DYNAMIC CAPPING OF WAITING INSTRUCTION BUFFER IN SIMULTANEOUS MULTI-THREADING PROCESSORS

by
Harun Güngörer

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
2022

DYNAMIC CAPPING OF WAITING INSTRUCTION BUFFER IN SIMULTANEOUS MULTI-THREADING PROCESSORS

APPROVED BY:

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

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

Prof. Dr. Haluk Rahmi Topçuoğlu ……
(Marmara University)

DATE OF APPROVAL: ..../..../2022

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 Harun Güngörer

Signature …..…….……..

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my Professor Gürhan Küçük. Pursuing my thesis under her supervision has been an experience which broadens the mind and presents an unlimited source of learning.

Finally, I would like to thank my family for their endless love and support, which makes everything more beautiful.

ABSTRACT

DYNAMIC CAPPING OF WAITING INSTRUCTION BUFFER IN SIMULTANEOUS MULTI-THREADING PROCESSORS

In simultaneous multi-threaded (SMT) systems, processors are designed to use instruction-level parallelism per thread to improve the performance of traditional superscalar processor systems. Multiple threads compete for the resources of processors to issue instructions in each cycle is the main reason for higher instruction throughput for SMT systems. Considering that threads compete for resources of the processor, utilization of critical shared resources and key datapath components is very important in SMT systems. One of the most important shared resources in SMT systems is the Waiting Instruction Buffer (WIB) or a.k.a. the Issue Queue (IQ). We need to control the WIB entry allocations of threads to be able to prevent threads from excessively allocating WIB entries which leads to decreases in overall throughput. Therefore, we determine a capping value to control the WIB entry allocation of threads. There are several approaches for resource allocation and utilization such as setting pre-defined capping values and dynamically calculating capping values for the system. In this paper, we propose a dynamic calculation of the capping value for each thread separately by using an efficiency metric based on committed instructions per resource entry. We also compare our method with previous studies which utilize WIB allocation by setting global capping value for threads in particular window sizes. With our proposed method of dynamically calculating optimum cap value, we increase the overall instruction per cycle (IPC) value and also keep fairness metric stable. For the 4-threaded system, we improve the overall IPC value of the system by an average of 7.46% and for the 8- threaded system we improve it by 9.2% while achieving better fairness result than the Lin’s study.

ÖZET

EŞ ZAMANLI ÇOKLU İŞ PARÇACIKLI İŞLEMCİLERDE KOMUT BEKLEME ARA BELLEĞİNİN DİNAMİK OLARAK SINIRLANDIRILMASI

Eşzamanlı çok iş parçacıklı sistemlerde (EÇİP) işlemciler, geleneksel süperskalar işlemci sistemlerinin performansını iyileştirmek için iş parçacığı başına komut düzeyinde paralellik kullanacak şekilde tasarlanmıştır. EÇİP sistemlerinin daha yüksek komut verimine sahip olmalarının ana nedeni birden çok iş parçacığının işlemcilerin kaynakları için her döngüde komut yayınlamak için rekabet etmesidir. İş parçacıklarının işlemcinin kaynakları için rekabet ettiği düşünüldüğünde, EÇİP sistemlerinde paylaşılan kritik kaynakların ve veri yolu bileşenlerinin etkili kullanımı çok önemlidir. EÇİP sistemlerinde en önemli paylaşılan kaynaklardan biri Bekleyen Komut Belleğidir (BKB). BKB alanlarının kontrolsüz ve aşırı bir şekilde iş parçacıkları tarafından tahsis edilmesi, sistemin genel veriminde düşüşlere yol açmasından dolayı, bu BKB alanlarının iş parçacıkları tarafından tahsislerini kontrol altına alınması gerekmektedir. Bu nedenle, iş parçacıklarının BKB alanlarını tahsis etmesini kontrol etmek için bir sınır değeri belirlenmesi gerekmektedir. Kaynakların tahsisi ve etkili kullanımı için önceden tanımlanmış sınır değerlerinin ayarlanması, sistem için sınır değerlerinin dinamik olarak hesaplanması şeklinde çeşitli yaklaşımlar bulunmaktadır. Bu tezde, kaynak tahsisi başına taahhüt edilen komuta dayalı verimli bir ölçüm kullanarak, her bir iş parçacığı için ayrı ayrı sınırlama değerlerinin dinamik bir şekilde hesaplanması önerilmektedir. Ayrıca bu tezde sunduğumuz yöntemimizi, belirli zaman aralıklarında iş parçacıkları için genel sınır değeri atayarak, BKB alanlarının tahsisinin etkili bir şekilde yapılmasını sağlayan önceki çalışmalarla karşılaştırmaktayız. İş parçacıkları için optimum üst sınır değerini dinamik olarak hesaplamak için önerdiğimiz yöntemimizle, genel döngü başına komut (DBT) değerini artırıyor ve aynı zamanda iş parçacıkları arasındaki adaleti korumaktayız. 4 iş parçacıklı sistem için sistemin genel DBT değerini ortalama % 7,46 ve 8 iş parçacıklı sistem için adaleti Lin'in çalışmasından daha iyi bir seviyede tutarken DBT değerini ortalama % 9,2 iyileştirmiş bulunmaktayız.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

AC	Adaptive capping
ACNL	Adaptive capping no limit
Ar	Adjustment result
ARPA	Adaptive resource partition algorithm
CIPRE	Committed instruction per resource entry
Cp	Current performance
DCRA	Dynamically controlled resource allocation
HPIWW	History-based predictive instruction window weighting
ILP	Instruction level parallelism
IPC	Instruction per cycle
IQ	Issue queue
Pp	Previous performance
RF	Register file
ROB	Re-order buffer
Sd	System data
SIWW	Speculative instruction window weighting
SMT	Simultaneous multi-threading
TLP	Thread level parallelism
WIB	Waiting Instruction Buffer

1. INTRODUCTION

Unlike traditional superscalar processors which use single thread instruction-level parallelism (ILP), simultaneous multi-threading (SMT) processors introduce multi-threading by utilizing ILP with thread-level parallelism (TLP).

In traditional superscalar processors, functional units are not utilized efficiently since one thread is running at a time, and also switching between threads causes overhead in the CPU because of operating system intervention. In simple multi-threaded processors, despite the fact that task switching by the operating system is eliminated, common shared resources and functional units are not fully utilized because a single thread does not have sufficient ILP. Unlike SMT processors, both coarse-grained multi-threaded processors and fine-grained processors do not allow to issue of instructions from different threads in the same clock cycle [1]. However, while interleaving of instructions from other threads is not allowed in course-grained multi-threaded processors, cycle to cycle interleaving is allowed in fine-grained processors. To fully exploit the potential of common shared resources, SMT processors allow issuing instructions from different threads in the same clock cycle.

Moreover, SMT systems require less hardware than using several superscalar machines for similar overall performance thanks to sharing of key datapath components. Control-complexity-wise easier and cost-wise better to share components are the most common shared resources in SMT systems. In Figure 1.1, a common pipeline stage organization of the SMT system is represented with functional units and common shared resources.

In SMT processor design, in order to improve resource utilization, multiple threads share the key datapath components. Despite the fact that sharing these resources increase the overall performance of the system, it brings up extra complexity to managing the critical path operations. To be able to retain profiting from the gain achieved by using instruction-level parallelism with thread-level parallelism, a valid solution must be provided to reduce the complexity of sharing common resources. Short latency hazards can be easily masked by executing instructions from different threads thanks to the ability that allows processors to choose different threads in each clock cycle. This technique solves the issue that threads with low instruction-level parallelism prevent the processor from exploiting ILP by using thread-

level parallelism to compensate for delay slots with instructions from different threads. Therefore, it prevents pipeline stalls and improves overall system performance.

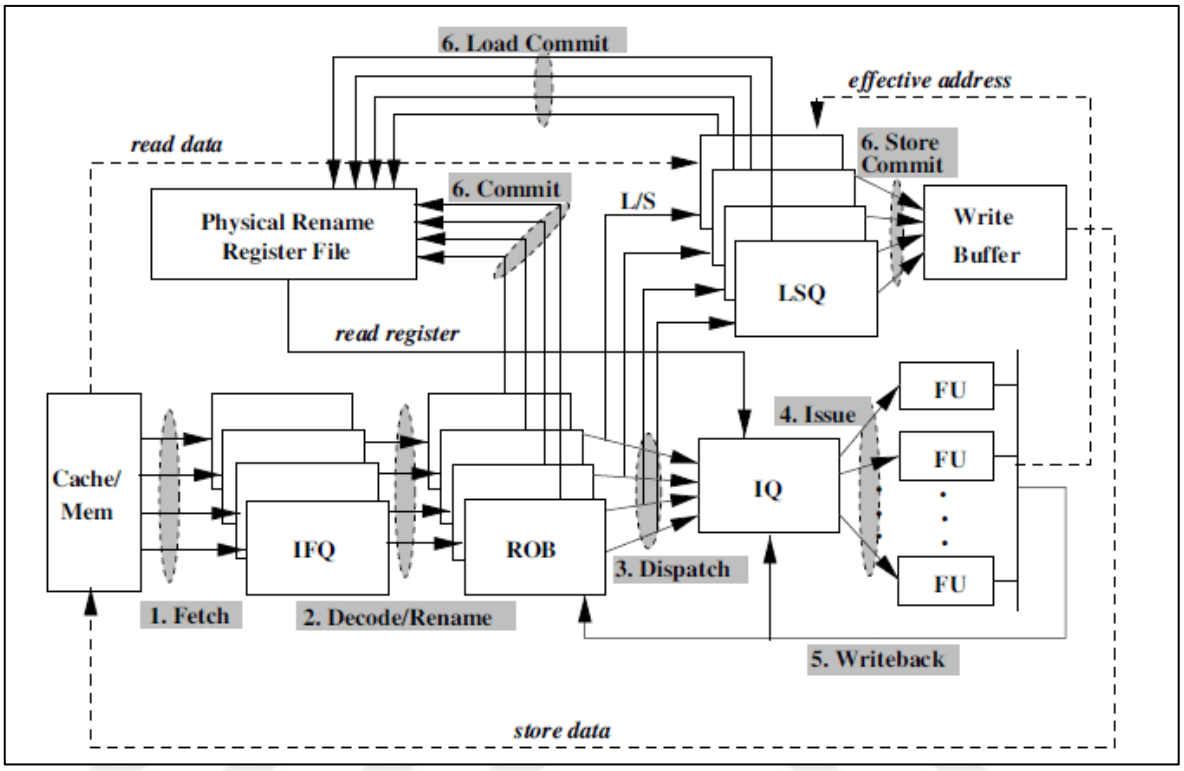

Figure 1.1. Pipeline stages in a 4-threaded SMT system [1]

SMT systems show better overall performance than other architectures by exploiting instruction-level parallelism and thread-level parallelism together. However, starvation and bottleneck cannot be avoided because of the sharing of common resources without a proper intelligent distribution mechanism. Therefore, to provide a fair and effective distribution of resources among threads, valid and accurate intelligence must be integrated into this sharing mechanism.

Lately, there have been several studies published that focus on improving overall performance in SMT systems by adjusting different stages in the critical path. Predicting transient values from misprediction to avoid register allocation, deallocating registers with cache misses, giving higher fetching priority in pre-issue stages if the thread has fewer instructions, introducing a new fetch policy about memory-level parallelism, and capping the number of write buffer entries that threads can allocate for the better distribution of resources are some of the topics that researchers focus on to improve performance [2]. For example, To solve the issue of L2 cache misses, STALL and FLUSH [3] adopt fetch policies;

to exploit parallelism free of the stalled memory operations, a dynamically fetch policy is proposed [4] which is focused on the memory performance of threads, and many other techniques proposed in the past to solve allocation of resources according to some real-time parameters and system-specific settings. There is no specific technique that brings solutions without extra hardware implementation and aims to solve issuing stage problems among these techniques mentioned before.

There are alternative proposed methods to utilize WIB along with capping distribution of WIB entries among threads, recalling stalled instructions from the WIB, and reducing the miss-speculation-based flush-out instructions. With these kinds of approaches, performance improvement can be provided on a predefined target of system settings with a specific set of workloads. When the system settings and workload profiles are changed, the expected improvement in the systems cannot be provided by these approaches. Because, all of the techniques mentioned before using a global cap value for each thread or a fixed capping technique, and none of them consider the real-time performance of each thread to allocate WIB entries.

In this paper, we aim to develop an algorithm that works on different workloads and system settings without the need of knowing the details of these systems beforehand. Instead of the increasing size of the shared resources, we want to provide an adaptive algorithm that controls the allocation of the shared resources. Although increasing the size of the shared resources is a simple solution, it cost more storage space and consumes more energy when we increase the size of the shared resources. We aim to improve the overall system performance without sacrificing more space and energy by proposing a solution for distributing the shared resources dynamically.

We propose a dynamic allocation algorithm to set the capping value for each thread separately by calculating the throughput performance of each thread on specific window size. This proposed technique provides enhanced utilization of WIB entries among threads compared to other fixed or global capping approaches. Moreover, we improve overall system performance in terms of total throughput and fairness. By controlling the allocation of the shared resources, we make sure the threads get the resource entries they need and keep the fairness among threads by preventing the overallocation of the resource. We provide high total throughput performance for the system by optimizing the utilization of the shared resources.

2. BACKGROUND

In the traditional superscalar processors, even though four or more instructions per cycle can be executed, only one or two instructions are executed since low instruction-level parallelism of applications. There is a suggested solution for better performance which is using several superscalar processors on a chip. However, this is not seen as an efficient answer because, besides low instruction-level parallelism, performance is reduced when there is little thread-level parallelism. Therefore, developing a processor that can utilize different types of parallelism at the same time provides a better solution. SMT processors utilize both thread-level parallelism and instruction-level parallelism. Multi-threading, parallel programs, or independent applications in multiprogramming workload are resources of thread-level parallelism.

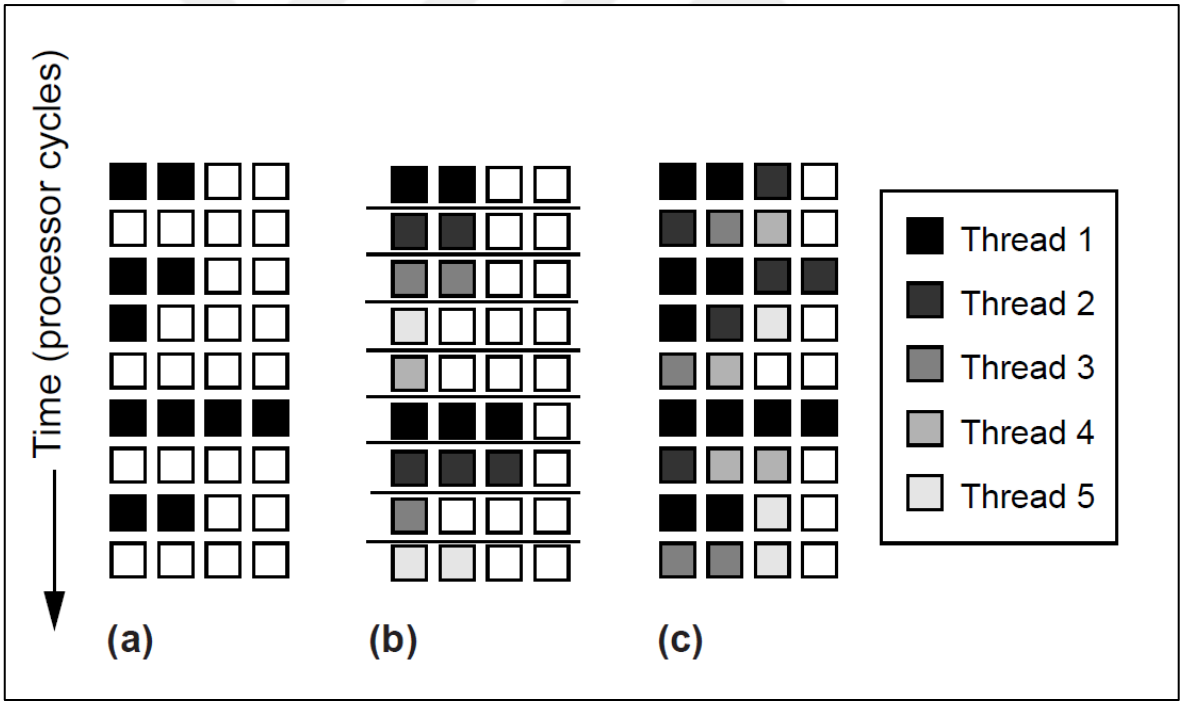

Figure 2.1. Partition of issue slots in different architectures: a superscalar (a), a multithreaded superscalar (b), and an SMT processor(c) [5]

In figure 2.1, we can see the difference between three processor designs which are superscalar processor, multithreading processor, and SMT processor design. This figure shows us a piece of the execution order of these three architectures. Rows show the issue

slots of an individual execution cycle. In a single cycle, if an instruction is found to be executed, is it represented with a filled box in that issue slot.

On the other hand, unused slots are shown as empty boxes. The unused slots are defined as vertical or horizontal waste. Generally, poor instruction-level parallelism causes horizontal waste. Vertical waste can be seen in the case of a completely unused cycle which happens due to a long latency instruction that prevents issuing new instructions [5].

A conventional superscalar processor design is shown in Figure 2.1.a. A single program is executed to find instructions to issue. Vertical waste and horizontal waste occur when instructions are not found since this causes unused issue slots.

In Figure 2.1.b, a sequence of a multi-threaded process architecture is represented. Hardware states are included for multiple threads in this multi-threaded processor and instructions from one of the threads are executed on every cycle. The main advantage of multi-threaded processor architecture is the high tolerance of long-latency operations which eliminates the vertical waste but not the horizontal waste. Therefore, like superscalar processors, multi-threaded processors become restricted by the instruction-level parallelism while the instruction issue width keeps expanding.

