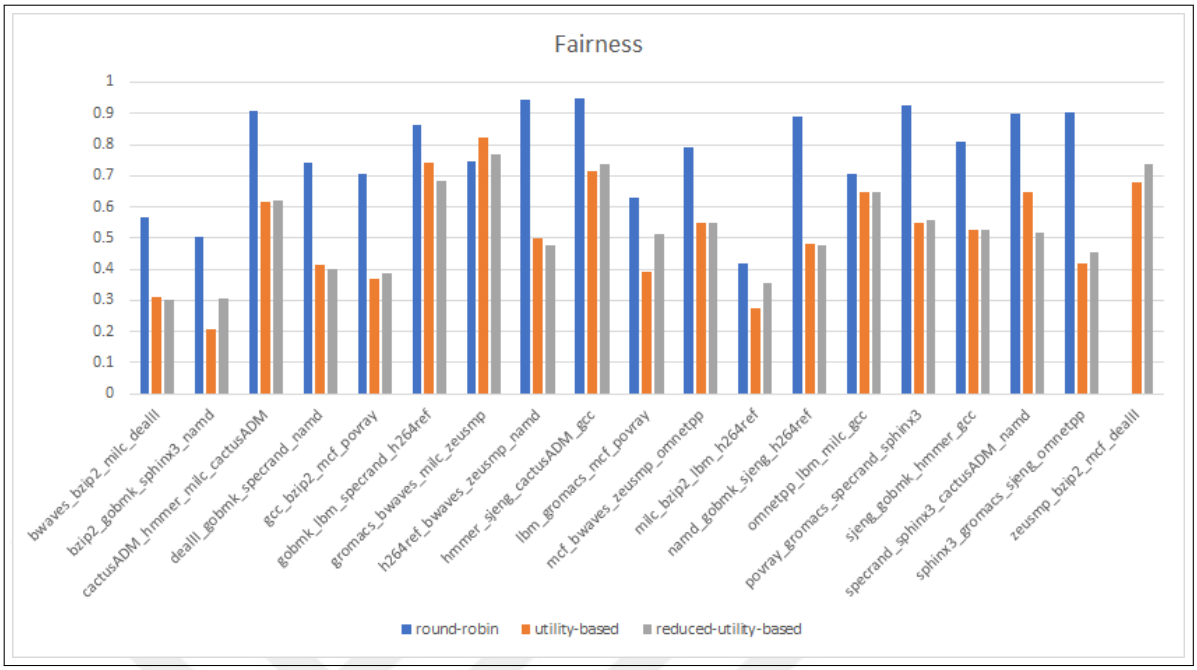

Figure 4.38. Scheduler offline fairness comparison based on baseline-SMT performance (MULTI)

Fairness between the threads are better on this test set as well, thanks to less starvation-prone workload combinations compared to the DOUBLE test set. According to Figure 4.38, RR, UB, and RUB schedulers maintain fairness values of 77.17%, 51.90%, and 52.73%, respectively.

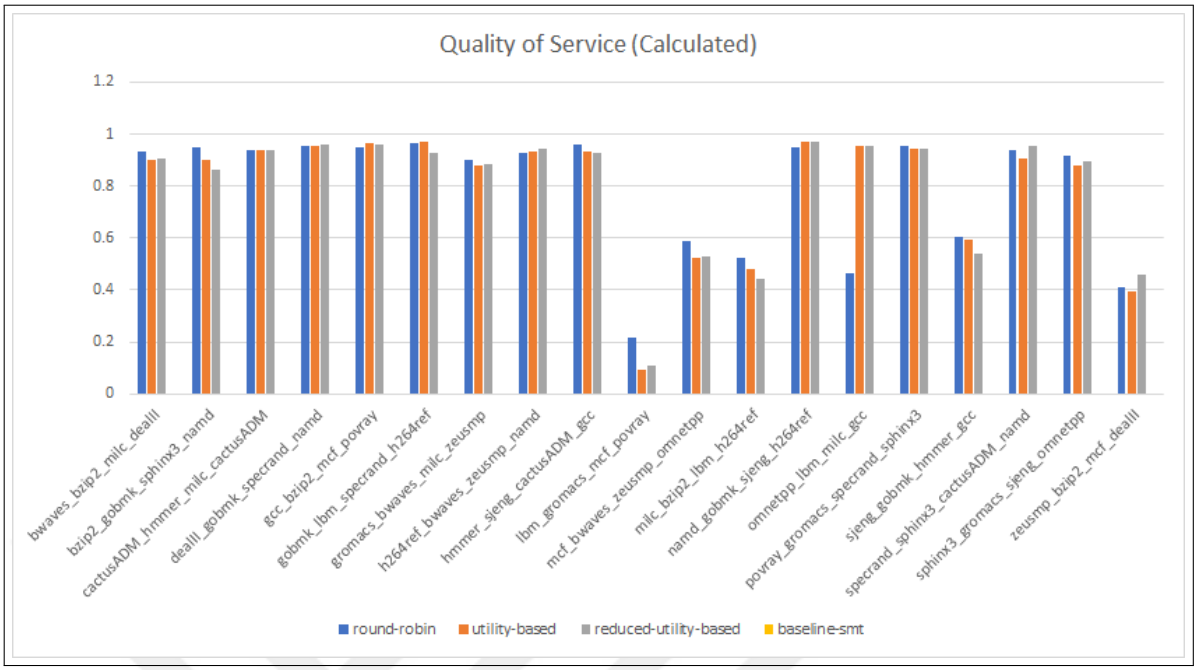

Figure 4.39. Scheduler runtime QoS comparison based on predictions (MULTI)

With the workloads varying, contention delay calculations through different resources increase as well. Based on the results obtained from Figure 4.39, RR, UB, and RUB schedulers showcase average mean absolute errors of 0.28, 0.30, and 0,29, respectively; which are the lowest average mean absolute errors in these test sets.

The summary of all the results above are provided in the Table 4.2 below:

Table 4.2. Summary of Scheduler Analysis

	QoSMT-RR	**QoSMT-UB**	**QoSMT-RUB**	**SMT-BASE**
Average QoS	0.5638	0.5964	0.6014	0.3609
Average Speedup	0.9534	0.8953	0.9291	1.0000
Average Fairness	0.7876	0.5071	0.5456	1.0000
Average QoS MAE	0.3372	0.3594	0.3206	0.2615

When we examine the overall efficacy between all the tested scenarios, it is hard to determine an absolute "best" approach between the three proposed scheduler schemes. We can only claim that Utility-Based scheduler is weaker than the Reduced Utility-Based scheduler since

it was eclipsed by the latter in terms of all the results with a higher hardware overhead. This is most likely due to the reduced utility-based scheduler utilizing resource fetches from the lower priority threads more accurately since its focus was not to protect highly-utilized LPTs, causing a delayed resource starvation on the LPTs, therefore, providing higher QoS.