How SMT processors find instructions from different threads to execute can be seen in Figure 2.1.c. Instruction-level parallelism is utilized by gathering instructions from any thread that can be issued. Then resources are dynamically scheduled among all instructions to achieve the highest possible resource utilization. Multiple threads with low instruction-level parallelism can be executed to improve the performance. Therefore, both horizontal and vertical waste problems are solved by recovering empty issue slots as shown in the figure.

Sharing key datapath components among multiple threads enhances resource utilization, and this is one of the main advantages of the SMT processors. Furthermore, all thread-specific datapath components are presumed to remain in per-thread ownership. These hardware components generally remain busy to hold more instructions from threads considering the requirement in resource sharing. Despite sharing of these resources assuring the overall performance increase in SMT systems, it cost extra complexities in administering processors cycle and the critical path. One of the common shared resources is the waiting instruction buffer (WIB) and it pretends like a buffer for the functional units that execute on the

instruction operands. The over-consumption of this shared resource leads to starvation among threads and a decrease in overall performance.

There are many kinds of research that target to improve the performance of SMT systems via different scheduling algorithms, system settings, and real-time parameters. The real-time parameter can be based on different metrics such as how many cycle instructions stall in buffers, comparing resource allocation of threads or instructions after and before the issue stage. These kinds of parameters are used to adjust resource allocation and distribution among threads. To achieve optimal system performance, the adjustment size for allocation of resources is important and hard to decide in all systems.

Improvement in system performance mostly comes from these aforementioned techniques on specific system settings with a set of different workloads. The performance gained by these techniques does not always provide the expected amount of improvement when the system settings and workload profiles are changed. A set of criteria and pre-adjusted settings that works efficiently on every system setting and workload combination cannot be determined definitely. In this paper, we aim to develop an allocation technique to dynamically allocate one of the common shared resources independent of what system setting and what kind of workload profile is used.

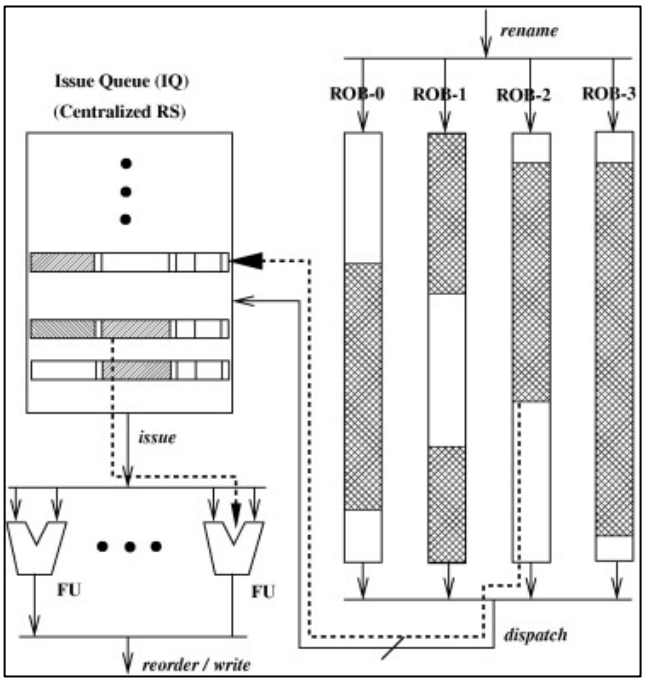

Figure 2.2. Dispatch stage of an SMT system [11]

In Figure 2.2, we can see the dispatch stage of an SMT system. The instruction dispatch stage is one of the most important stages in the system since it sends instructions to a shared resource that has high competition for allocation. Moreover, a bottleneck can be turned up quickly in the pipeline which can affect the overall system performance negatively. Some studies focus on the dispatch stage concerning instruction recalling of threads that are inactive and controlling the number of instructions a thread can allocate in the waiting instruction buffer.

A fixed capping value for WIB entries allocated by threads is one of the proposed techniques in the past [6]. With the optimal cap values obtained from the different benchmarks and system testing, a considerable performance improvement can be achieved. However, various cap values can be observed for different workloads and systems and this leads to inconsistencies in performance gain when the system settings are changed significantly. In the light of these findings, a dynamic and self-determining algorithm is proposed to overcome this system setting dependency. The proposed algorithm observes the system performance and resource utilization and then calculates the optimal cap value for each thread separately in specified window sizes, continuously. As it can be seen in the following chapters, the simulation results show significant improvement in overall system performance for different system settings and workloads.

2.1. LIN'S STUDY

In the previous chapter, we discussed that the instruction dispatch stage is important for the system to run efficiently. Also, we mentioned that some studies are focusing on the dispatch stage and providing a solution for threads that control unused space in resources such as the waiting instruction buffer (WIB).

Zhang et al. have one of the studies that we mentioned above focuses the WIB and proposes a mechanism to improve the overall system performance. They propose a dynamic cap value for all threads and with this given cap value, they try to achieve efficient usage of the WIB.

The proposed autonomous capping system observes the system performance periodically and calculates a performance metric based on the dispatching activity. Then, they compare the monitored performance of the current and previous windows. After the comparison, they

decide on the adjustment direction. If the result of the comparison shows an improvement, then they propose to keep the same adjustment direction otherwise they make the adjustment direction opposite direction of the current direction. This process is repeated periodically to achieve the optimal cap value that produces maximum performance output autonomously.

In Figure 2.3, the dispatching design of the proposed autonomous capping process in four threaded systems is represented. It is stated that no more instruction is dispatched when the number of instructions in the WIB of a thread reaches the number of cap value given to that thread.

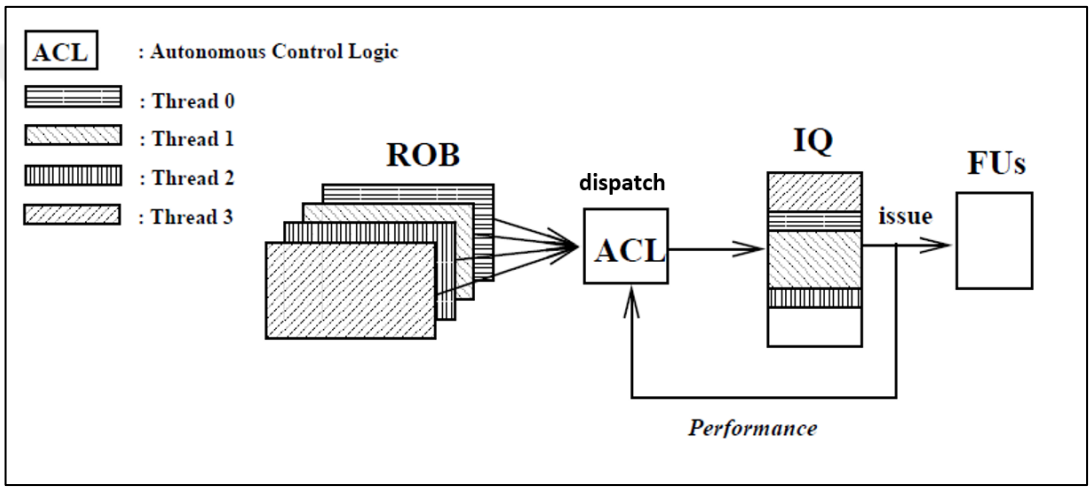

Figure 2.3. Auto capping design for SMT system [1]

This cap value is decided as a result of this control logic, and it is used as a global cap value for all the threads for the next sampling window. In this study, the issue rate is used as a performance indicator as shown in Figure 2.3. The number of instructions issued from WIB is described as the issue rate which is one of the performance indicators mentioned previously.

The window size in this proposed method is set to 1000 clock cycles due to two concerns, which are cache misses and misspeculation. These two concerns are the reasons for sudden changes in the behavior of threads. There is also a threshold set for avoiding adjustment in case of non-significant changes in the performance of threads. This threshold is determined as three percent of the performance of the previously calculated sampling window.

The adjustment logic described in the study is that if the difference between the current performance and the previous performance is bigger than the threshold value of the recently

adjusted window, the adjustment direction is not changed. If the difference between the previous and the current performance values is bigger than the threshold value of the recently adjusted window, the adjustment direction is changed.

Lastly, it is stated that if the absolute value of the difference between the current and the previous performance is smaller than the threshold value of the previous performance value, there will be no adjustment in the process.

In Algorithm 2.1, it can be seen how the baseline study was implemented to compare our proposed technique. The qualified cycle and the number of issued instructions are the most important two parameters of the algorithm. The qualified cycle is started with 1000 as stated in the proposed design and decreases only when the cycle included a minimum of one thread that dispatches. The number of issued instructions is another crucial parameter in this design since it is used for calculating performance indicators. We set the number of issued instructions to zero at the beginning of the program then we increase it by 1 in every cycle when an instruction is issued. After calculating these two parameters, we perform the control logic every thousand cycles as stated in the proposed study.

In the autonomous control logic, first, the current performance is calculated by dividing the number of issued instructions by the number of qualified cycles. We calculate the difference between the current and the previous performance of sampling windows. Also, the threshold value of the previous performance values is calculated. Then, we control the qualified cycle if it is reduced to zero then we pass this control cycle. Otherwise, we continue the proposed algorithm and control if the absolute value of the difference between the current and the previous performance metrics is greater than the threshold of the previous performance metric which we calculate at the beginning of the control logic. If it is not greater then, we do not adjust the cap values of threads.

On the other hand, if the value is greater, we keep moving through the control logic. If the difference between the current performance and the previous one is greater or equal to the threshold value of the previous performance value, we increase the global cap value which is used for all threads for the next window. However, if the difference between the previous performance and the current performance is greater than the threshold value of the previous performance then we decrease the global cap value. There is more control logic we add to this algorithm to handle various corner cases. The first one is that we do not increase the

global cap value if the value exceeds the size of WIB. Also, we check if the current global cap value is reduced to one then we do not decrease the value to keep its minimum value at one. After we adjust the global cap value, then we set this value to every thread for the next window.

Algorithm 2.1. The pseudocode of the Lin's algorithm

```
1: init qualifiedCycle, numOfIssuedInst, threshold
2: init currentPerformanceMeasurement, previousPerformance
3: init cMinusp, threshMultiplypDiv100, pMinusc
4: currentPerformanceMeasurement = numOfIssuedInst / qualifiedCycle
5: cMinusp = currentPerformanceMeasurement - previousPerformance
6: threshMultiplypDiv100 = threshold * previousPerformance / 100
7: pMinusc = previousPerformance - currentPerformanceMeasurement;
8: if qualifiedCycle >  0 then
9:   if fabs(cMinusp) >= threshMultiplypDiv100 then
10:       if cMinusp >= threshMultiplypDiv100 then
11:        if globalCappingValue < WIBSize then
12:                globalCappingValue ++
13:                previousPerformance = currentPerformanceMeasurement
14:       else if pMinusc >= threshMultiplypDiv100 then
15:           if globalCappingValue > 1 then
16:             globalCappingValue --
17:             previousPerformance = currentPerformanceMeasurement
18:       for (i = 0 to numThreads)
19:           Threads[i].capSize = globalCappingValue
```

3. RELATED WORK

In this chapter, we review the previous work which studies similar issues on resource distribution on simultaneous multi-threaded systems. Moreover, there are many studies that especially focus on allocation and distribution of waiting instruction buffer (WIB) entries to improve the overall system performance of simultaneous multi-threaded (SMT) systems.

Zhang et al. propose an adaptive capping technique to control the allocation and distribution of WIB entries in SMT systems. The proposed technique is an autonomous adjustment method for controlling WIB distribution in real-time. The adjustment decision is made by observing the performance of the current and the previous sampling windows of the program. The performance of the system is measured according to the number of instructions issued from the WIB [1].

Cazorla et al. propose a dynamically controlled resource allocation (DCRA) policy for the resource allocation in SMT systems. In this policy, each thread's resource usage is observed and the distribution of critical shared resources among the threads is controlled by the policy. Moreover, there is a mechanism defined for threads to make them borrow resources from other threads that do not need that kind of a resource. As a result of this borrowing mechanism, the underuse of shared resources is significantly reduced.

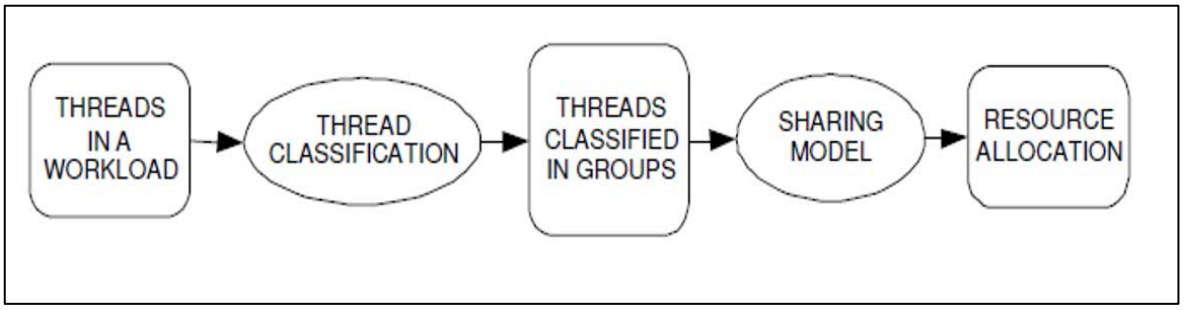

Figure 3.1. DCRA policy workflow [4]

In this DCRA policy, first, the threads are classified according to their resource requirements [4]. The distribution of resources among the threads is decided by the DCRA policy based on the previous classification. Resource usage is directly monitored by the DCRA then it detects the overallocation of resources and stalls the thread until it is no longer overuse those resources. In figure 3.1, we can see the workflow of the proposed DCRA policy. Considering

different execution phases of threads is important to implement an effective resource allocation. It is stated that many of the threads show various behavior patterns, and it causes changes in their needs for resources. There are important steps in this policy, which are thread classification, thread-phase classification, resource usage classification, and the sharing model. In thread classification, threads are classified according to the number of resources needed for exploiting ILP efficiently. Also, it is considered which shared resource is actually used by the thread. It is a dynamic and continuous process of evaluation to keep up with the changes in the thread's behavior. Therefore, the critical shared resources are allocated by the threads that actually need resources.

The other important step is the thread-phase classification. Threads are not classified for their all lifetime but phases of execution of threads are classified by collecting changes in the thread's need for resources. It is also stated that a thread's resource requirements are not necessarily needed to be known by the policy. Threads are classified into different groups which are slow and fast groups according to the cache behavior of threads. Threads run slower, and resources are not released for a long time when cache misses occurs. Moreover, allocated WIB entries are held by the instructions that depend on the missing load until the resources are released and the missing load is committed.

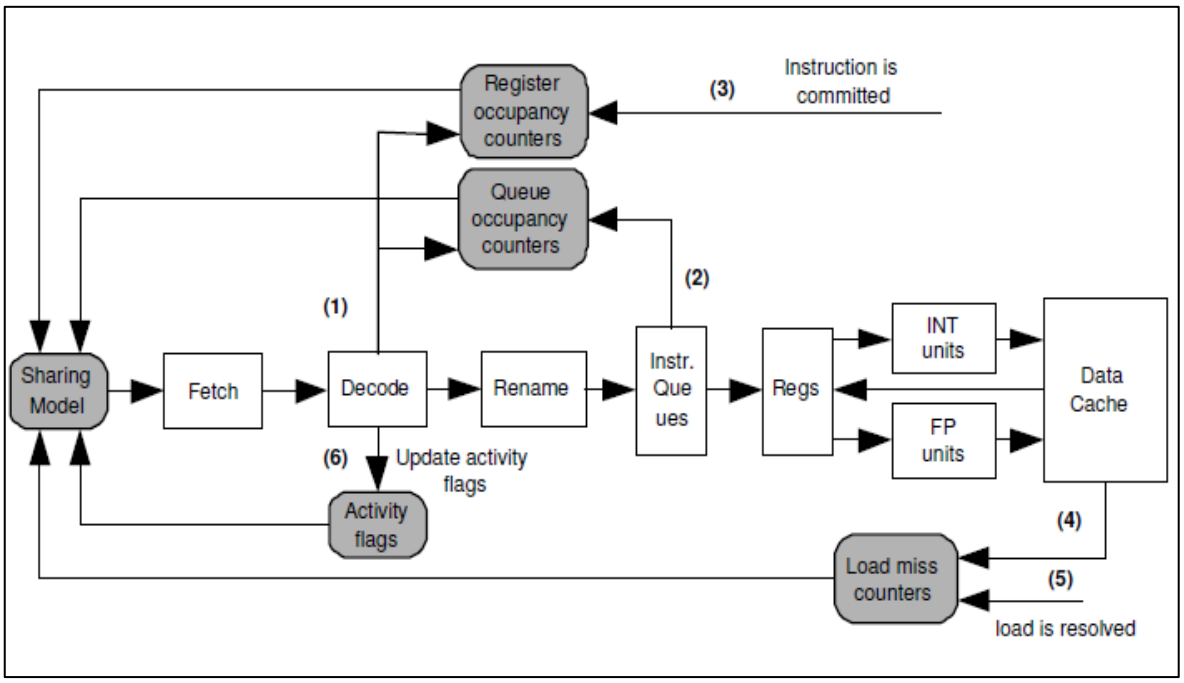

Figure 3.2. Implementation of the dynamic allocation policy [4]

In the resource usage classification phase, according to various resources of processors, threads are classified as active and inactive threads. Because all available resources are not utilized by all threads in their lifetime. The distribution of shared resources among threads that do not need that type of a resource significantly reduces the system performance and leads to resource scarcity. Therefore, threads that use the given resources are classified as active and if the given resource is not used after a specific time, these threads are classified as inactive threads. It is assumed that a thread does not need that resource when a thread is classified as an inactive thread for that given resource. Then the share of that thread is distributed among other threads that need that resource. The implementation of the allocation policy can be seen in Figure 3.2.

Wang et al. propose an adaptive resource partition algorithm (ARPA) that allows the distribution of shared resources among the threads based on changes in the behaviors of threads [7]. The algorithm examines the resource usage of threads in specific periods and determines the threads which can use resources more efficiently. The main goal of the ARPA is to improve overall system performance by improving resource utilization.

Zhang et al. propose a technique to enhance the system performance by recalling instructions of idle threads [8]. The WIB is one of the important shared resources, and, with this technique, the stalls of idle threads on resources like WIB can be reduced significantly. It is also stated that there is no significant hardware overhead or negative effect on clock timing.

There is also another study from Zhang et al. that focus on controlling the distribution of write buffer entries among threads [9]. In this proposed method, a simple control mechanism for the assignment of the write buffer is added to the default system's commit stage. The main purpose of the system is to prevent the overuse of the write buffer. A cap value is introduced to limit the write buffer entries that a thread can allocate. Threads stop committing instruction when the number of entries occupied by the thread reaches the cap value. In Figure 3.3, the proposed technique is shown as a modified commit algorithm. The write buffer, the write port, and the committed bandwidth are shared resources that may cause interruption for a write instruction that goes through the reorder buffer. If it is not managed, the waiting for the write buffer happens much more frequently than other delays. It is stated that all of the methods for eliminating the overuse of shared resources will have a disadvantage in exploiting the adaptability offered by shared resources.

The setting of the cap value is the main reason for agreement between the advantages and disadvantages of the proposed technique. Controlling the overuse of resources by a thread will not be effective if a cap value that is too high is determined. Furthermore, if there are not enough concurrent writes from different threads due to the setting of a cap value that is too low, the overall system performance is significantly reduced.

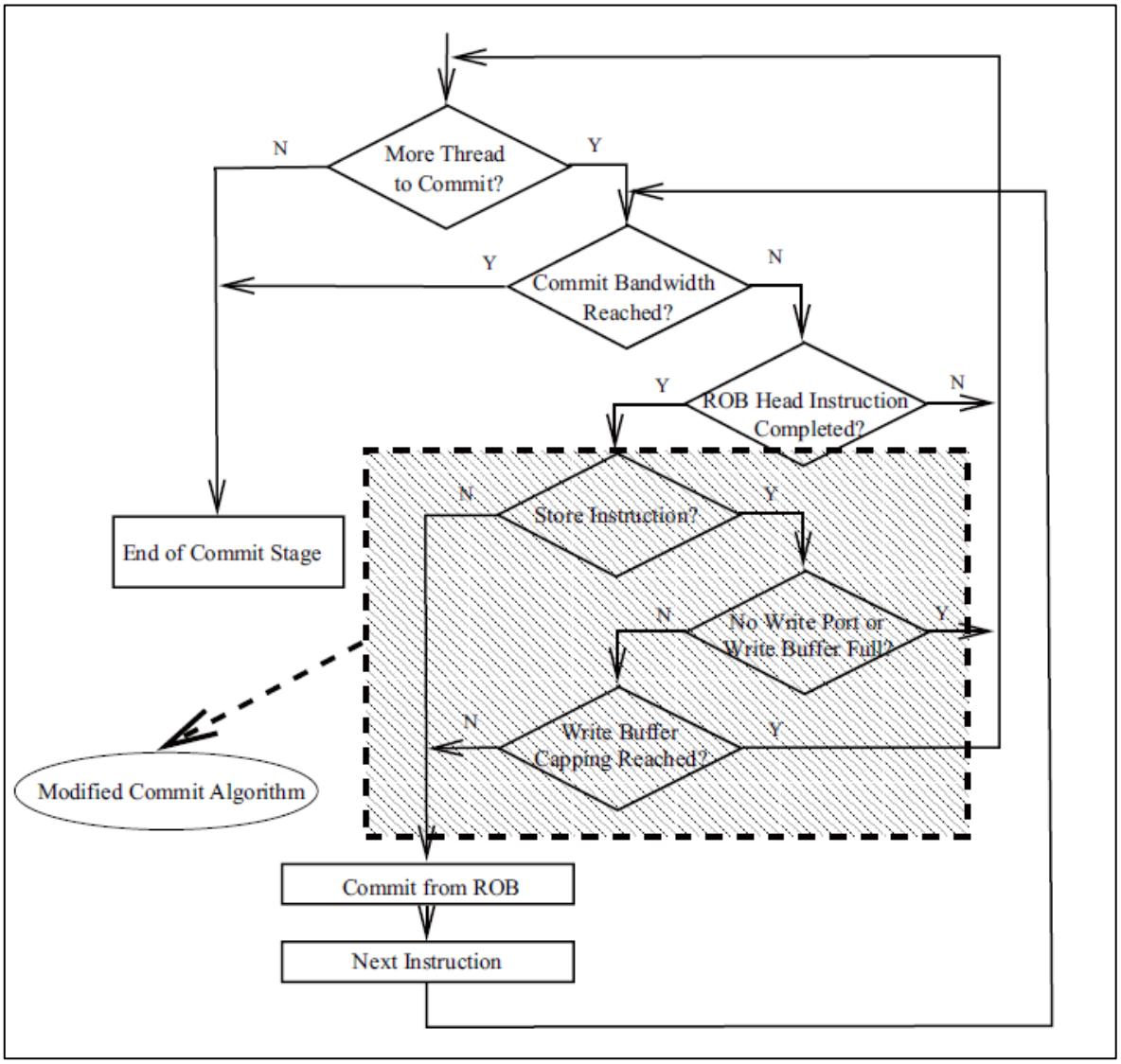

Figure 3.3. Flow chart of the commit algorithm [9]

Sahba et al. propose an algorithm based on the instruction dispatching and issuing activeness to adjust the distribution of WIB entries among threads. By observing these issuing and dispatching instruction activities, a cap value is set for each thread. The cap-value assignment is determined according to the prediction of the thread's demand for WIB in the next coming time period [2]. Predicting the activeness of threads in the next cycles is the

foundation of the proposed method for determining an optimized cap value for threads. Monitoring the dispatched instructions and issued instructions in the current period are the two indicators of the activeness of a thread. To set the cap values of threads, these activeness indicators are used. Setting a cap value for each thread and calculating the cap value based on the real-time data of the thread are the two major advantages of the proposed technique over the fixed capping technique. There are four states where a cap value can be set in the state machine, as can be seen in Figure 3.4. The changes in the activeness of threads make a transition between the states and cause a new cap value to be set for threads.

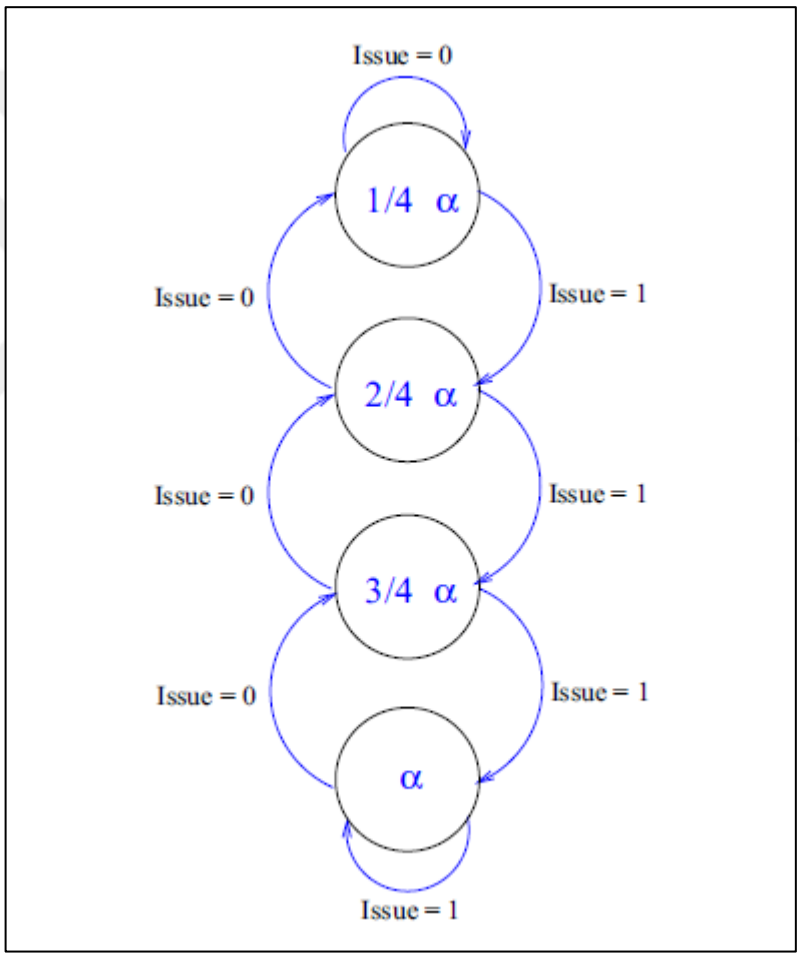

Figure 3.4. State machine for activeness capping [2]

Carroll et al. propose a technique that aims to control and decrease the overuse of the WIB to improve the system performance in SMT systems. To improve the system performance, they limit the number of entries that a thread can allocate from the WIB. They introduce two different cap values, which are the static cap and the slow cap values [10]. It is stated that the static cap value is a permanent limit for threads on the allocation of WIB entries. The

slow cap is set for threads that are classified as slow threads according to their resource utilization.

In another study, Sahba et al. propose a dynamic algorithm to adjust the limit of threads at run-time based on the number of individual memory instructions of each thread. A cap value is set for threads after a memory instruction is dispatched from the WIB to improve the utilization of the resource. It is stated that this technique shows a higher utilization of the WIB than that of the fixed capping approach [11]. Moreover, minimum extra hardware is necessary for the implementation of this algorithm to improve the overall performance of the system.

Zhang et al. propose a global resource utilization method for the most important shared resources, as they claim, in SMT systems which are the WIB, the write buffer, and the register file. All the resources are controlled based on their sizes and proportionally distributed among the threads. It is stated that the proposed method does not add much extra control logic and overhead for hardware. By using this thread, they claim that the instruction per cycle (IPC) of the system improves better in 8-threaded systems than that of 4-threaded systems [12].

Vandierendonck et al. propose a technique based on speculative instruction window weighting (SIWW) to develop SMT fetch policies. To increase energy efficiency, SIWW is proposed. SIWW predicts the amount of work in the pipeline for fetch gating. This paper provides the way of applying SIWW to SMT systems to use for resource allocation [13].

Kucuk et al. propose a method similar to SIWW technique which is called history-based predictive instruction window weighting (HPIWW) [14]. This study claims that instead of speculating everything, a much simpler way of finding the threads' cumulative weights can be found. While the HPIWW circuitry has insignificant impact on power of processor, the circuitry of SIWW spends as much power as PRF. It is stated that performance of the HPIWW is greater than the SIWW by 3 % for all simulated workloads.

4. PROPOSED DESIGN

In this chapter, we discuss the details of our proposed designs and examine the algorithm we developed for controlling the distribution of WIB entries. First, we explain how our proposed dynamic algorithm works with a flowchart diagram and different figures that show the autonomous adjustment logic and sampling windows. Then we explore the algorithm of our proposed autonomous capping technique.

4.1. GENERAL DESCRIPTION

In this thesis, a dynamic algorithm to adjust the capping value of WIB entries that each thread allocate is proposed to improve both the instruction per cycle (IPC) value and fairness of the system. The algorithm we propose is based on the number of committed instructions for each thread on a specified window size. With this algorithm, we calculate a performance metric for each thread and decide if we increase or decrease the cap size of each thread separately in real-time.

First of all, we observe the number of committed instructions of the thread and calculate our efficiency metric then compare it with the previous window. In every sampling window, we decide if our calculated performance metric which is called CIPRE efficiency shown in Eq. 4.1 is higher or lower than the previous one.

If the current performance of the thread is higher than the one in the previous window, it means the thread we observe utilizes WIB entries efficiently. Then, we assume that this thread can use more WIB entries since it is observed that the thread has growing resource utilization and we increase the cap size for the specific thread in the next window. However, if the currently calculated performance value is less than the one that is calculated in the previous window, we decrease the cap value of that specific tread. If the current and the previously calculated performance metrics are equal, we make no adjustment for the cap value of the thread as shown in Figure 4.1.

$$CIPRE = \frac{committed\ instruction\ count}{number\ of\ WIB\ entries} \tag{4.1}$$

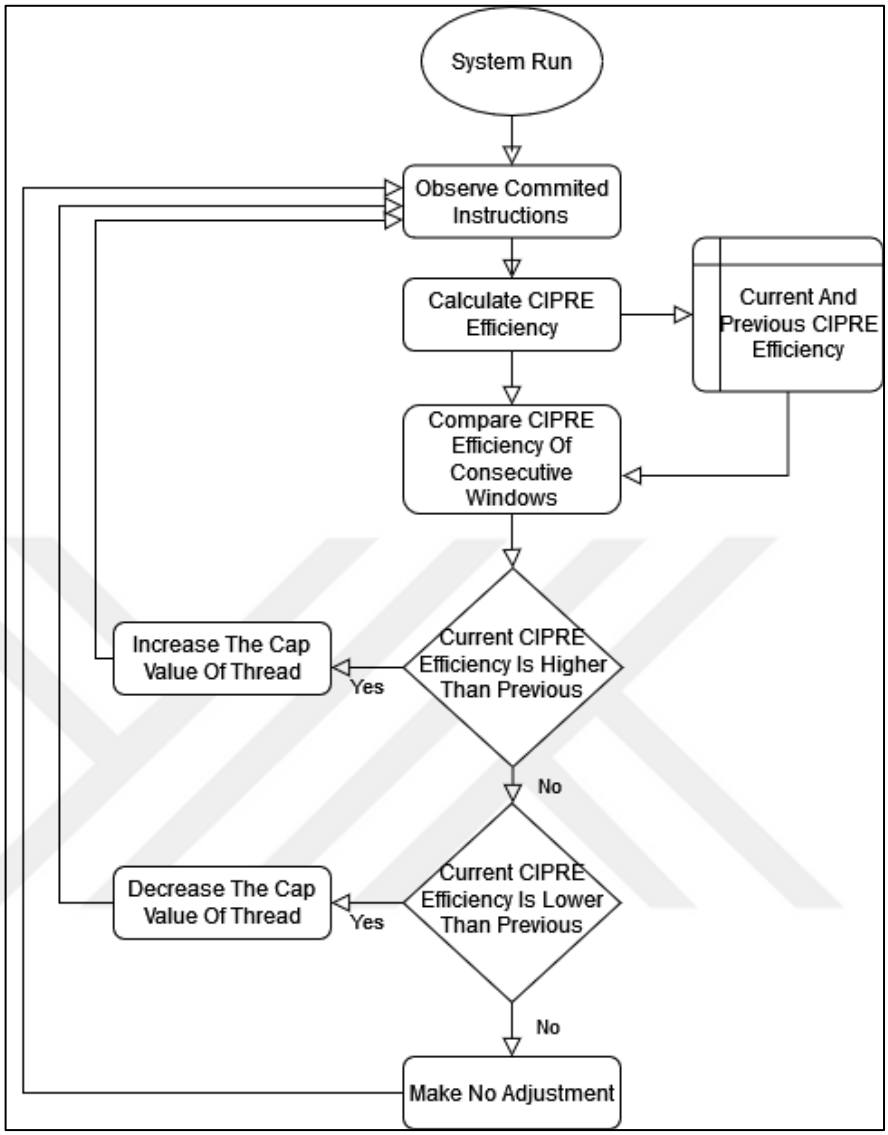

Figure 4.1. Flowchart of the dynamic capping system

This technique is used to develop an autonomous adjustment process for deciding the optimal cap value for each thread to achieve the highest performance output in real-time. A diagram of this autonomous adjustment system is presented in Figure 4.2.

If we look at the figure, from the beginning of the system run, we observe the system and send the collected system data (Sd) to calculate the current performance metrics. After calculation of the current performance (Cp), we send this data to our comparing logic processor. In this comparing process, we gather performance metric which is calculated in the previous window, and then we compare this previous performance (Pp) metric with the current performance.

With the result from the comparing mechanism, we send this adjustment result (Ar) to the adjustment process where we decide if we increase the cap value or decrease it with a pre-defined adjustment value. After this process cycle, cap values of all threads are calculated separately for the next window.