Between the RR and RUB schedulers, it is much harder to make a choice. But it can be said that if one prioritizes overall throughput increase of 2.85% and overall fairness increase of 68.33% more than the 6.25% quality of service increase, RR scheduler can be utilized. RUB would be the better fitting model for achieving our primary goal of obtaining higher QoS on HPT, though.

4.4. HARDWARE OVERHEAD

Since RR and RUB were the best two choices as scheduler implementations, and RUB requires marginally higher amount of resources due to its slightly more complex nature, we calculated the transistor requirements of the proposed control circuitry. As aforementioned before, we implemented the logic related to the proposed design only on SystemVerilog, and synthesized it with YoSys 0.9.0 using UC Berkeley's ABC library for CMOS technology in order to get transistor count estimations. It should be noted that different synthesis tools and technologies would cause differing results.

According to the synthesis analyses, RUB scheduler implementation with tag-array supported utility-based cache partitioning, dynamic delta and normalized delay register cost the gate counts provided in the Table 4.3 below:

Table 4.3. Synthesis Gate and Transistor Analysis

ABC Results	**Cell #**	**CMOS Transistor #**	**Overhead per Core (SMT-8)**	**Overhead # for IBM Power 10**	**Overhead % for IBM Power 10**
NOT cells	1,268	2,536	20,288	608,640	0.003382%
NAND cells	2,924	2,924	23,392	701,760	0.003898%
NOR cells	3,672	14,668	117,504	3,525,120	0.019586%
Total logic cells	7,864	20,148	161,184	4,835,520	0.026864%

According to these results, a single core implementation for an 8-threaded SMT core would cost 161,184 transistors per core. An IBM Power10 E1080 has 30 SMT-8 cores [37]. Considering that we shall implement this circuitry to all the cores, the total cost becomes 4,835,520 transistors, which is only a 0.0269% of hardware overhead. Based on these results, we can easily claim that our proposed approaches are significantly lightweight in terms of hardware overhead, and these proposed designs improve TCO and performance per Watt significantly by its efficiency for the data center applications.

4.5. ADDITIONAL ANALYSES

There are more analyses which wouldn't impact the design decisions but provide further overview on these proposed designs, though. Those can be listed as:

- Hot-starting the systems on HPT-focused resource amounts
- Running the systems with different internal QoS targets
- Examining resource cap occurrences

4.5.1. Hot-Started HPT

In our test scenarios, we run the simulations for 1,000,000 max cycles with a decision cycle count of 10,000 for all the schedulers. Also, we start each four threads on the core with an equal amount of resources while we aim for a 90% predictive QoS target. This means that our proposed designs had only 100 decision cycles at most to actually bring the HPT into a resource amount that satisfies the 90% QoS target. Therefore, the first few decision cycles of the tests end up with low QoS ratios that can negatively affect our results, which would impact the results even worse on the shorter tests.

In order to clear out such negative effects, we also ran the tests with a hot-start configuration. This configuration applies totally the same implementation as our previously discussed designs but starts the HPT with the highest possible amount of resources at the initial simulation time. So, we know that our obtained QoS averages wouldn't be negatively

affected through the initial decision cycles.

Figure 4.40. Hot-start offline QoS comparison based on single-thread throughput (RR)

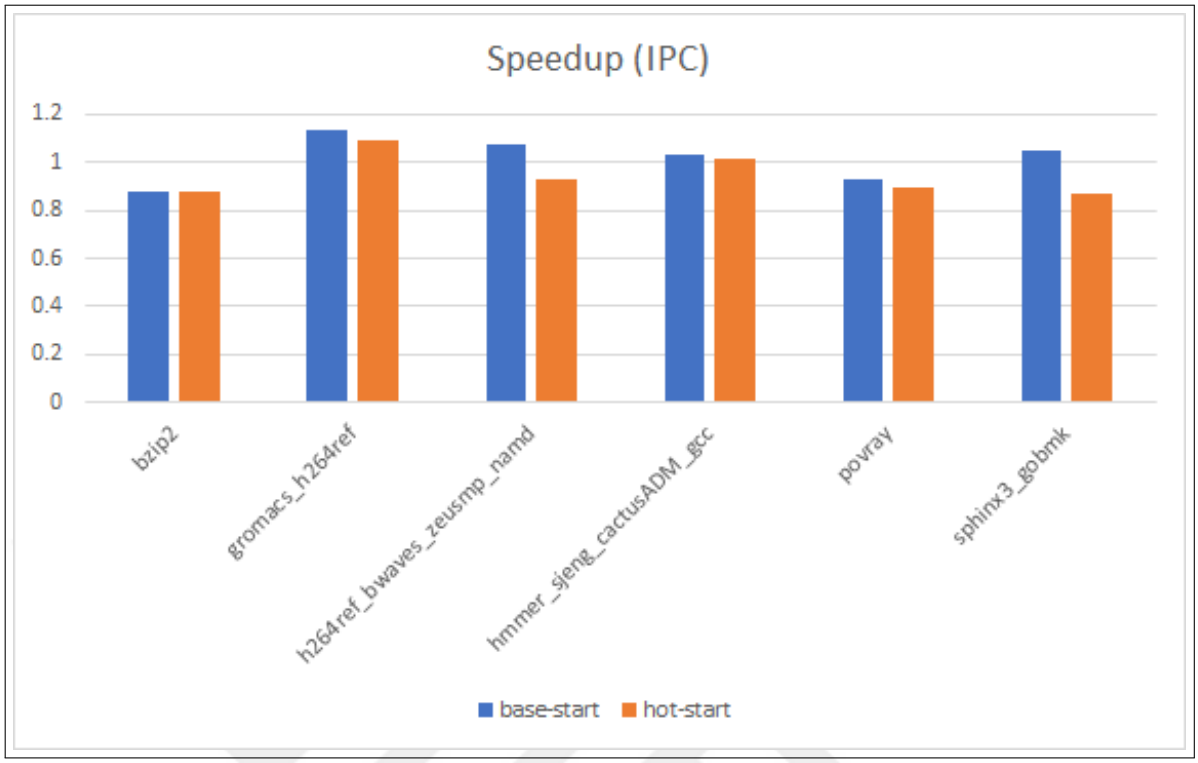

Figure 4.41. Hot-start offline speedup comparison based on baseline-SMT performance (RR)

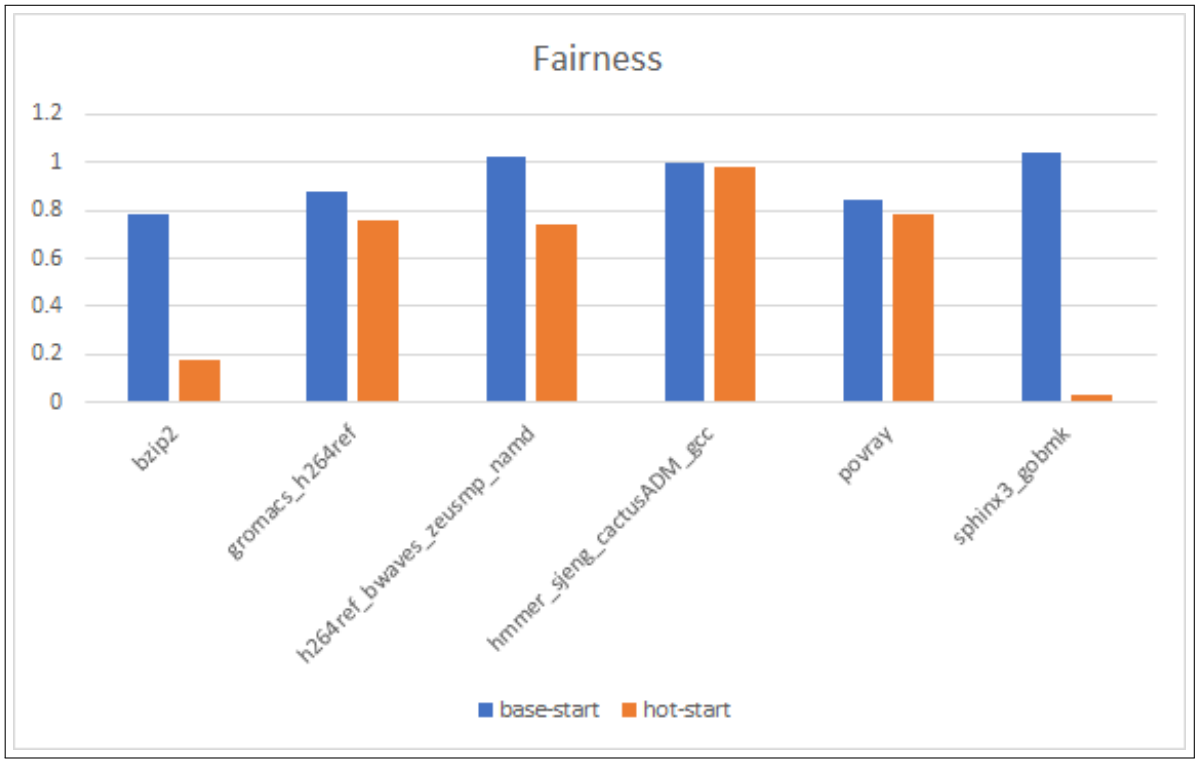

Figure 4.42. Hot-start offline fairness comparison based on baseline-SMT performance (RR)

According to the results obtained from Figures 4.40, 4.41, and 4.42; it can be said that QoS changes positively with the hot-start, improving the original average of the RR scheduler of 54.29% to 61.85%, providing a 13.93% relative increase. The increased QoS also sharpens the speedup and fairness losses, though, with the average speedup reduced from 1.02 to 0.95 and average fairness reduced from 92.99% to 57.90%. The relative losses for the improved QoS are somewhat high for the round-robin scheduler, making hot-start approach for RR a not-an-obvious choice.

Figure 4.43. Hot-start offline QoS comparison based on single-thread throughput (UB)

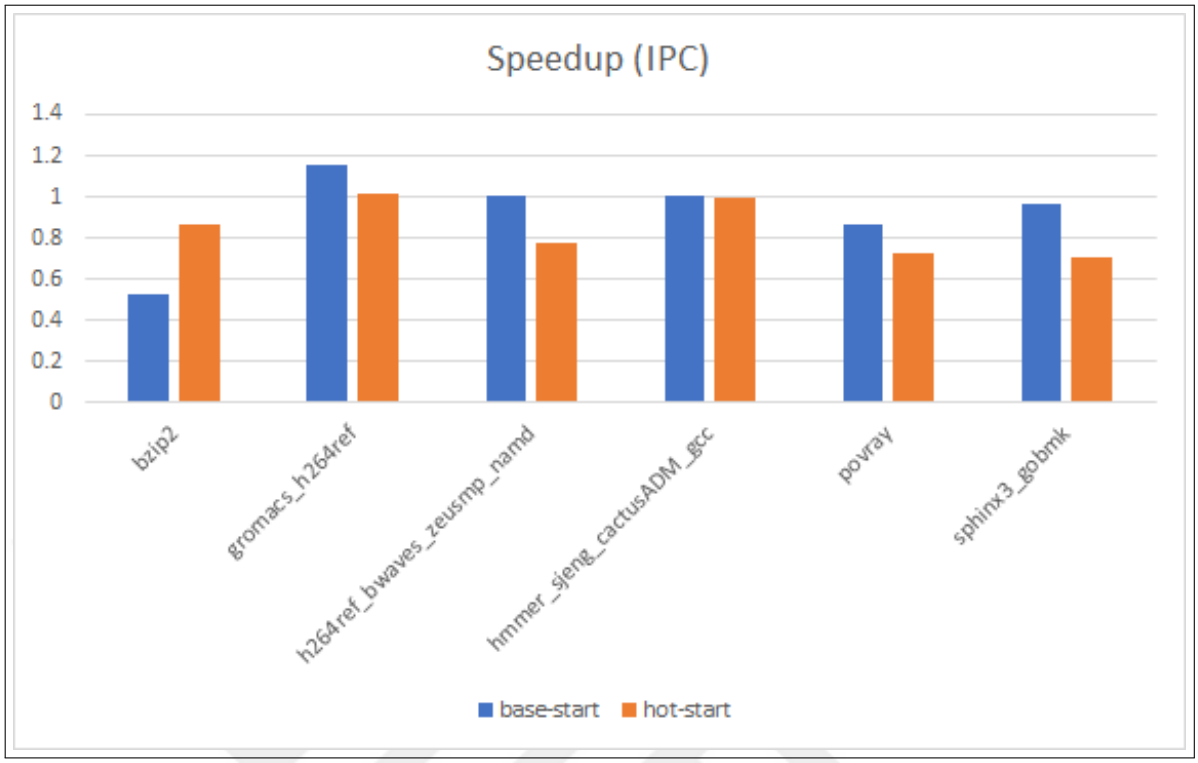

Figure 4.44. Hot-start offline speedup comparison based on baseline-SMT performance (UB)