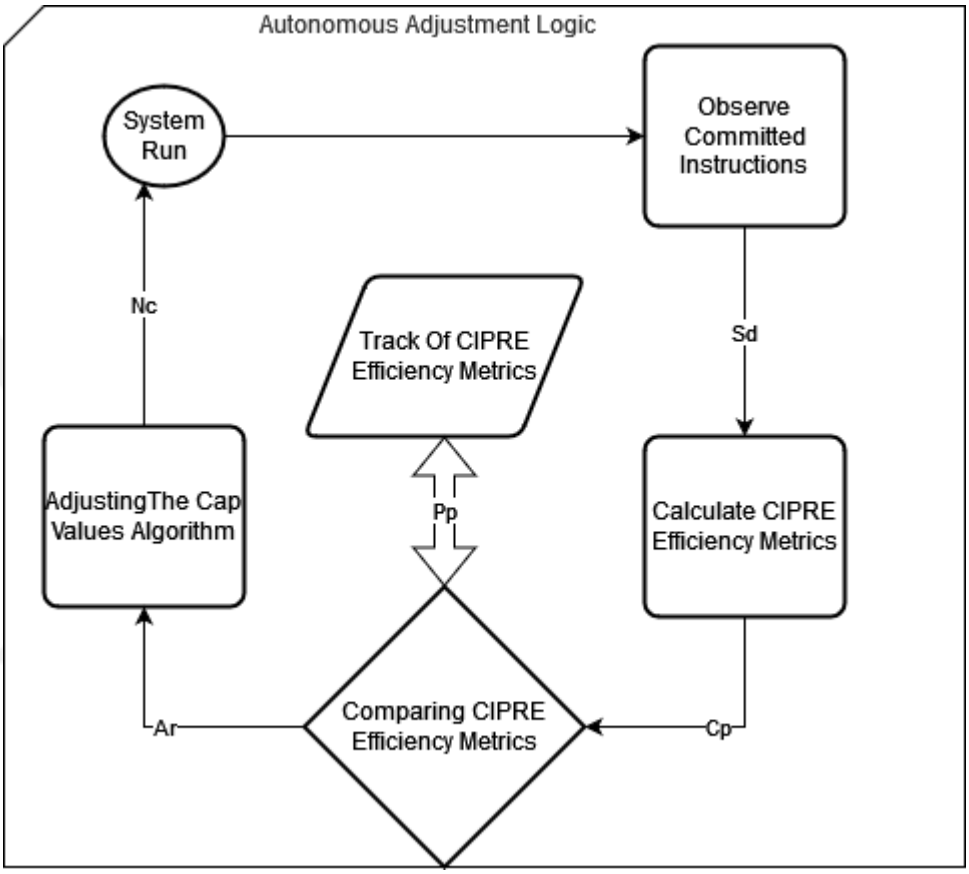

Figure 4.2. Autonomous adjustment logic

There are several important parameters we need to give high attention to for keeping this adjustment logic running efficiently without degrading the overall performance of the system. These parameters are the window size, the adjustment size, and the initial cap value.

The first important parameter is the window size, which is used for performance sampling as can be seen in Figure 4.3. It has a profound impact on the adaptivity and flexibility of the process. The adjustment will not be able to keep up with the changes in resource demands of the workloads if we set the window size too large. In contrast, if we set the window size too small, the result of the performance comparison of the two windows may be inconsistent and inaccurate. Because the performance we calculate for that short period of time may be highly dynamic and unstable.

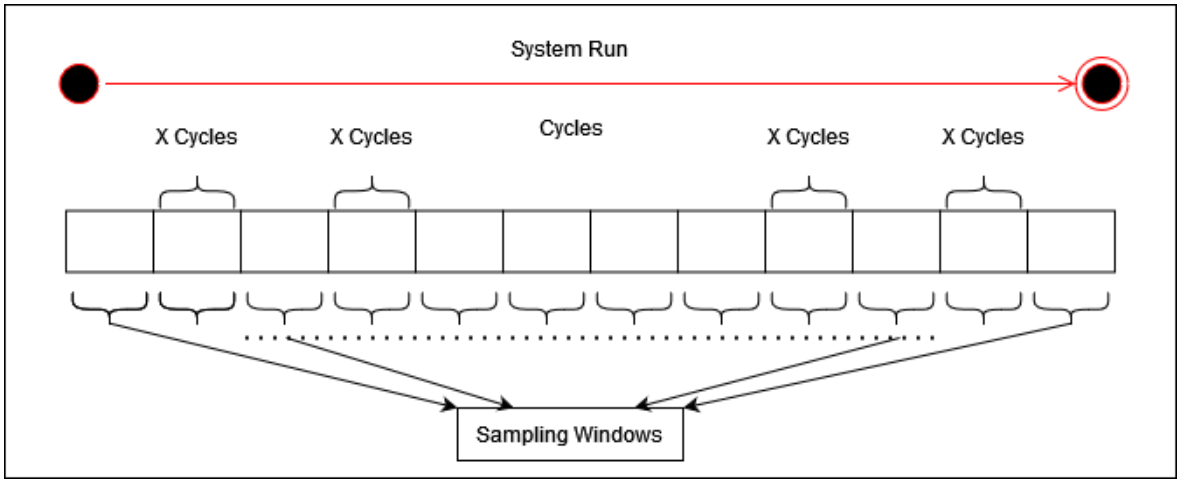

Figure 4.3. Sampling windows at run-time

We need to make sure that the performance we calculate, is not affected by the rapid and inconsistent behavior changes of threads. Therefore, the window size must be adjusted delicately to avoid unpredictable arbitrary changes in thread's performance.

The second parameter is the adjustment size which also has an impact on the performance of the system. The adjustment size can be defined as a value that we use to change the cap value of the threads in run-time. For example, if we have a performance gain from the previous window, we increase the cap value of the thread by the adjustment size. We use this adjustment value both to increase and decrease the cap values of threads.

When determining this adjustment value we need to consider the WIB size and the number of threads in the system. If the number of threads is higher, then we need to keep the adjustment value at a reasonable scale. Also when we have a small WIB size, we need to set a smaller value for the adjustment size. If we set the adjustment size too large, we may give a thread a higher number of WIB entries than it really needs. As a result of this, we would be taking the number of WIB entries that another thread may use since it has a higher utilization over other threads. Therefore, the adjustment size must be decided by considering various parameters mentioned before to achieve an efficiently working algorithm.

The last parameter is the initial cap value of threads. The initial cap value is defined before run-time as an initial capping for threads until the first sampling period completes. After the first window is completed a new capping value is calculated and updated for all threads. We decide the initial cap value according to our test based on different settings and workloads. It is set before run-time to make threads run efficiently without starvation and overuse of

WIB entries. This way we can achieve the best overall performance result since we start with the initial capping applied system instead of the non-capping system.

There are two different types of this algorithm we propose for the IPC improvement. The first algorithm we propose is an adaptive capping (AC) algorithm that controls the WIB capping value of threads according to the maximum number of WIB entries that exist. The second algorithm that we propose is also an adaptive capping algorithm like the first one but without limiting (ACNL) the total number of capping values given to the threads at run-time.

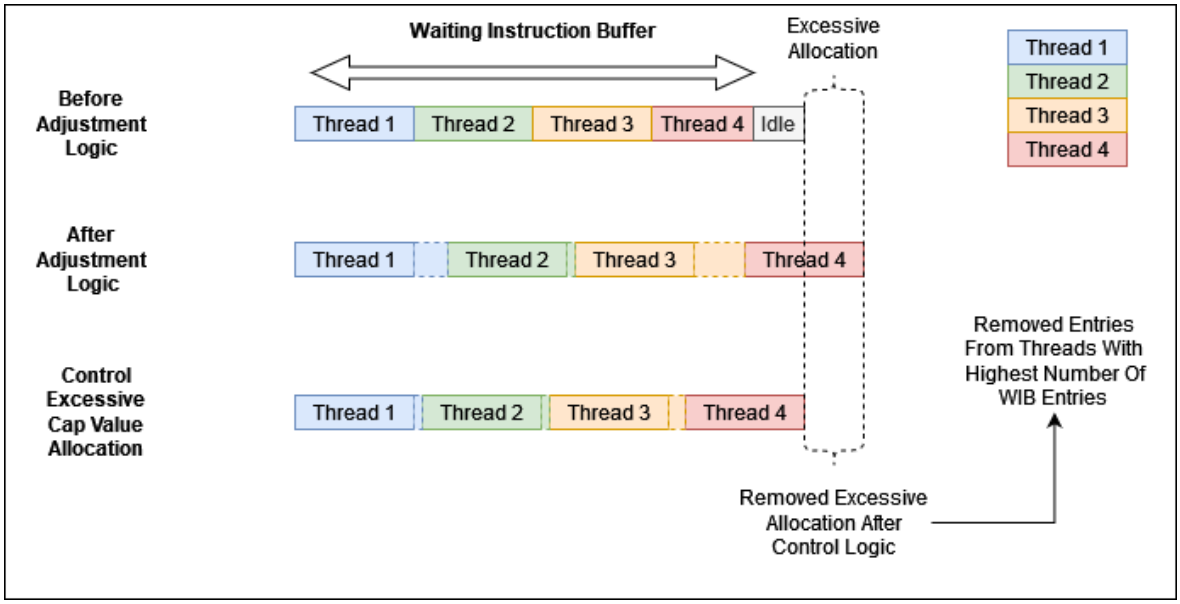

Figure 4.4. The adjustment logic

In the first algorithm, we control the total number of cap values that we give threads according to the total WIB size. In our algorithm, we calculate the current performance at the end of every window, and we compare it with the performance result of the previous window. Then according to this comparison, we adjust the cap values of threads positively or negatively. We make this change of cap values by the adjustment size for each thread separately.

After we made the required adjustments for each thread, we calculate the total cap values given to these threads. Then we compare the current total cap values of threads with the total size of WIB. If the total size of cap values is greater than the total size of the WIB, then we need to decrease the cap values of threads until we match the total cap values of threads with

the total size of the WIB. We make the decrease from the threads with the highest cap values by the adjustment size as can be seen in Figure 4.4.

The second algorithm that we propose is very similar to the first algorithm. However, there is no limit set for the total number of cap values of threads. In this method, if the total number of cap values exceeds the total size of the WIB, we do not adjust the cap values like non-capping systems. On the other hand, we keep checking the performance of the system at the end of every sampling window, and we compare the current performance with the previous one. If the current performance is lower than the current performance, we still make the reduction of the current cap value by the adjustment size.

Since we do not control if the total number of cap values exceeds or not, the system looks like a traditional non-capping system. However, we still make adjustments to the cap values of threads and control the allocation of the WIB entries. With this method, we make the threads, which have the highest utilization, race over the resources instead of limiting one of them or both from using the resources that they need.

4.2. ALGORITHM EXPLANATION

In this chapter, we will examine the pseudo-code of the proposed algorithm and explain the code step-by-step following the given code snippets. We decided to go through over the proposed algorithm in four steps as shown in Figure 4.5.

The first step is the initialization of data and collecting statistics about the system such as committed instruction numbers, cycle counts, and simulated instruction number. In the second step, we calculate and compare the performance metric and then make adjustments to the cap values of the threads according to the comparison result. In the final step, we control the total cap size of the threads and compare it with the total WIB size.

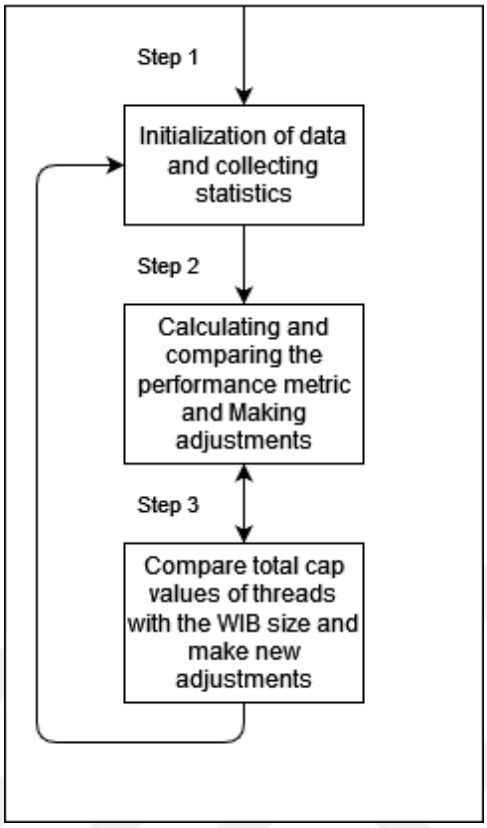

Figure 4.5. Steps of the proposed algorithm

As shown in Algorithm 4.1, the first step of our dynamic capping algorithm is consist of data initialization and collecting statistic. First of all, we define some important constants that we use through this algorithm. The initialCappingSize is what we use as a starting cap value for all threads. The incrementOfCapSize is the adjustment size of our dynamic capping algorithm, which is used for the adjustment logic after we calculate and compare our performance metric. Previously committed instructions are also stored to be able to calculate the performance of the system in earlier periods. In addition, we store the difference in committed instruction between windows.

Algorithm 4.1. Pseudocode of the proposed algorithm - 1

```
1: init totalCipreVal, initialCappingSize, incrementOfCapSize, prevCommitedInst[]
2: for i=0 to numOfThreads
3:     Threads[i].capSize = initialCappingSize
4: for i=0 to numOfThreads
5:     commitedInstChange[i] = Threads[i].simNumInst – prevCommitedInst[i]
```

The second part of the algorithm is where we calculate and compare the performance metric and make adjustments according to this comparison process as can be seen in Algorithm 4.2. Firstly, we define two different arrays for our calculated performance metric of both the current and the previous sampling windows. Moreover, we define some variables for finding the max cap value since we use this max value later for checking if the total cap value exceeds the total size of the WIB.

After we define the required variables and arrays, we calculate our performance metric, which is Committed Instructions Per Resource Entry (CIPRE) value in the presented algorithm. This CIPRE value is calculated as a division of difference in committed instruction count by the cap size of a thread. We also find the total CIPRE value of threads by calculating all CIPRE values of threads using for loop since we use it to calculate current CIPRE values of all threads. Then for each thread, we calculate the current CIPRE value by dividing the CIPRE value by the total CIPRE value we calculate earlier. We compare this calculated new CIPRE value with the previously calculated CIPRE value, and if the current value is greater than the previous CIPRE value, we increase the cap value of the thread by the increment of cap size which we mentioned as the adjustment size in the earlier chapters. When the current CIPRE value is greater than the previous one, it means that this thread is using its allocated resources in an efficient manner.

Next, we find the maximum cap value and sort threads by their current adjusted cap values. The reason why we sort these threads is that the maximum available WIB size may be exceeded as a result of this adjustment we made. Then, we need to make a new adjustment on cap values of thread to match the size of the WIB. At the end of this stage, we check if the previous CIPRE values are greater than the current CIPRE values. If they are greater then, we decrease the cap value of the thread by the adjustment size which is denoted as an increment of the cap size in the algorithm.

Algorithm 4.2. Pseudocode of the proposed algorithm - 2

```
1: init: maxCapVal, tempMaxCapVal, temp, temp1
2: init: currentCipreValues[], previousCipreValues[]
3: for i=0 to numofThreads
4:     cipreValues[i] = commitedInstChange [i] / Threads[i].capSize
5: for i=0 to numOfThreads
6:    totalCipreVal += cipreValues[i]
7: for i=0 to numOfThreads
8:     currentCipreValues[i] = cipreValues[i] / totalCipreVal
9:     if currentCipreValues[i] > previousCipreValues[i] then
10:        Threads[i].capSize = Threads[i].capSize + incrementOfCapSize
11:      flag[n] = k
12:      maxCapVal = Threads[i].capSize
13:      for m = 0 to n
14:         temp1 = flag[k]
15:           maxCapVal = Threads[temp1].capSize
16:         if maxCapVal > maxCapVal then
17:            temp = flag[n]
18:            temp1 = flag[m]
19:            flag[m] = temp
20:            flag[n] = temp1
21:            maxCapVal = Threads[flag[i]].capSize
22:    else if currentCipreValues[i] > previousCipreValues[i] then
23:              Threads[i].capSize = Threads[i].capSize – incrementOfCapSize
24:    end if
```

In the last stage of the algorithm, we modify the cap values that we make adjustment on them in the previous part of the algorithm. This step of the algorithm is the control logic which is only applied if the total number of cap values exceeds the total size of the WIB. As can be seen in Algorithm 4.3, we define variables such as the number of changes, total cap value, and difference in cap size. First, we calculate the total number of cap size of threads that we made adjustment on cap values. Then, we find the difference between the total number of cap values and the total size of the WIB. If there is a difference between them, then we find the number of threads we need to change to be able to match the WIB size. Then we start to reduce the cap values of threads that have the highest cap values. Next, we assign the current cap values to the previous cap values and also assign the number of simulated instructions to the previously committed instruction count array.

Algorithm 4.3. The pseudocode of the proposed algorithm - 3

```
1: init: numOfChange, count, totalCapVal, capSizeDifference
2: for i=0 to numThreads
3:      totalCapVal += Threads[i].capSize
4: capSizeDifference = totalCapVal - WIBSize
5: numOfChange = capSizeDifference / incrementSize
6: for j=0 to numOfThreads
7:     if  totalCapVal > WIBSize & numOfChange > count then
8:         for z=0 to numOfChange
9:             if flag[z] != -1 then
11:                Threads[flag[z]].capSize=Threads[flag[k]].capSize - incrementSize
12:            count ++
13:            end if
14:    end if
15: for m=0 to numThreads
16:     previousCipreValues [m] = currentCipreValues [m]
17: for n=0 to numThreads
18:     prevCommitedInst[n] = Threads[n].simNumInst
```

5. EXPERIMENTAL METHODOLOGY

In this chapter, we show which simulation environment we used for our thesis work. Also, we give details of our benchmark workloads that are used for both 4-thread and 8-thread system tests. Moreover, we explain the metrics that we used in our system tests.

5.1. SIMULATION ENVIRONMENT

For the proposed algorithm in our thesis, M-sim processor simulator is used as the simulation environment [15]. It is a multi-threaded microarchitectural simulation environment that has a cycle-accurate model for the pipeline structures. Performance estimation and power analysis are two of the many important features of this simulation environment. Also, it consists of cycle-accurate models including integer and floating-point register files, the waiting instruction buffer (WIB), the load-store queue, and the re-order buffer (ROB). Moreover, single-threaded execution and multi-threaded execution which is operated according to the SMT model is supported in the M-sim simulator. The WIB, register files, caches, and execution units are some of the shared resources among threads in the SMT mode of the simulator.

Table 5.1. Configuration of the simulated processor

Parameter	**Configuration**
Machine Width	4-wide fetch/dispatch/issue/commit
L/S Queue size	48 entry Load/Store queue
ROB	128 entry ROB
WIB size	16/32-entry WIB
Physical Registers	256 integer and 256 floating-point
L1 I-cache	64KB, 2-way set-associative 64-byte line
L1 D-cache	64KB, 4-way set-associative 64-byte line, write-back, 1-cycle access latency
L2 Cache unified	512KB, 16-way set-associative 64-byte line, writeback, 10-cycle access latency
BTB	512 entry, 4-way set-associative
Branch Predictor	Bimod: 2K entry
Memory	32-bit wide, 300 cycles access latency
Period Length	1 Million cycles

5.2. BENCHMARKS

For our simulation runs, we run 100 million cycles and fast-forward the first 10 million instructions of each benchmark. In our tests, we set the sampling window to 1 million cycles. We use the SPEC CPU2006 benchmark suite to evaluate our proposed design. For 8-thread workloads, we have 10 combinations of workloads mixtures that can be seen in Table 5.2. For 4-thread workloads, we create 20 combinations of workload mixtures that can be seen in Table 5.3.