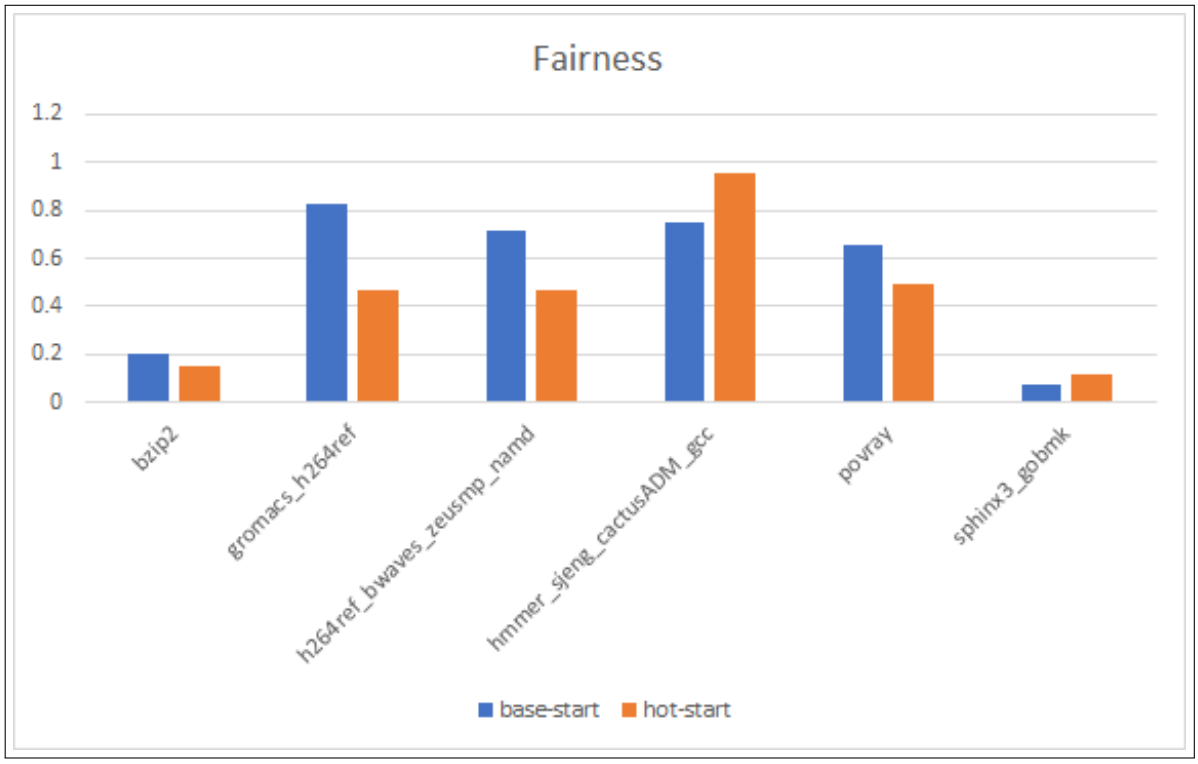

Figure 4.45. Hot-start offline fairness comparison based on baseline-SMT performance (UB)

When we examine Figures 4.40, 4.41, and 4.42; we observe a much better looking set of results compared to the round-robin approach. The average QoS increases from 50.66% to 65.04% which results in a 28.39% relative QoS increase meanwhile only losing 7.74% in terms of speedup and 17.50% in terms of fairness.

Figure 4.46. Hot-start offline QoS comparison based on single-thread throughput (RUB)

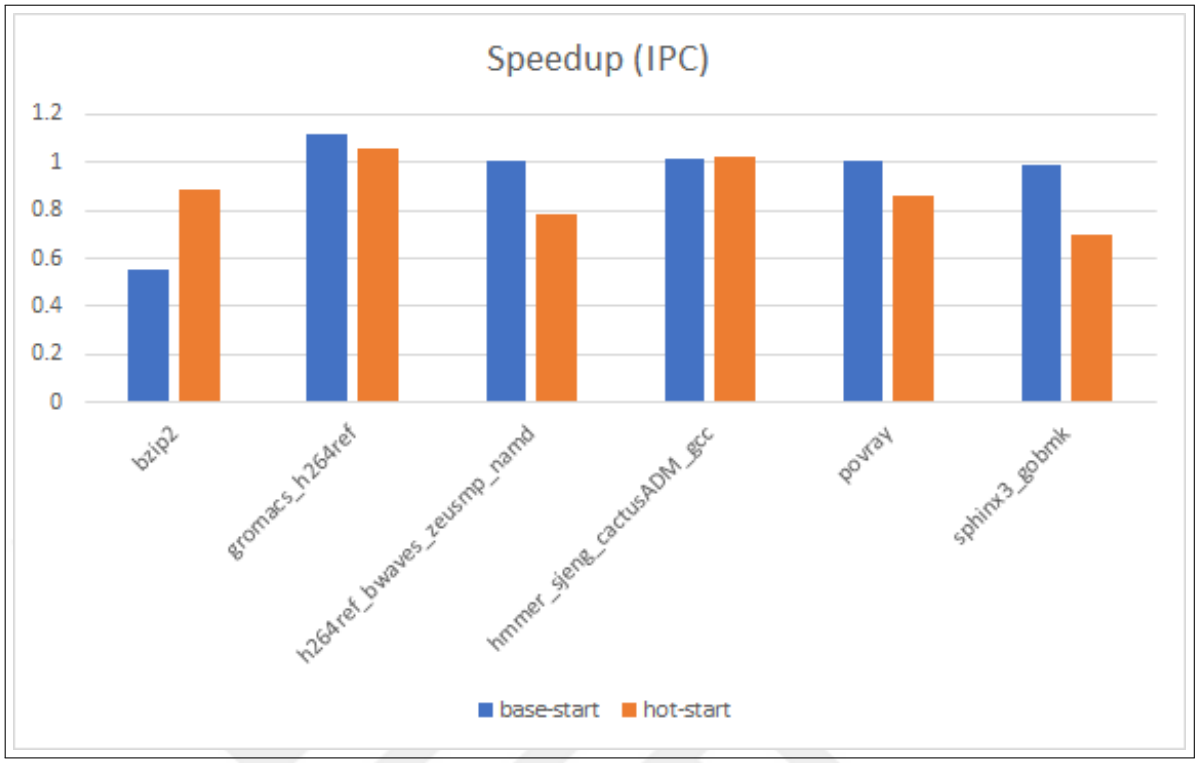

Figure 4.47. Hot-start offline speedup comparison based on baseline-SMT performance (RUB)