Table 5.2. Workloads for the 8-thread system

Mixtures	**Benchmarks**
Mixture 1	libquantumNS, dealIINS, gromacsNS, namdNS, hmmerNS, sjengNS, gobmkNS, mcfNS
Mixture 2	libquantumNS, bzip2NS, gobmkNS, mcfNS, gromacsNS, dealIINS, lbmNS, cactusADMNS
Mixture 3	gobmkNS, dealIINS, lbmNS, bzip2NS, sjengNS, namdNS, cactusADMNS, hmmerNS
Mixture 4	gobmkNS, dealIINS, hmmerNS, milcNS, namdNS, mcfNS, lbmNS, libquantumNS
Mixture 5	gobmkNS, gromacsNS, cactusADMNS, bzip2NS, libquantumNS, sjengNS, lbmNS, mcfNS
Mixture 6	h264refNS, sphinx3NS, bwavesNS, zeusmpNS, libquantumNS, sjengNS, lbmNS, mcfNS
Mixture 7	leslie3dNS, perlbenchNS, sphinx3NS, omnetppNS, specrandNS, gccNS, h264refNS, bwavesNS
Mixture 8	libquantumNS, dealIINS, gromacsNS, namdNS, bwavesNS, sphinx3NS, leslie3dNS, specrandNS
Mixture 9	leslie3dNS, gccNS, specrandNS, omnetppNS, gobmkNS, dealIINS, hmmerNS, milcNS
Mixture 10	sjengNS, namdNS, cactusADMNS, hmmerNS, gobmkNS, dealIINS, hmmerNS, milcNS

Table 5.3. Workloads for the 4-thread system

Mixtures	**Benchmarks**
Mixture 1	libquantumNS, dealIINS, gromacsNS, namdNS
Mixture 2	hmmerNS, sjengNS, gobmkNS, mcfNS
Mixture 3	libquantumNS, dealIINS, gobmkNS, mcfNS
Mixture 4	gromacsNS, dealIINS, lbmNS, cactusADMNS
Mixture 5	hmmerNS, sjengNS, lbmNS, bzip2NS
Mixture 6	libquantumNS, sjengNS, lbmNS, mcfNS
Mixture 7	namdNS, dealIINS, hmmerNS, milcNS
Mixture 8	namdNS, mcfNS, lbmNS, milcNS
Mixture 9	gobmkNS, gromacsNS, cactusADMNS, bzip2NS
Mixture 10	sjengNS, namdNS, cactusADMNS, hmmerNS
Mixture 11	h264refNS, sphinx3NS, bwavesNS, zeusmpNS
Mixture 12	bwavesNS, specrandNS, gccNS, leslie3dNS
Mixture 13	leslie3dNS, perlbenchNS, sphinx3NS, omnetppNS
Mixture 14	specrandNS, gccNS, h264refNS, bwavesNS
Mixture 15	h264refNS, sphinx3NS, gccNS, perlbenchNS
Mixture 16	bwavesNS, sphinx3NS, leslie3dNS, specrandNS
Mixture 17	leslie3dNS, gccNS, specrandNS, omnetppNS
Mixture 18	sphinx3NS, h264refNS, zeusmpNS, leslie3dNS
Mixture 19	specrandNS, omnetppNS, bwavesNS, sphinx3NS
Mixture 20	perlbenchNS, leslie3dNS, gccNS, h264refNS

5.3. METRICS

In this thesis, there are two metrics that we use for measuring the performance improvement and evaluating the fairness.

The first metric is what we use for the performance improvement by calculating the percentage of the difference between the baseline system's performance and the performance of the proposed system. The *AC* is the proposed adaptive capping method and the *Baseline* is the default SMT system. To calculate this metric, we take the difference between the IPC value of the *AC* and the IPC value of the *Baseline* then divide it by the IPC value of the *Baseline* and then we multiply it by 100 to get the improvement percentage. We do this calculation for each thread which is denoted by *n* the number of threads, then the sum of all threads calculation gives us the overall system improvement percentage. We formulate this metric as the Throughput Improvement in Eq. 5.1.

IPC_k^{AC} : The IPC value of the AC (The adaptive capping algorithm)

$IPC_k^{Baseline}$: The IPC value of the Baseline

$$Throughput\ Improvement = \sum_{k=0}^{n} \frac{IPC_k^{AC} - IPC_k^{Baseline}}{IPC_k^{Baseline}} \times 100 \qquad (5.2)$$

The other metric is the fairness of the algorithm shown in Eq. 5.2. *T* in the equation represents the number of threads. First, we calculate the division of the IPC value of the *baseline* algorithm to the IPC value of our *AC* for each thread. Then, we divide the number of threads by the sum of this calculation we made for each thread to calculate the fairness value.

IPC_k^{AC} : The IPC value of the AC (The adaptive capping algorithm)

$IPC_k^{Baseline}$: The IPC value of the Baseline

$$Fairness = \frac{T}{\sum_{k=0}^{n} \frac{IPC_k^{Baseline}}{IPC_k^{AC}}} \qquad (5.2)$$

6. TESTS AND RESULTS

In this chapter, we will review the test results of our proposed technique based on the simulation environment and the workload mixtures that we listed in the previous chapter. We test our proposed design against both the baseline setting and Lin's study that we explain in the background chapter. First, we review the test result of Lin's study for different settings. Then, we examine the test results of our adaptive capping system with the same setting against the baseline system and Lin's study.

6.1. LIN'S STUDY

As we described in chapter 2.1, we implemented the code of Lin's study and made tests based on this implementation. In this chapter, we present the test results of Lin's study against the default baseline system in terms of both throughput and fairness.

6.1.1. Throughput Results

In figure 6.1, we can see the throughput result which is the total IPC values of each one of the workload mixtures for Lin's study on a 4-thread system with an WIB size of 32. As can be seen in the figure, Lin's study has greater results than the baseline except workload 3, workload 5, and workload 20. Since Lin's study has short sampling windows it leads the inconsistent and unstable adjustments due to arbitrary changes in the behaviors of threads in short periods of execution. Therefore, it reduces the performance of the system in some workload settings. On the average, the baseline has slightly better throughput than the Lin's study.

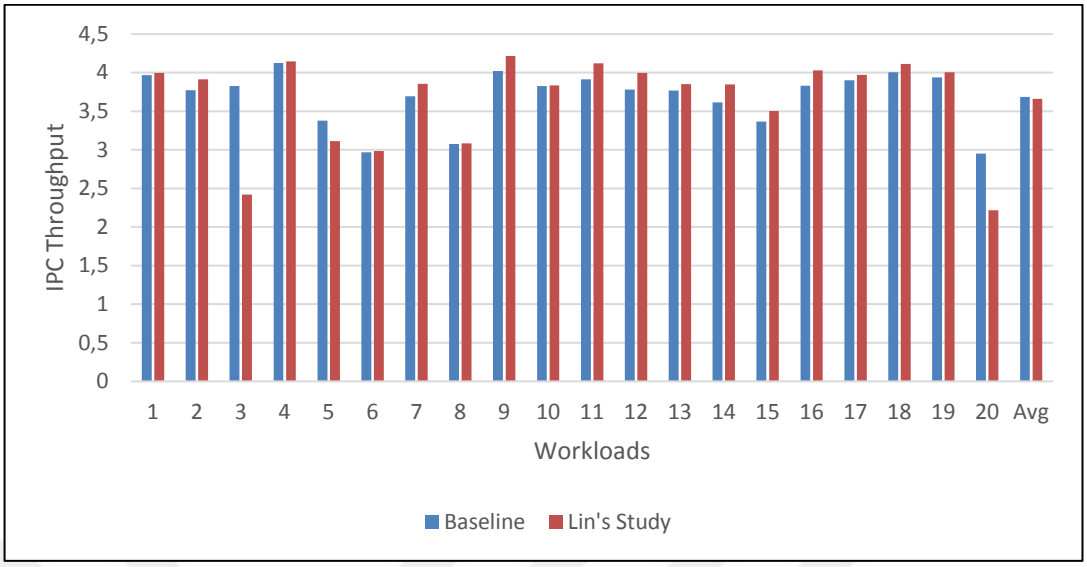

Figure 6.1. Throughput for the Baseline and Lin's study on a 4-thread SMT with a 32-entry WIB

Figure 6.2 shows the IPC throughput results of workloads on a 8-thread system with a 32-entry WIB. Except two workloads, Lin's study has a better performance on all workloads. Also, Lin's study has slightly better throughput than the baseline on the average.

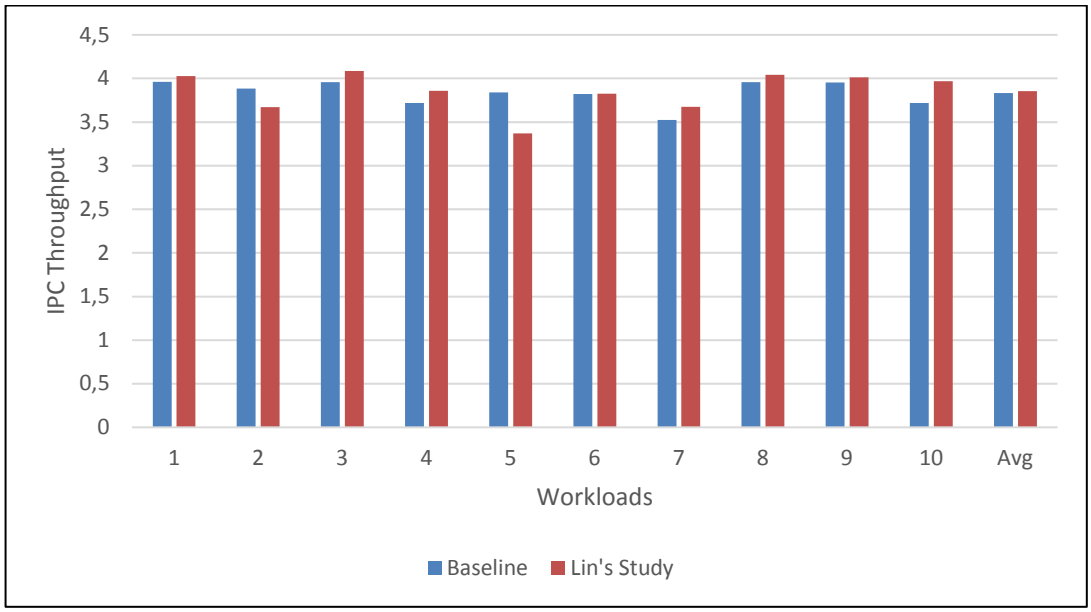

Figure 6.2. Throughput for the Baseline and Lin's study on 8-thread SMT with a 32-entry WIB

In figure 6.3, we can see the IPC throughput results of each workload on a 4-thread and WIB size of 16 system settings. While the baseline has a greater results on both workload 5 and workload 8, Lin's study has a better results on workload 1, workload 12, and workload 16. Since Lin's study has global capping for all threads it leads to starvation and over-use of resources and this reduces overall system performance in some workloads.

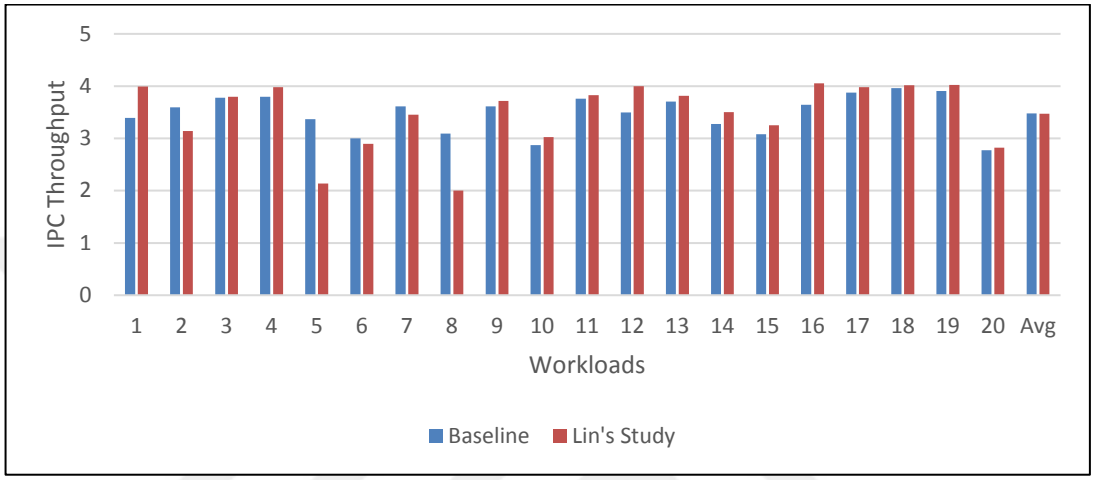

Figure 6.3. Throughput for the Baseline and Lin's study on a 4-thread SMT with a 16-entry WIB

We see the test results of IPC throughput of all workloads on a 8-thread SMT with a 16-entry WIB in Figure 6.4. Lin's study has a greater IPC throughput on all workloads except the workload 10. Also, Lin's study has average IPC of 3.71 while the baseline's average IPC is 3.48.

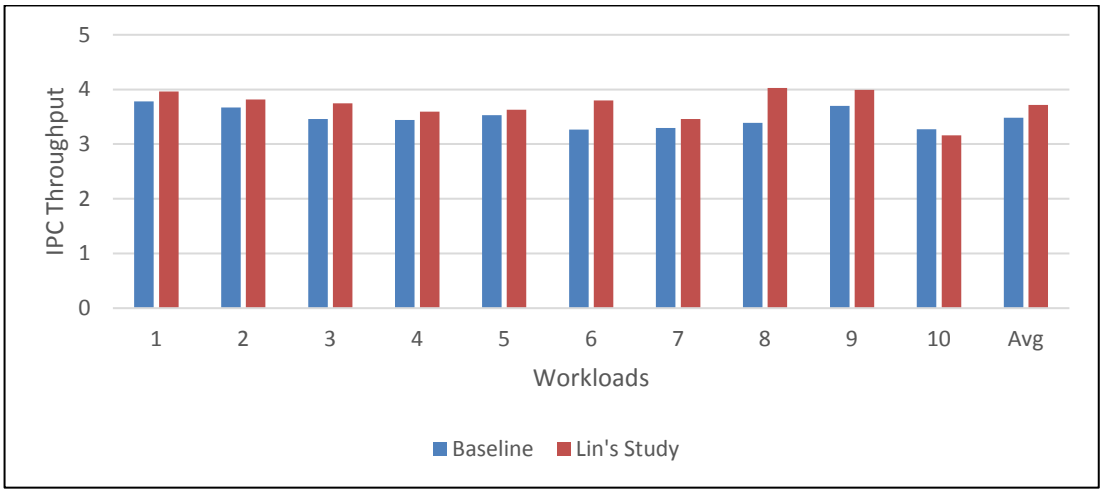

Figure 6.4. Throughput for the Baseline and Lin's study on a 8-thread SMT with a 16-entry WIB

6.1.2. Fairness Results

In figure 6.5, we can see the fairness test result which is calculated according to the fairness metric that we state in chapter 5.3 with the settings of a 4-Thread SMT with an WIB size of 32.

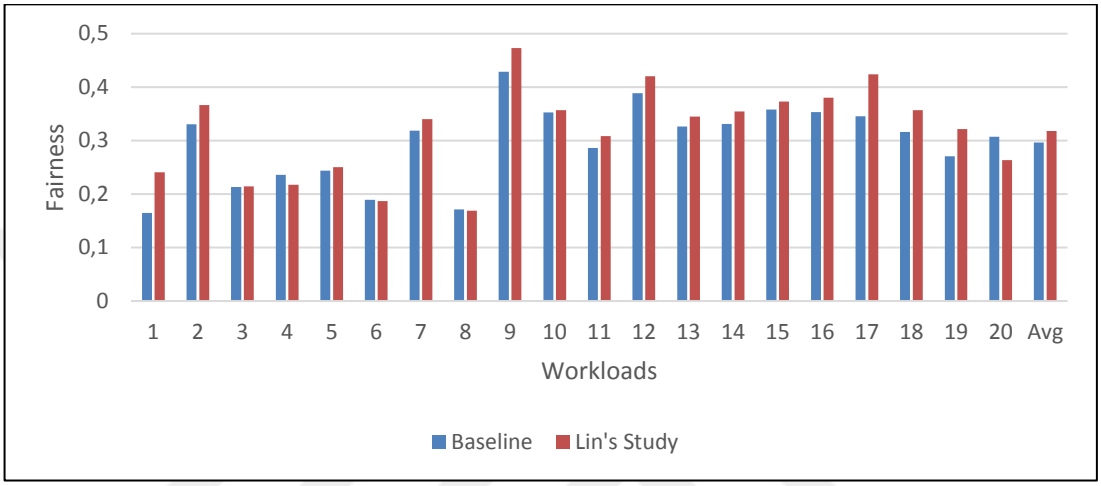

Figure 6.5. Fairness for the Baseline and Lin's study on a 4-thread SMT with a 32-entry WIB

Figure 6.6 shows the fairness results of workloads on a 8-thread SMT with a 32-entry WIB. Lin's study has a greater fairness result than the baseline for all workloads except workload 4 and workload 9. On the average, Lin's study has a fairness value of 0.13 while the baseline has fairness value of 0.10.

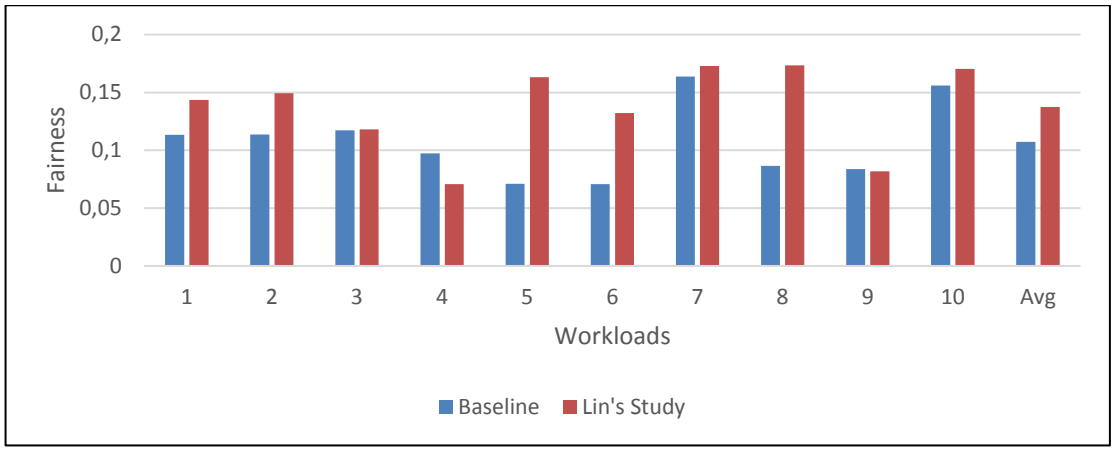

Figure 6.6. Fairness for the Baseline and Lin's study on a 8-thread SMT with a 32-entry WIB

In figure 6.7, we can see the fairness results of each workload on a 4-thread SMT with a 16-entry WIB.

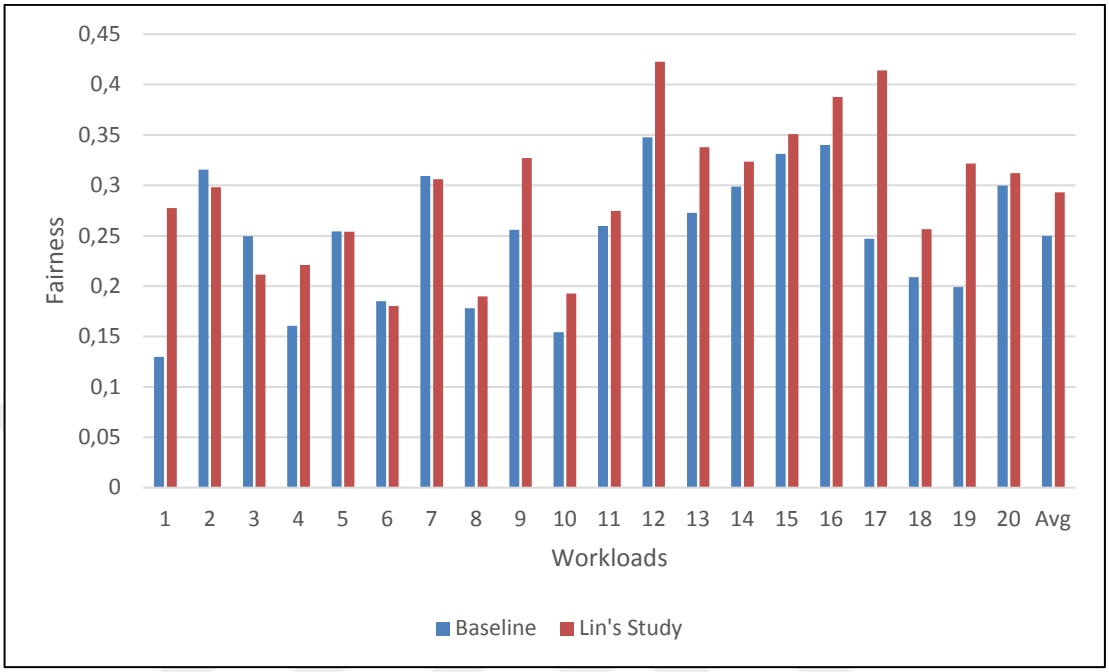

Figure 6.7. Fairness for the Baseline and Lin's study on a 4-thread SMT with a 16-entry WIB

We see the test results of fairness of all workloads on a 8-thread SMT with a 16-entry WIB in Figure 6.8.

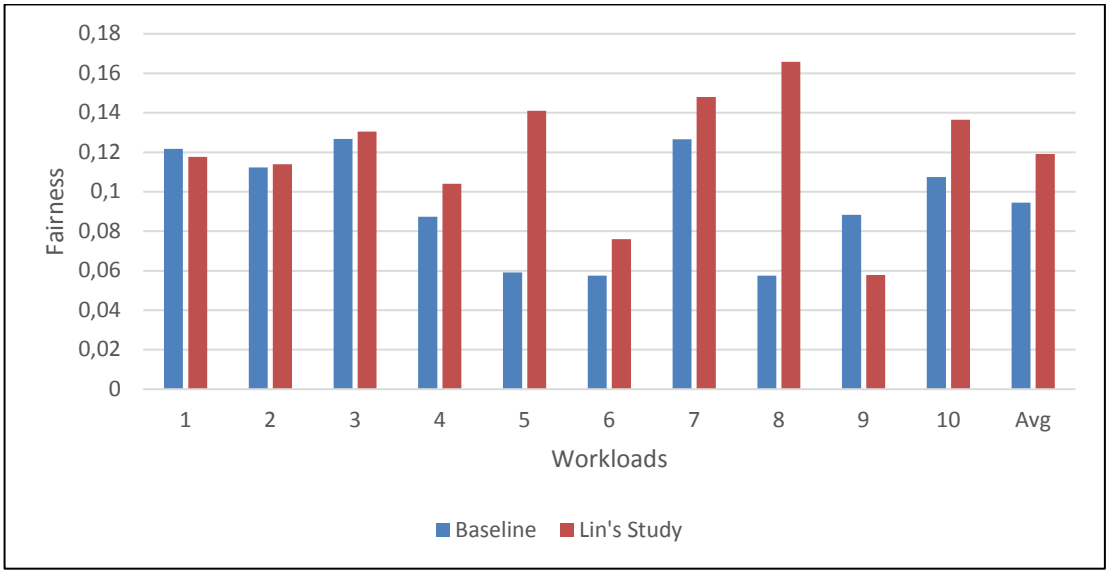

Figure 6.8. Fairness for the Baseline and Lin's study on a 8-thread SMT with a 16-entry WIB

6.1.3. Overall Improvement

In figure 6.9, we can see the improvement of the overall system that we calculate based on the throughput metric we described in chapter 5.3. Test results were calculated with settings of a 4-Thread SMT with a 32-entry WIB. Lin's study improve the baseline study on all workloads except three of them.

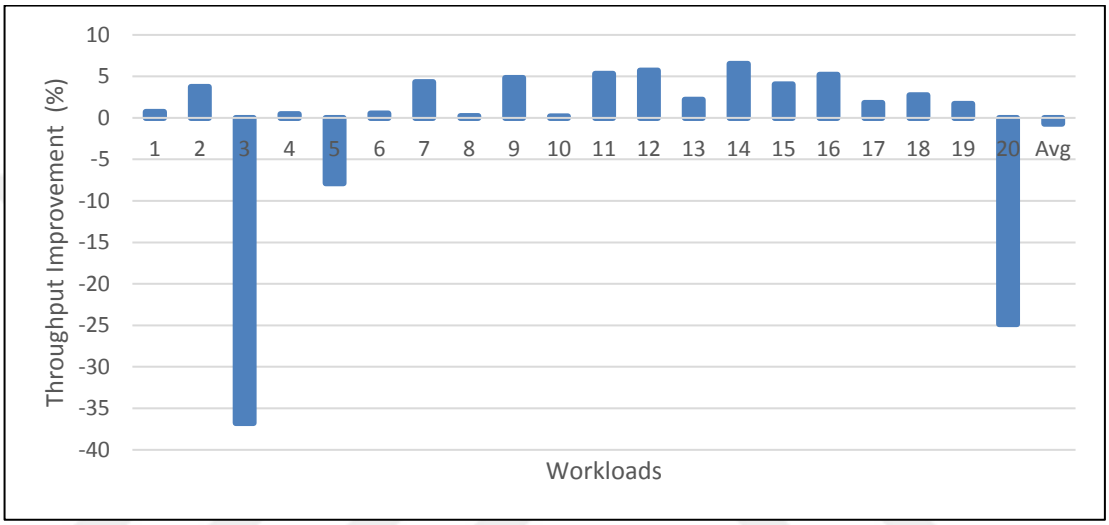

Figure 6.9. Improvement for the Baseline and Lin's study on a 4-thread SMT with a 32-entry WIB

Figure 6.10 shows the improvement percentage of the system results of workloads on the 8-thread SMT with a 32-entry WIB.

Figure 6.10. Improvement for the Baseline and Lin's study on a 8-thread SMT with a 32-entry WIB

In figure 6.11, we can see the improvement percentage of the system results of each workload on a 4-thread SMT with a 16-entry WIB. For workloads 5 and 8, Lin's study cause significant performance degradation while improving the throughput on most of the other workloads.

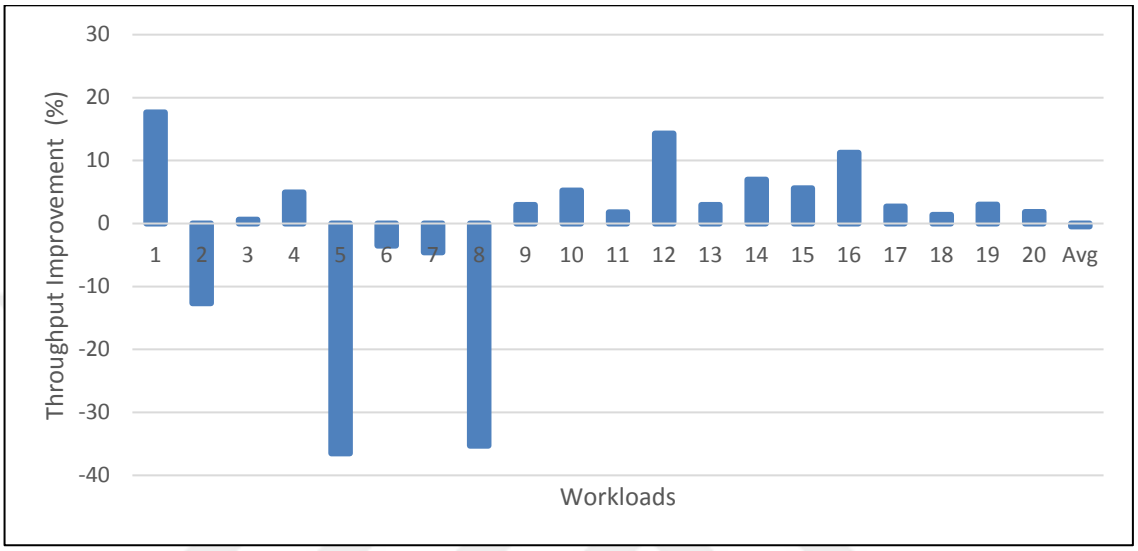

Figure 6.11. Improvement for the Baseline and Lin's study on a 4-thread SMT with a 16-entry WIB

We see the test results of the improvement percentage of the system for all workloads on the 8-thread SMT with a 16-entry WIB in Figure 6.12. For this setting, Lin's study improve the throughput of the system on all workloads except the last workload. It has an average throughput improvement of 6.9 %.

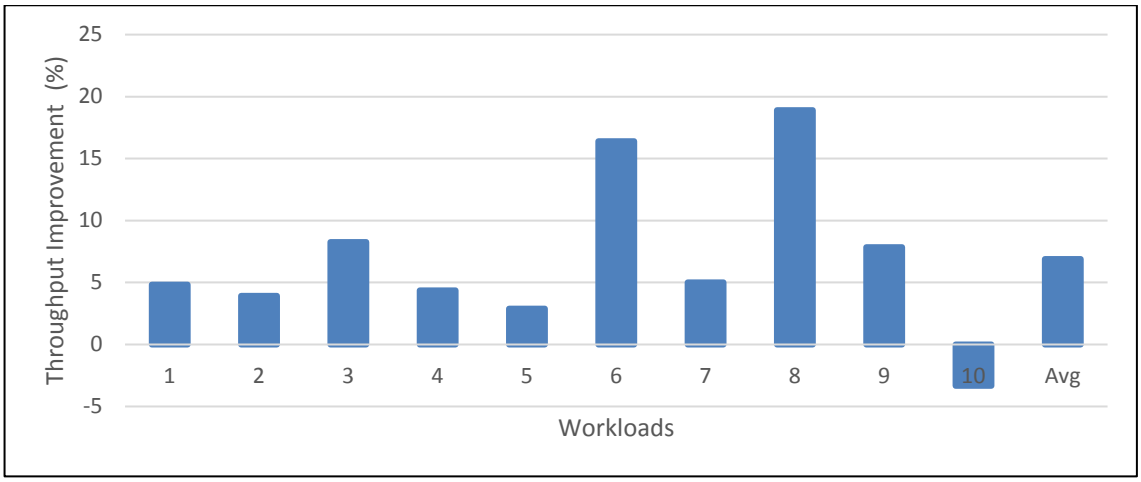

Figure 6.12. Improvement for the Baseline and Lin's study on a 8-thread SMT with a 16-entry WIB

6.2. PROPOSED STUDY

In this chapter, we present the test results of our proposed study compared to Lin's study and baseline results in terms of throughput and fairness.

6.2.1. Throughput Results

In figure 6.13, we can see the throughput results which is the total IPC values of each one of the workload mixtures for our proposed study on a 4-thread with 32-entry WIB system settings.

In Figure 6.14, the average values of all workloads for the baseline, Lin's study, and our proposed study are represented as a bar chart. From this average of workloads graphic, we can say that our proposed algorithm shows greater performance than Lin's study. Moreover, despite Lin's study having a greater performance on some workloads, it has worse performance results than the baseline performance, on the average.

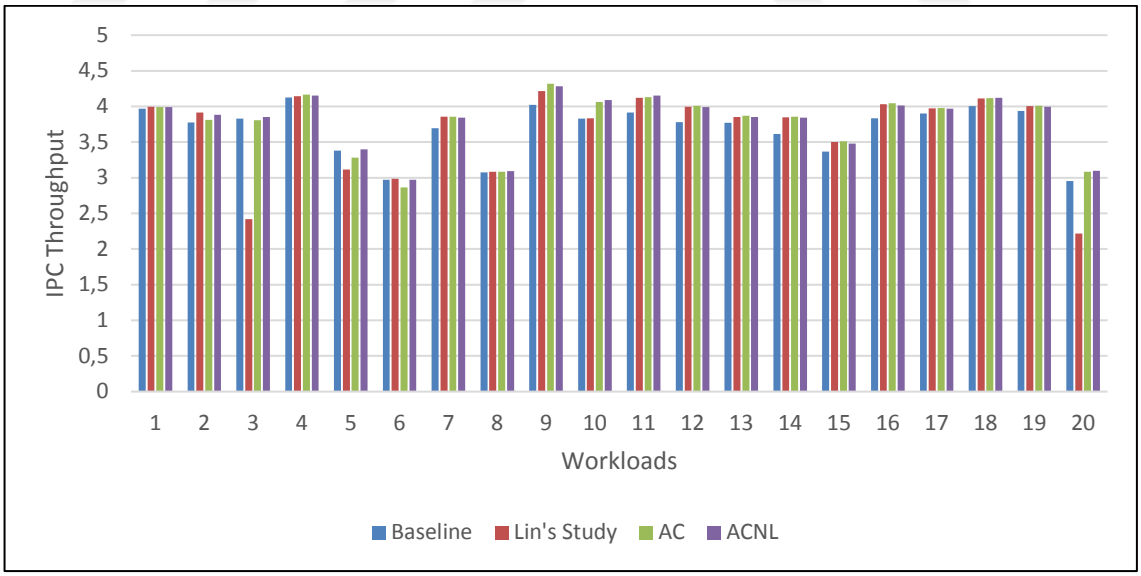

Figure 6.13. Throughput for all workloads on a 4-thread SMT with a 32-entry WIB

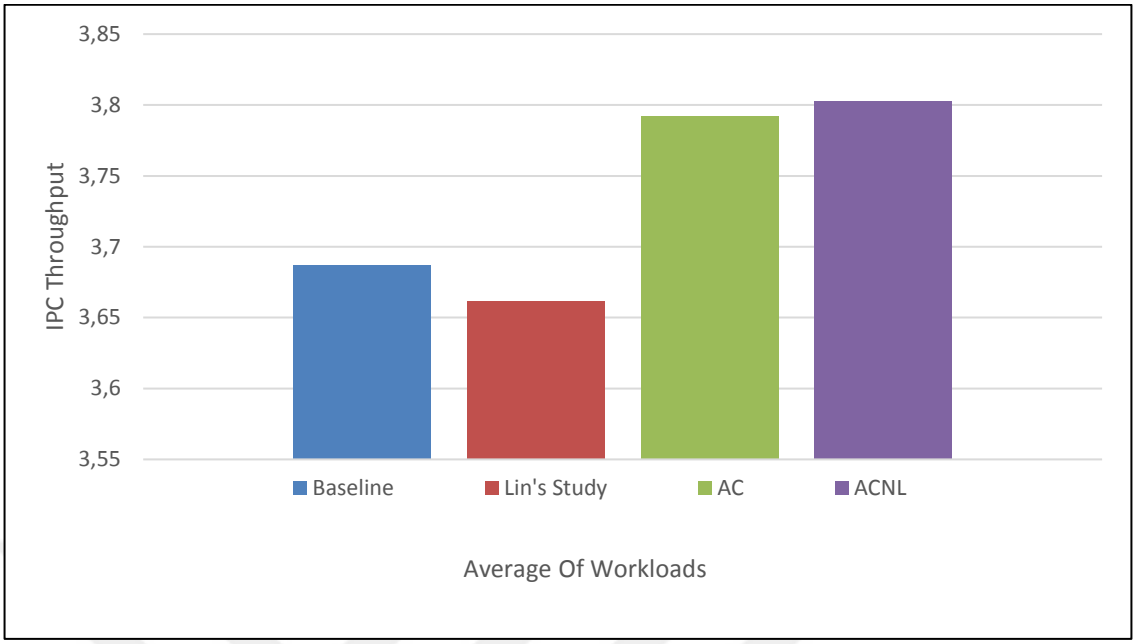

Figure 6.14. Average throughput for all workloads on a 4-thread SMT with a 32-entry WIB

Figure 6.15 shows the IPC throughput results of workloads on a 8-thread SMT with a 32-entry WIB settings. It can be seen that our proposed adaptive capping algorithm is still better than Lin's study in Figure 6.16. While Lin's study has an average IPC value of 3.85, our proposed algorithm AC has an average IPC value of 3.94 and ACNL has an average IPC value of 3.93.