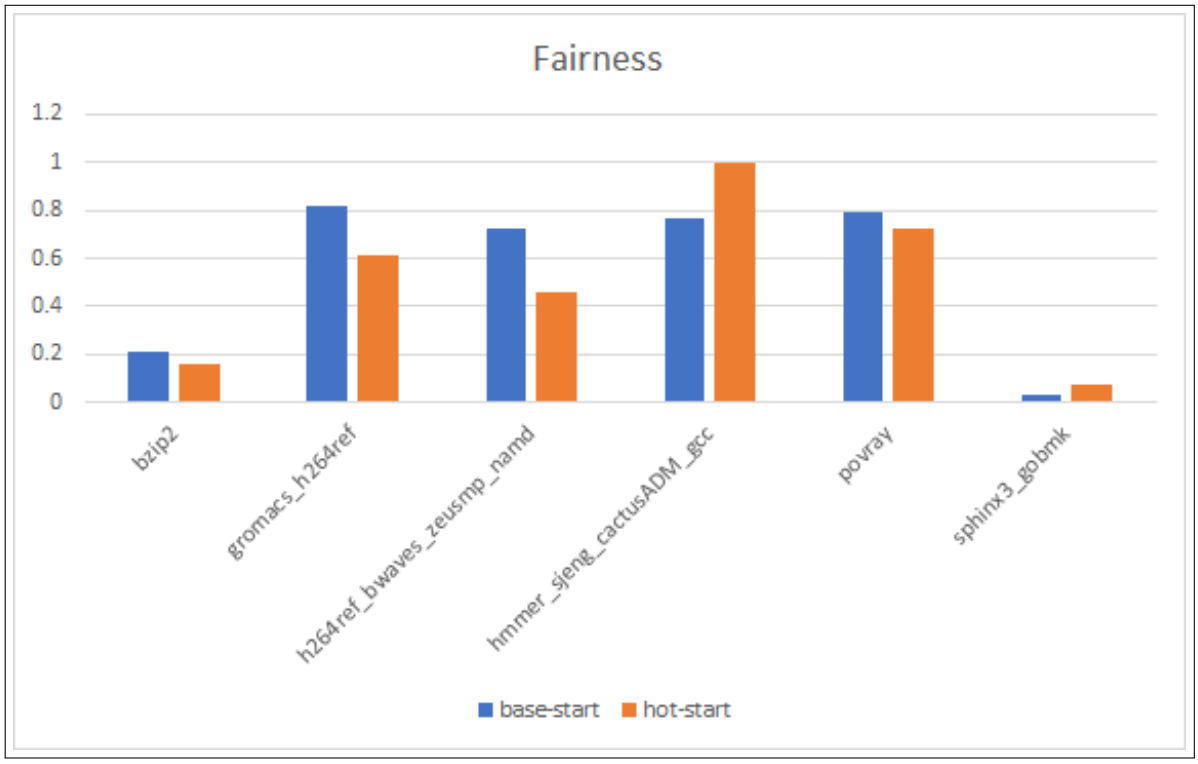

Figure 4.48. Hot-start offline fairness comparison based on baseline-SMT performance (RUB)

Based on Figures 4.40, 4.41, and 4.42; we can easily say that RUB scheduler reacts to the hot-start configuration in an almost identical manner as the UB scheduler. The average QoS improvement is 26.13%, resulting in 64.20% average QoS percentage with only 6.65% speedup and 9.62% fairness loss.

In light of these results, we can claim that utility-based schedulers have more potential to unlock behind the hot-start configuration. The hmmer-sjeng-cactusADM-gcc test seems to oscillate resources between HPT and LPTs in those schedulers, resulting in overall QoS decrease unlike the other test scenarios.

The hot-start approach is not an implementation difference; therefore, treated in a separate chapter than the previously evaluated setups but we can argue that it is important to achieve the highest efficacy from the RUB scheduler, which was one of the best proposed approaches. However, in the long run, processors run through several trillion cycles through a day, and a few thousands of cycles on the startup wouldn't actually change how a processor behaves in a real-life approach.

4.5.2. Different QoS Targets

A parametric internal QoS target wouldn't affect the proposed designs' efficacy, but it is an important aspect to consider especially if it could be a configurable parameter. With appropriate operating system support, through only a single internal register update made on core through a privileged instruction, the proposed designs could be switched not only on and off, but also could be utilized in different focus intensities. Therefore, a core utilizing the proposed circuitries can utilize an immense versatility in terms of quality of service and performance curve. We have tested our design with the internal QoS targets given below:

- 80 percent QoS target
- 70 percent QoS target
- 60 percent QoS target

4.5.2.1. 80 Percent QoS Target

This subset reduces the QoS requirements by 10% in order to double the HPT slack window and observe the changes based on varying QoS targets. This is the first of our three subsets to obtain a complete picture regarding the QoS and implementation efficacy pattern.

Table 4.4. Summary of 80% QoS Target Tests

	QoSMT-RR	**QoSMT-UB**	**QoSMT-RUB**
Average QoS	0.4512	0.4976	0.5504
Average Speedup	1.0241	0.9209	0.9650
Average Fairness	0.8049	0.5679	0.6888
Average QoS MAE	0.4077	0.3261	0.2394

When the results in Tables 4.2 and 4.4 are compared, a 14.90% average decrease in QoS is observed between 90% and 80% targets. This change can be said to be close to the relative 12.11% QoS target change. However, speedup and fairness increase accordingly as well. One of the salient aspects from these results is the observation of QoS prediction accuracy increase coming with lower requirements where the prediction mean average error reduces by approximately 5%.

4.5.2.2. 70 Percent QoS Target

The 70 percent QoS target in this study is one of the most similar setups to the 90 percent QoS target used in the original QoSMT study, from an LPT point of view. In the original study, there were only two threads, where one thread was sacrificed in order to maintain 90% QoS on the HPT. However, we have three threads here, and even though in SMT cores we cannot linearize thread resources with the QoS (otherwise, SMT would have no benefit; therefore, meaning at all), continuing to provide 90% of QoS to the HPT drains LPTs much more significantly than a 2-threaded setup.

Table 4.5. Summary of 70% QoS Target Tests

	QoSMT-RR	**QoSMT-UB**	**QoSMT-RUB**
Average QoS	0.4258	0.4887	0.5494
Average Speedup	1.0255	0.9167	0.9724
Average Fairness	0.8107	0.5646	0.6966
Average QoS MAE	0.3429	0.2566	0.1552

Based on the summaries provided in Tables 4.2 and 4.5, the 70% QoS target causes a 16.90% actual QoS degradation compared to the 22.22% reduction of the QoS target; which leads us to the result that QoS targets are in line with the actual impact when it is considered with 80% QoS target results. Performance and fairness increase with the decreasing QoS as well. Also, it is worth noting that mean absolute errors of the QoS predictors decrease by 25.81% with the newly assigned 70% QoS target, which may be indicative of easier-to-meet goals providing higher accuracies throughout the design.

4.5.2.3. 60 Percent QoS Target

This subset reduces the QoS requirements by 40% in order to maximize the HPT slack window and observe the changes based on varying QoS targets. Since this is the easiest-to-satisfy target in our system, we expect the highest throughput on this approach.

Table 4.6. Summary of 60% QoS Target Tests

	QoSMT-RR	**QoSMT-UB**	**QoSMT-RUB**
Average QoS	0.3759	0.4847	0.5304
Average Speedup	1.0270	0.9256	0.9927
Average Fairness	0.8309	0.5623	0.7486
Average QoS MAE	0.3114	0.2136	0.1054

When we examine our results provided in Tables 4.2 and 4.6, alongside the other QoS target scenarios, the picture becomes much clearer. With an average actual QoS reduction of

21.04% occurring with a 33.34% prediction target decrease, the pattern emerges that actually higher QoS targets cause less overall effectiveness in terms of QoS. The trend of speedup and fairness increase continues as well. The most remarkable result from this test case is, however, the fact that our mean average errors of QoS predictors decreased by 38.03%, with UB and RUB schedulers benefiting from it the most.

In light of all these results, we can confidently claim that our predictor models' accuracies are in line with the demanded requirements and able to satisfy varying strictness levels reliably. Also, the performance targets that are easier to achieve provide much fewer errors, which indicates that the systems proposed can hugely benefit from improvements in this area.

4.5.3. Resource Cap Occurrences

Since our proposed mechanisms aim to obtain resources from low-priority threads to provide quality of service to the high-priority thread, the low-priority threads are left with low resource amounts most of the time. However, even though LPTs are almost starving for resources, HPT may demand more resources from them with the goal of satisfying its predictive QoS target. In these cycles, if the relative LPTs aren't capable of completely satisfying the HPT's, they may end up providing only a partial amount of what is desired or nothing at all, causing that epoch cycle for decisions to be wasted. In this section, we examine the occurrences of such bottleneck contentions in order to obtain a better insight regarding the proposed systems' optimization.

Table 4.7. Bottleneck Contention Average Through Test Sets

Contention # Avg.	**QoSMT-RR**	**QoSMT-UB**	**QoSMT-RUB**
SINGLE	0.47	58.07	55.33
DOUBLE	0.58	53.91	48.56
MULTI	0.51	44.09	44.56

According to Table 4.7 above, utility-based schedulers provide very high contention averages overall. The primary reason is these schedulers being much more likely to starve singular

threads since they seek utility-based choices instead of fair approaches, unlike the round-robin counterpart.

Another trend observed through these results is that the average bottleneck amount reduces through the workload variation increases. This is caused by the less contentious nature of more varying workloads.

In light of these results, we can say that bottlenecks occur in all kinds of implementations, and the proposed designs could be improved by extending LPT resource-providing mechanisms with substitute LPTs. In other words, the proposed designs can be extended with a mechanism so that if an LPT lacks a resource that is requested of it if any other LPT can provide it, the said mechanism can provide it through the other threads, saving epoch cycles to respond to the QoS demands of the HPT. The increased hardware cost would require a reassessment of the hardware overhead, though.

5. CONCLUSION AND FUTURE WORK

In this thesis, we proposed methods to enable simultaneous multithreading in data center operations without incurring severe quality of service degredations. Our baseline study was "QoSMT: supporting precise performance control for simultaneous multithreading architecture" of Jin et al. in our approaches yet we implemented our solutions to a four-threaded SMT core rather than the reference study which was implemented to a two-threaded SMT core. This difference brought a requirement for scheduling algorithms and we proposed three: Round robin, utility-based and reduced-utility-based. Besides, we implemented some optimizations regarding our baseline architecture.

During our preliminary analyses, we observed some anomalies in some test cases that contradict the best approaches chosen and explained. The first one of them is the unexpected almost-zero quality of service results obtained from the sphinx3_gobmk workloads. Since the lower quality-of-service resulting approaches handled that scenario significantly better than the higher-yielding approaches, that scenario could be treated as anomalous. The reason behind that is most likely the critical resource amounts a workload would require for operating smoothly. We can imagine the mentioned scenario like this: A workload constantly attempts to access six cache lines repeatedly and doesn't need more cache lines than that. They would be redundant. However, if they were left with 5 cache lines because they were performing highly and provided some resources to the other threads, there would be a cache miss for each six cache accesses. Therefore, that workload is highly prone to losing performance for losing that sixth cache line. The highly quality-of-service providing approaches most likely manage to pass that critical line since they seize the resource partitioning opportunity better, but that costs a significant performance loss in that particular case. The loss of fairness between those scenarios also supports this claim. This could be treated as an Achilles heel of the suggested implementations through this study. The second anomaly is that the systems chosen as the best do not result in the highest QoS between all six sample workloads they have been tested through. This also could be explained by the reasons of the first anomaly mentioned, in milder examples.

According to our results, our approaches showcased an overall success by enabling significant quality of service when multiple workloads colocate with a high-priority workload. Our

quality of service centric implementations display performance degradations up to 10.47% compared to a baseline 4-threaded SMT core in order to prioritize the quality of service of high-priority thread. However, as our results have shown, QoSMT-RR and QoSMT-RUB approaches proved to be the most consistent ones in terms of their performances. QoSMT-RR and QoSMT-RUB achieved 56.20% and 66.63% overall QoS increases compared to the baseline-SMT respectively by only losing 6.65% to 7.09% of throughput, in terms of IPC, respectively.

Utility-based approach provided some high peaks compared to its counterpart approaches due to its utilization of resources. However, it suffered from starving resources in colocating threads in some scenarios which tumbled the approach to be less consistent than its counterparts. Round-robin and reduced utility-based approaches on the other hand, did not cause lower quality of service values compared to the baseline SMT at any point except a single outlier between 57 different tests, making them the best scheduling algorithms for this study. Besides, they requires the least amount of additional circuitries, which makes them even better.

The hardware overhead analysis also shown that both of these approaches are immensely lightweight due to their simplistic nature and strong efficacy considering their sizes. These analyses could be extended with assigning the hardware overhead to providing different resources to the core, but this would mean a vast amount of "what if" scenarios that would require a significant amount of builds and tests.

Fairness seems to be the weakest aspect of the proposed designs, since the proposed approaches have tendencies to starve LPTs from time to time, especially with the utility and reduced utility based approaches. Round-robin approach manages to achieve some decent fairness results thanks to its fair approach. On the other hand, our proposed design was always about providing privileges to our HPT over the LPTs; therefore, this metric may be considered as the most negligible one when assessing the proposed designs.

However, even though they have displayed strong solutions, the quality of service predictions did not prove to be completely accurate and the targets for it weren't completely satisfied in all the test scenarios. This shows that this study could be improved with better test sets which would last for longer execution cycles so that target quality of service target satisfaction

could be observed clearer. More importantly, our study could be improved with better performance prediction mechanisms since there is a large room for alternative designs and potential improvements.

In order to achieve this, there are several approaches that can be taken: Improving the calculations and metrics is the obvious way. As the different QoS target tests have shown, the predictors work quite consistently with the varying requirements. Easing the requirements and/or making improvements on our predictors considering the effects of intense demands could be a decent starting point. Also, one of the most obvious of the possible implementations is providing some thread resource substitution mechanisms to the proposed design. So, when HPT asks for resources from the three LPTs but one of them cannot provide any more resources, its resources could be fetched from the second choice LPT. This could improve the reaction time of providing HPT significantly. Besides, for the bottleneck-prone approaches such as utility-based and especially reduced utility-based, it could provide a significant efficacy improvement. Considering that reduced QoS targets positively affect UB and RUB implementations the most, this change could also help with the first-mentioned scenario.