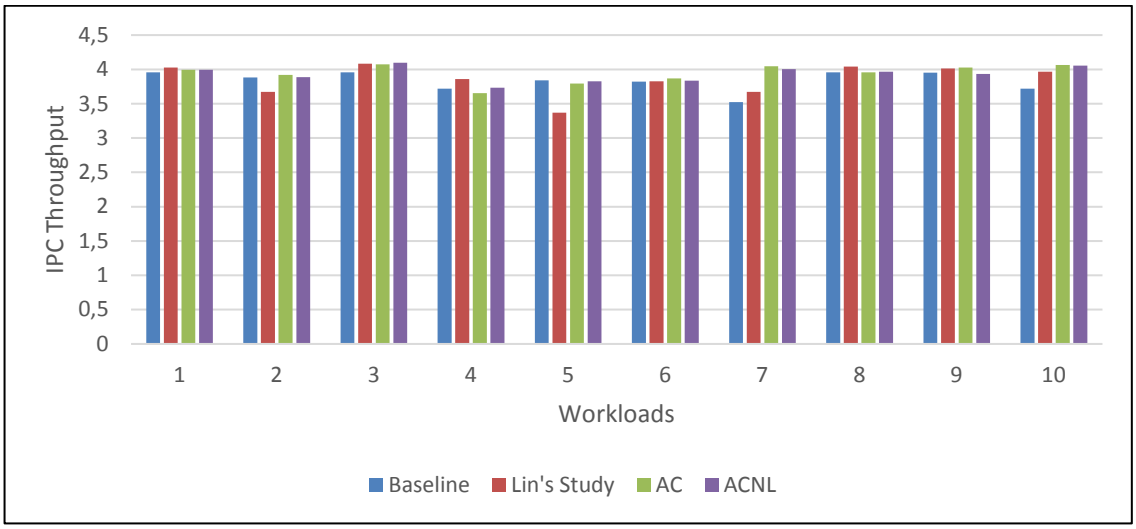

Figure 6.15. Throughput for all workloads on a 8-thread SMT with a 32-entry WIB

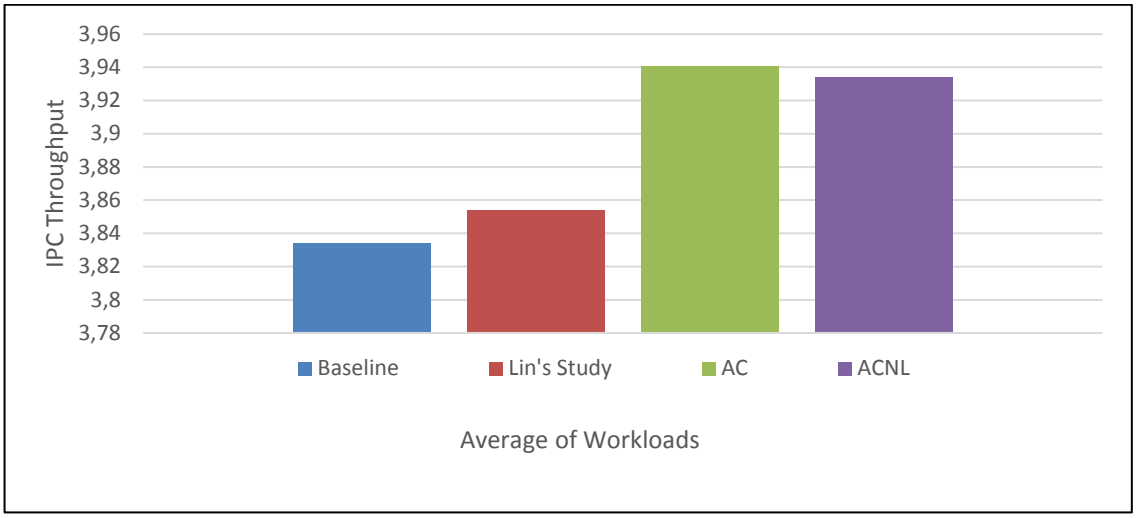

Figure 6.16. Average throughput for all workloads on a 8-thread SMT with a 32-entry WIB

In figure 6.17, we can see the IPC throughput results of each workload on a 4-thread SMT with a 16-entry WIB settings. Since Lin's study has short sampling windows it leads the inconsistent and unstable adjustments due to arbitrary changes in the behaviors of threads in short periods of execution. Therefore, it reduces the performance of the system in some workloads. As shown in Figure 6.18, our proposed algorithm AC has an average IPC value of 3.73 and ACNL has an average IPC value of 3.57 which is still higher IPC output than Lin's study which has an average IPC value of 3.47.

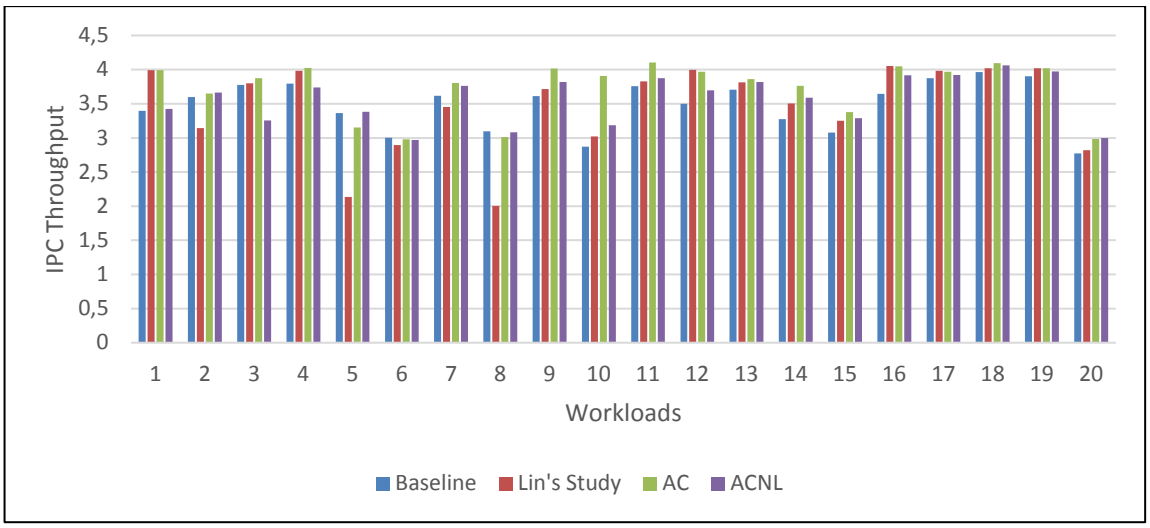

Figure 6.17. Throughput for all workloads on a 4-thread SMT with a 16-entry WIB

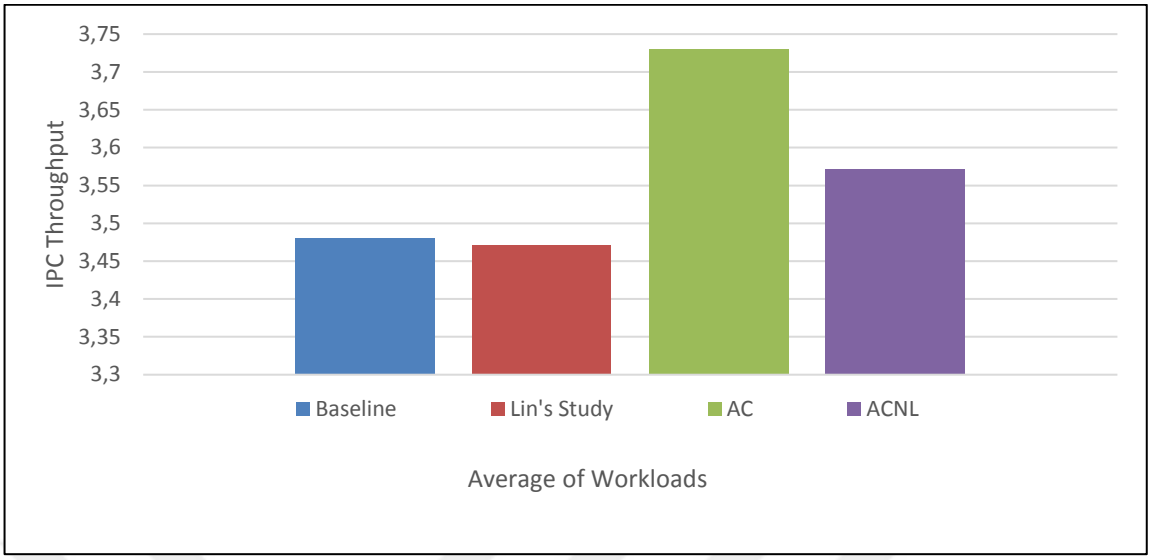

Figure 6.18. Average throughput for all workloads on a 4-thread and SMT with a 16-entry WIB

We see the test results of the IPC throughput of all workloads on a 8-thread SMT with a 16-entry WIB system settings in Figure 6.19.

In Figure 6.20, we see Lin's study has a better IPC value than the results of other system settings. However, the proposed algorithm still has higher performance for both the adaptive capping algorithm (AC) and the adaptive capping algorithm with no limit (ACNL).

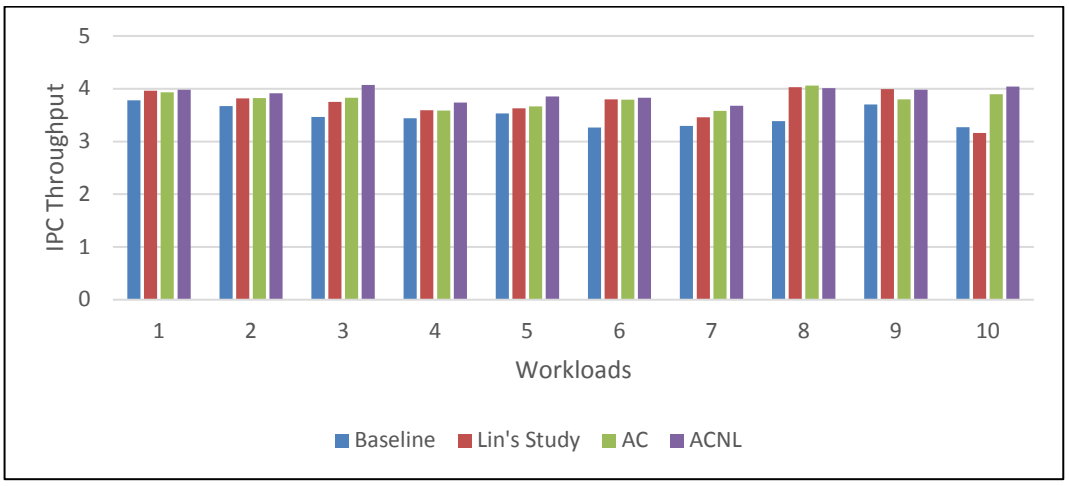

Figure 6.19. Throughput for all workloads on a 8-thread SMT with a 16-entry WIB

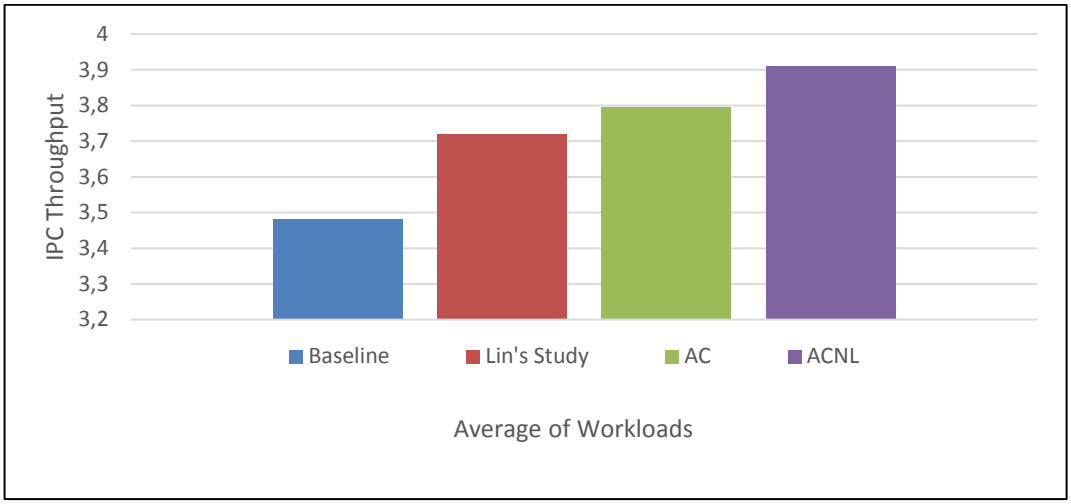

Figure 6.20. Average throughput for all workloads on a 8-thread SMT with a 16-entry WIB

6.2.2. Fairness Results

In figure 6.21, we can see the fairness test results which is calculated according to the fairness metric that we state in chapter 5.3 with settings of 4-Thread SMT with a 32-entry WIB.

As can be seen in Figure 6.22, despite Lin's study having slightly better fairness than the ACNL algorithm, our proposed algorithm AC has the highest average fairness of 0.33 among three of them.

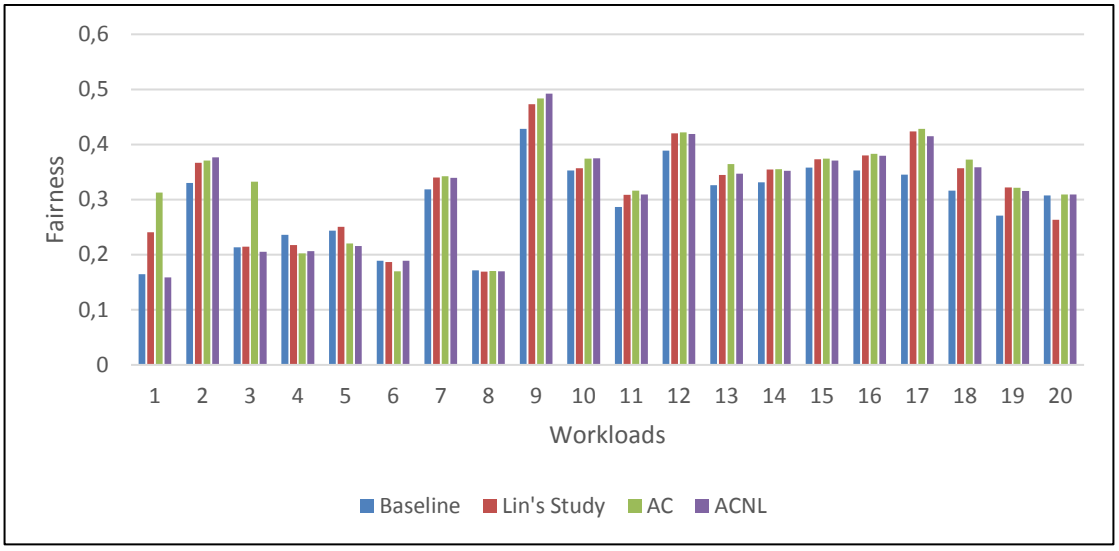

Figure 6.21. Fairness for all workloads on a 4-thread SMT with a 32-entry WIB

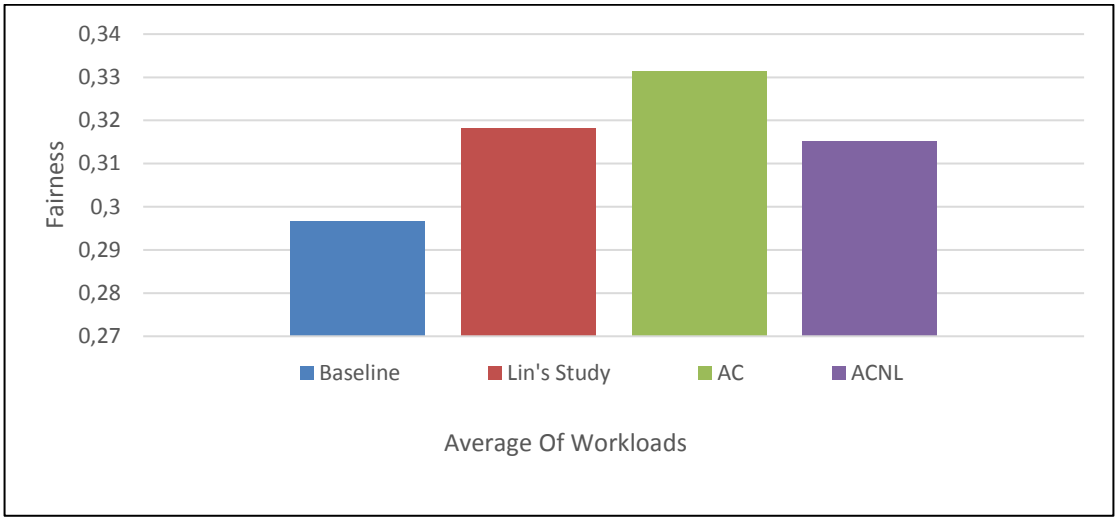

Figure 6.22. Average fairness for all workloads on a 4-thread SMT with a 32-entry WIB

Figure 6.23 shows the fairness results of workloads on the 8-thread SMT with a 32-entry WIB system settings. Since Lin's study has global capping for all threads it leads to starvation and over-use of resources and this reduces overall system performance in some workloads and fairness among threads. As can be seen in Figure 6.24, despite Lin's study having slightly better fairness than the ACNL algorithm, our proposed algorithm AC has the highest average fairness of 0.15 among three of them.