Other and more abstract ideas could include the following: dynamic delta implementation could be tweaked in a way that in a decision time if the chosen LPT lacks adequate resources, it would choose another LPT to complete the required HPT allocation with some additional hardware cost. Another idea would be separating back-end resource partitioning of IQ, ROB, and LSQ for finer resource granularity, again, with additional hardware cost.

Besides these algorithmic tweaks, machine learning algorithms ran with offline training could vastly improve such predictions' accuracy could be a potential improvement as this has been a proven concept within dynamic processing element partitioning within processors [38]. Also, another way to improve would be actually lying within our root study, what QoSMT aimed to surpass, is Cazorla way of executing single-thread real-time data collection cycles which would provide not speculative but a totally measured data regarding the decisions [39].

Lastly, more scheduling algorithms between resource allocations could be explored which would provide further elaboration and possibly could achieve higher success than our proposed approaches.

In light of all these, we can confidently claim that in today's world where data center and cloud applications expand with a rapid pace, there is a significant domain where performance per Watt and TCO per dollar could be and should be improved upon and we aim that our study can be a at least a dim light that would illuminate this vast yet not-yet-adequately-explored region, at least partially.

REFERENCES

1. Data Centres and Data Transmission Networks - International Energy Agency [cited 2023 11 January]. Available from: https://www.iea.org/reports/data-centres-and-data-transmission-networks.

2. Koot M, Wijnhoven F. Usage impact on data center electricity needs: A system dynamic forecasting model. *Applied Energy*. 2021;291:116798.

3. Levy M, Subburaj A. Emerging Trends in Data Center Management Automation. *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. 2021:0480-5.

4. Margaritov A, Gupta S, Gonzalez-Alberquilla R, Grot B. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019:15-27.

5. Jin X, Zhou Y, Huang B, Yu Z, Zhan X, Wang H, et al. QoSMT: Supporting Precise Performance Control for Simultaneous Multithreading Architecture. *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. New York, NY, USA: Association for Computing Machinery. 2019:206–216.

6. Raasch SE, Reinhardt SK. The impact of resource partitioning on SMT processors. *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. 2003:15-25.

7. Alpert D, Avnon D. Architecture of the Pentium microprocessor. *IEEE micro*. 1993;13(3):11-21.

8. Thornton JE. The CDC 6600 Project. *Annals of the History of Computing*. 1980;2(4):338-48.

9. Anderson DW, Sparacio FJ, Tomasulo RM. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*. 1967;11(1):8-24.

10. Pentium® II Processor Developer's Manual. 1997.

11. Burtsev A. Computer Systems Architecture: Out-of-order execution. 2019.

12. Abhishek Desai BM. Issue Queue for a Superscalar Processor. 2003.

13. Baer JL. Reorder Buffer: register renaming and in-order completion. 2007.

14. Bojnordi MN. Out-of-Order Loads/Stores. 2018.

15. Park I, Ooi CL, Vijaykumar TN. Reducing design complexity of the load/store queue. *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*. 2003:411-22.

16. Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*. 1997;17(5):12-9.

17. Tullsen DM, Eggers SJ, Levy HM. Simultaneous Multithreading: Maximizing on-Chip Parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95. New York, NY, USA: Association for Computing Machinery. 1995:392–403. Available from: https://doi.org/10.1145/223982.224449.

18. Nezir U, Lus B, Kucuk G. Improved Resource Scheduling for Lightweight SMT-COP. *2021 6th International Conference on Computer Science and Engineering (UBMK)*. 2021:575-80.

19. Zhang X, Tune E, Hagmann R, Jnagal R, Gokhale V, Wilkes J. CPI2: CPU Performance Isolation for Shared Compute Clusters. *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: Association for Computing Machinery. 2013:379–391. Available from: https://doi.org/10.1145/2465351.2465388.

20. Lo D, Cheng L, Govindaraju R, Ranganathan P, Kozyrakis C. Heracles: Improving resource efficiency at scale. *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015:450-62.

21. Quality of Service (QoS) in High-Priority Applications. Minneapolis, MN: Transition Networks; 2016.

22. Yang X, Blackburn SM, McKinley KS. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016:309-22.

23. Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, et al. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*. 2002;6(1).

24. Liu C, Gaudiot JL. Static partitioning vs dynamic sharing of resources in simultaneous multithreading microarchitectures. *Advanced Parallel Processing Technologies: 6th International Workshop, APPT 2005, Hong Kong, China, October 27-28, 2005. Proceedings 6*. Springer. 2005:81-90.

25. Wang H, Koren I, Krishna CM. An adaptive resource partitioning algorithm for SMT processors. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008:230-9.

26. Koufaty D, Marr DT. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*. 2003;23(2):56-65.

27. Sinharoy B, Kalla R, Tendler J, Eickemeyer R, Joyner J. POWER5 system microarchitecture. *IBM Journal of Research and Development*. 2005 08;49:505 521.

28. Sharkey J, Ponomarev D, Ghose K. M-sim: a flexible, multithreaded architectural simulation environment. *Techenical report, Department of Computer Science, State University of New York at Binghamton*. 2005.

29. Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. *Computer*. 2002;35(2):59-67.

30. Nguyen KT. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. 2016. Available from: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

31. Henning JL. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*. 2006;34(4):1-17.

32. Nezir U. Adaptive Synth-Core: A power efficient synthesis core [Bachelor's Thesis].

2020.

33. Sharkey J, Balkan D, Ponomarev D. Adaptive reorder buffers for SMT processors. *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 2006:244-53.

34. Li D, Rhu M, Johnson DR, O'Connor M, Erez M, Burger D, et al. Priority-based cache allocation in throughput processors. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015:89-100.

35. Wolf C, Glaser J, Kepler J. Yosys-a free Verilog synthesis suite. *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 2013:97.

36. Brayton R, Mishchenko A. ABC: An academic industrial-strength verification tool. *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer. 2010:24-40.

37. Anselmi G, Arora M, Bozhinov I, Das D, Genc T, Grabowski B, et al.. IBM Power E1080 Technical Overview and Introduction. 2021.

38. Yin S, Ouyang P, Tang S, Tu F, Li X, Zheng S, et al. A high energy efficient reconfigurable hybrid neural network processor for deep learning applications. *IEEE Journal of Solid-State Circuits*. 2017;53(4):968-82.

39. Cazorla FJ, Knijnenburg PM, Sakellariou R, Fernandez E, Ramirez A, Valero M. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*. 2006;55(7):785-99.

APPENDIX A: TAG ARRAY IMPLEMENTATION IN SIMULATOR

Algorithm A.1. Tag Array Main Function

```
718     if (context_id < 4)
719     {
720         if (strcmp(cp->name,"dl1")==0) // if dl1 access
721         {
722             indexTA_DL1[context_id]=lookupTA_DL1(tag, set, context_id);
723             if (indexTA_DL1[context_id]<0) {
724               replaceBlockTA_DL1(tag, set, context_id);
725             }
726         }
727         else if (strcmp(cp->name,"il1")==0) // if il1 access
728         {
729             indexTA_IL1[context_id]=lookupTA_IL1(tag, set, context_id);
730             if (indexTA_IL1[context_id]<0)
731                 replaceBlockTA_IL1(tag, set, context_id);
732         }
733         else if (strcmp(cp->name,"ul2")==0) // if l2 access
734         {
735             indexTA_L2[context_id]=lookupTA_L2(tag, set, context_id);
736             if (indexTA_L2[context_id]<0)
737                 replaceBlockTA_L2(tag, set, context_id);
738         }
739     }
```

Algorithm A.2. Tag Array Lookup Function

```
175   int lookupTA_DL1(md_addr_t t, md_addr_t s, int context_id)
176   {
177       int i=0;
178       while (TA_DL1[context_id][s][i] != t && i<8)
179       {
180           i++;
181       }
182       DL1_AA[context_id]++;
183       if (i<8) // Tag Array hit
184       {
185           TS_DL1[context_id][s][i]=++TS_DL1_cnt;
186           DL1_HA[context_id][i]++;
187           //printf("[DL1-%d-%d]", i, context_id);
188           return (i);
189       }
190       return (-1);
191   }
```

Algorithm A.3. Tag Array Replace Function

```
193  void replaceBlockTA_DL1(md_addr_t t, md_addr_t s, int context_id)
194  {
195      int i=1;
196      long min_TS=TS_DL1[context_id][s][0];
197      int min_idx=0;
198      while (i<8)
199      {
200          if (TS_DL1[context_id][s][i] < min_TS)
201          {
202              min_TS=TS_DL1[context_id][s][i];
203              min_idx=i;
204          }
205          i++;
206      }
207      TA_DL1[context_id][s][min_idx]=t;
208      TS_DL1[context_id][s][min_idx]=++TS_DL1_cnt;
209  }
```

APPENDIX B: UTILITY BASED CACHE PARTITIONING IMPLEMENTATION IN SIMULATOR

Algorithm B.1. Utility-Based Cache Partitioning Main Function

```
// THESIS : Cache partitioning scheme : START
// Allocate 1 initial way to each thread (DL1)
int greedy_util[4] = {CACHE_UTIL_DL1[0][0], CACHE_UTIL_DL1[1][0], CACHE_UTIL_DL1[2][0], CACHE_UTIL_DL1[3][0]};
int max_util_tracker = -1;
int max_util = 0;
int thr_way_it[4] = {[0 ... 3] = 1}; // Compatible with gcc. May cause problems with other compilers.

for (way_it = 4; way_it < 8; way_it++) {
  if ((CACHE_UTIL_DL1[0][thr_way_it[0]] - greedy_util[0]) >= (CACHE_UTIL_DL1[1][thr_way_it[1]] - greedy_util[1])) {
    max_util = CACHE_UTIL_DL1[0][thr_way_it[0]] - greedy_util[0];
    max_util_tracker = 0;
  }
  else {
    max_util = CACHE_UTIL_DL1[1][thr_way_it[1]] - greedy_util[1];
    max_util_tracker = 1;
  }

  if (max_util < (CACHE_UTIL_DL1[2][thr_way_it[2]] - greedy_util[2])) {
    max_util = CACHE_UTIL_DL1[2][thr_way_it[2]] - greedy_util[2];
    max_util_tracker = 2;
  }

  if (max_util < (CACHE_UTIL_DL1[3][thr_way_it[3]]) - greedy_util[3]) {
    max_util = CACHE_UTIL_DL1[3][thr_way_it[3]] - greedy_util[3];
    max_util_tracker = 3;
  }

  greedy_util[max_util_tracker] = CACHE_UTIL_DL1[max_util_tracker][thr_way_it[max_util_tracker]];
  thr_way_it[max_util_tracker]++;

  for (context_it = 0; context_it < 4; context_it++) {
    CACHE_PARTITION[context_it] = thr_way_it[context_it];
  }

  /*printf("Cache Partition: ");
  for (context_it = 0; context_it < 4; context_it++) {
    printf("%d, ", CACHE_PARTITION[context_it]);
  }
  printf("\n");*/
}
```

Algorithm B.2. Utility-Based Cache Partitioning Sampling Function

```
6973    // THESIS : Run the sampling function for Utility-Based Cache : BEGIN
6974    int context_it = 0;
6975    int way_it = 0;
6976    int ha_way_it = 0;
6977    for (context_it = 0; context_it < 4; context_it++) {
6978      for (way_it = 0; way_it < 8; way_it++) {
6979        for (ha_way_it = 0; ha_way_it <= way_it; ha_way_it++) {
6980          CACHE_UTIL_DL1[context_it][way_it] += DL1_HA[context_it][ha_way_it];
6981        }
6982      }
6983    }
6984    for (context_it = 0; context_it < 4; context_it++) {
6985      for (way_it = 0; way_it < 8; way_it++) {
6986        for (ha_way_it = 0; ha_way_it <= way_it; ha_way_it++) {
6987          CACHE_UTIL_IL1[context_it][way_it] += IL1_HA[context_it][ha_way_it];
6988        }
6989      }
6990    }
6991    for (context_it = 0; context_it < 4; context_it++) {
6992      for (way_it = 0; way_it < 8; way_it++) {
6993        for (ha_way_it = 0; ha_way_it <= way_it; ha_way_it++) {
6994          CACHE_UTIL_L2[context_it][way_it] += L2_HA[context_it][ha_way_it];
6995        }
6996      }
6997    }
6998    // THESIS : Run the sampling function for Utility-Based Cache : END
```

Algorithm B.3. Utility-Based Cache Partitioning Restriction Function

```
824     // UL2 partitioning below:
825     else if (strcmp(cp->name,"ul2")==0) {
826       if (CACHE_THREAD_ALLOC_L2[set][context_id] < CACHE_PARTITION_L2[context_id]) {
827         // Execute selective cache replacement from a non-current thread line.
828         while (cache_it != NULL) {
829           if (cache_it->context_id != context_id) {
830             break;
831           }
832           if (cache_it->way_prev == NULL) {
833             break;
834           }
835           cache_it = cache_it->way_prev;
836         }
837         repl = cache_it;
838
839         // Decrement the thread allocation value for the evicted addr
840         CACHE_THREAD_ALLOC_L2[set][cache_it->context_id]--;
841         // Increment the thread allocation value for the write addr
842         CACHE_THREAD_ALLOC_L2[set][context_id]++;
843       }
844       else {
845         // The thread allocation for the current thread is full. Execute a selective cache replacement.
846         while (cache_it != NULL) {
847           if (cache_it->context_id == context_id) {
848             break;
849           }
850           if (cache_it->way_prev == NULL) {
851             break;
852           }
853           cache_it = cache_it->way_prev;
854         }
855         repl = cache_it;
856       }
857     }
858     else {
859       repl = cp->sets[set].way_tail;
860     }
```