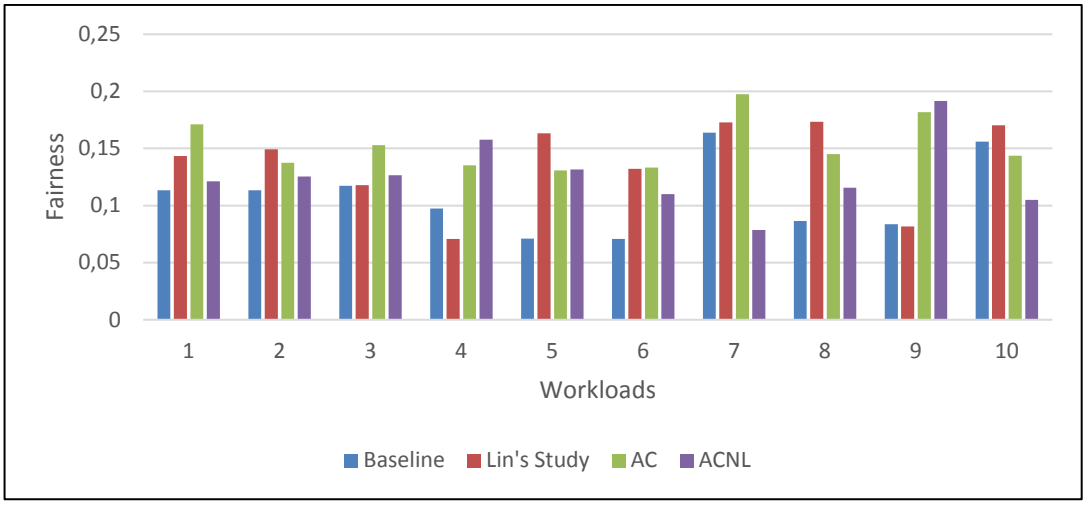

Figure 6.23. Fairness for all workloads on a 8-thread SMT with a 32-entry WIB

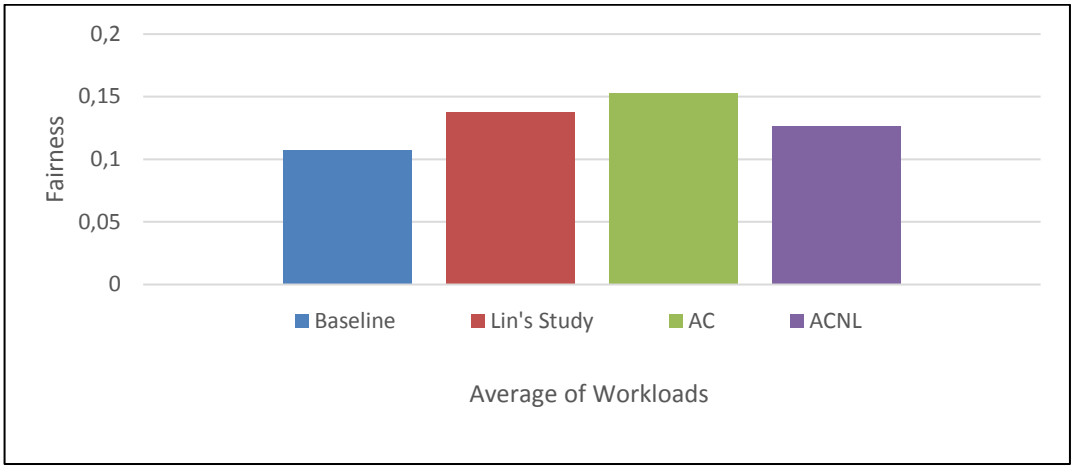

Figure 6.24. Average fairness for all workloads on a 8-thread SMT with a 32-entry WIB

In figure 6.25, we can see the fairness results of each workload on a 4-thread SMT with a 16-entry WIB system settings. Since Lin's study has short sampling windows it leads the inconsistent and unstable adjustments due to arbitrary changes in the behaviors of threads in short periods of execution. Therefore, it reduces the performance of the system in some workloads. As can be seen in Figure 6.26, despite Lin's study having slightly better fairness than the ACNL algorithm, our proposed algorithm AC has the highest average fairness of 0.33 among three of them.

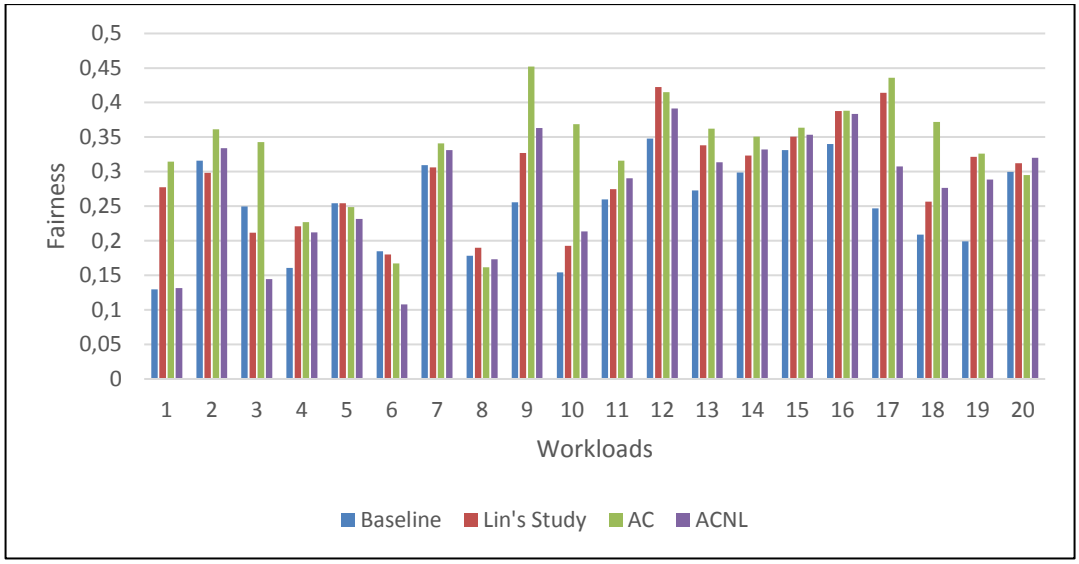

Figure 6.25. Fairness for all workloads on a 4-thread SMT with a 16-entry WIB

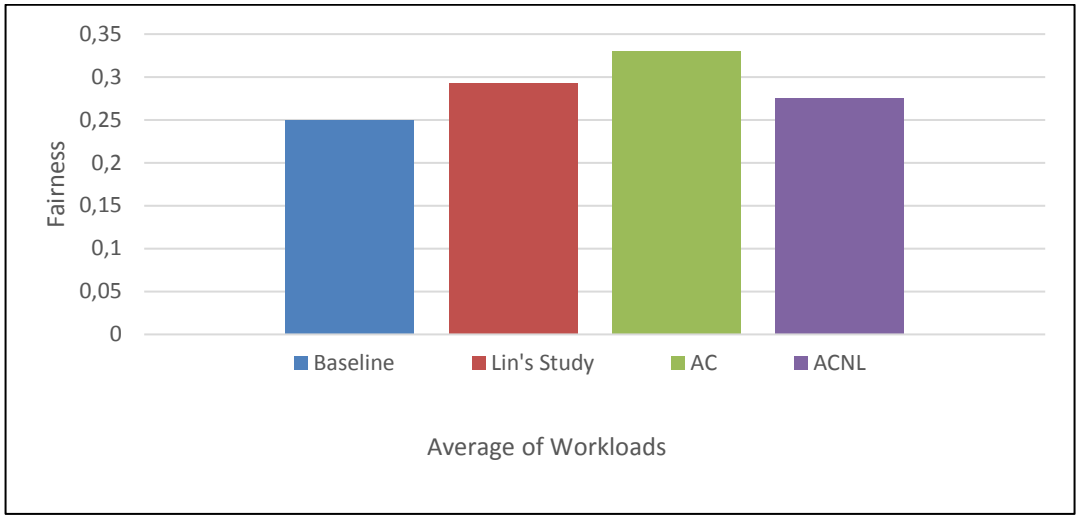

Figure 6.26. Average fairness for all workloads on a 4-thread SMT with a 16-entry WIB

We see the test results of fairness of all workloads on a 8-thread SMT with a 16-entry WIB system settings in Figure 6.27. Since Lin's study has short sampling windows it leads the inconsistent and unstable adjustments due to arbitrary changes in the behaviors of threads in short periods of execution. Therefore, it reduces the fairness among threads in some workloads. In figure 6.28, we can see that both proposed algorithms AC and ACNL has better fairness than the fairness result of Lin's study.

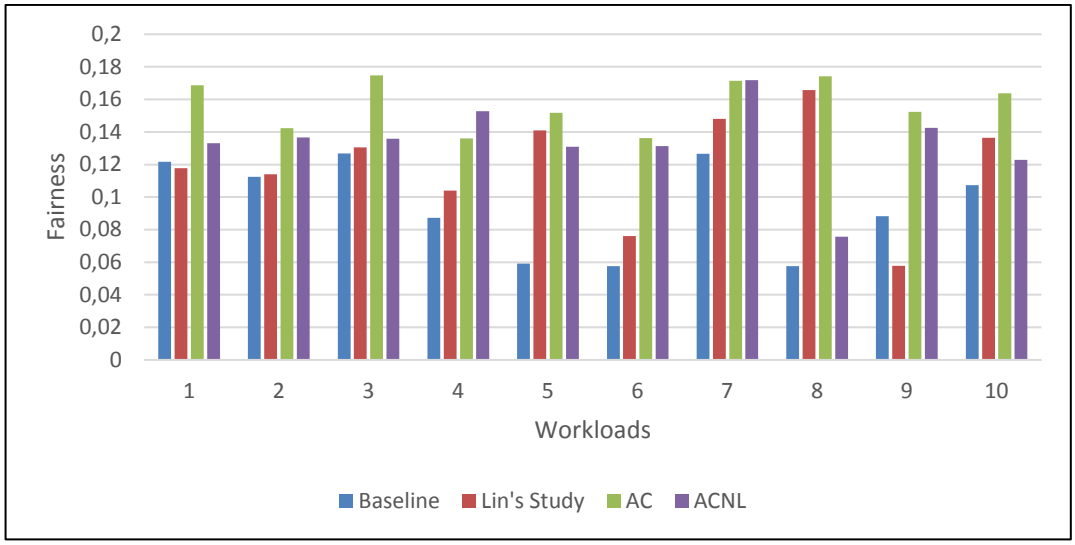

Figure 6.27. Fairness for all workloads on a 8-thread SMT with a 16-entry WIB

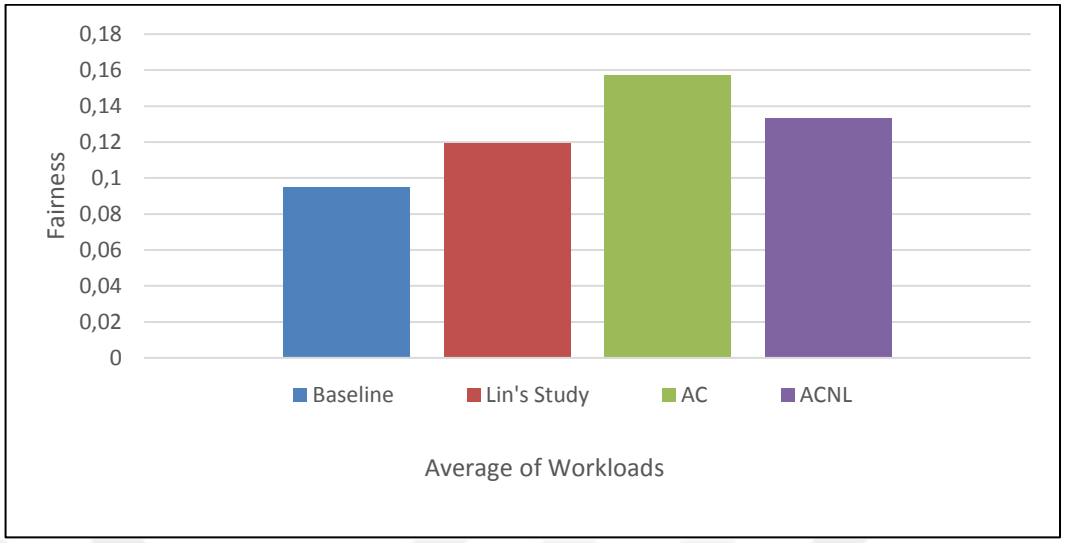

Figure 6.28. Average fairness for all workloads on a 8-thread SMT with a 16-entry WIB

6.2.3. Overall Improvement

In figure 6.29, we can see the improvement of the overall system that we calculate based on the throughput metric we described in chapter 5.3. Test results were calculated with settings of a 4-Thread SMT with a 32-entry WIB. As we can see in Figure 6.30, the Lin's Study has a performance degradation on the average while our proposed algorithms have an improvement over the baseline system.

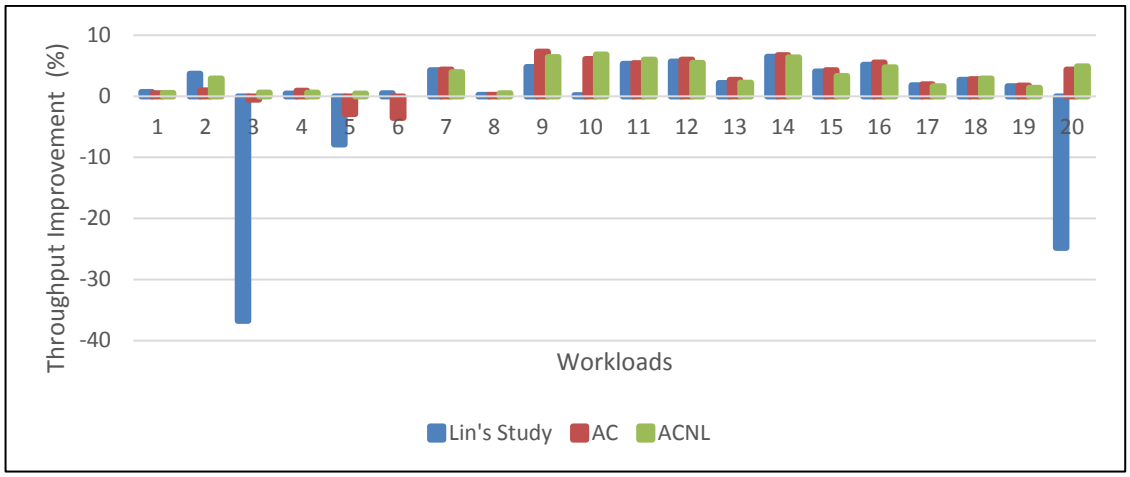

Figure 6.29. Improvement for all workloads on a 4-thread SMT with a 32-entry WIB

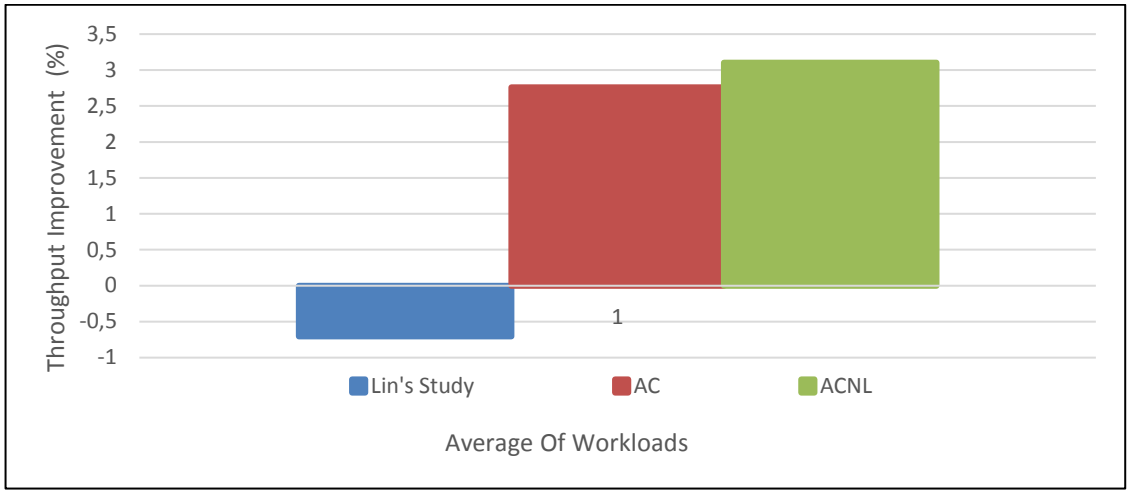

Figure 6.30. Average improvement for all workloads on a 4-thread SMT with a 32-entry WIB

Figure 6.31 shows the improvement percentage of the system results of workloads on the 8-thread SMT with a 32-entry WIB system settings. We can see that Lin's study has a negative impact on the system for at least two workloads. In figure 6.32, we observe the average improvement of our proposed algorithm has better performance than Lin's study.

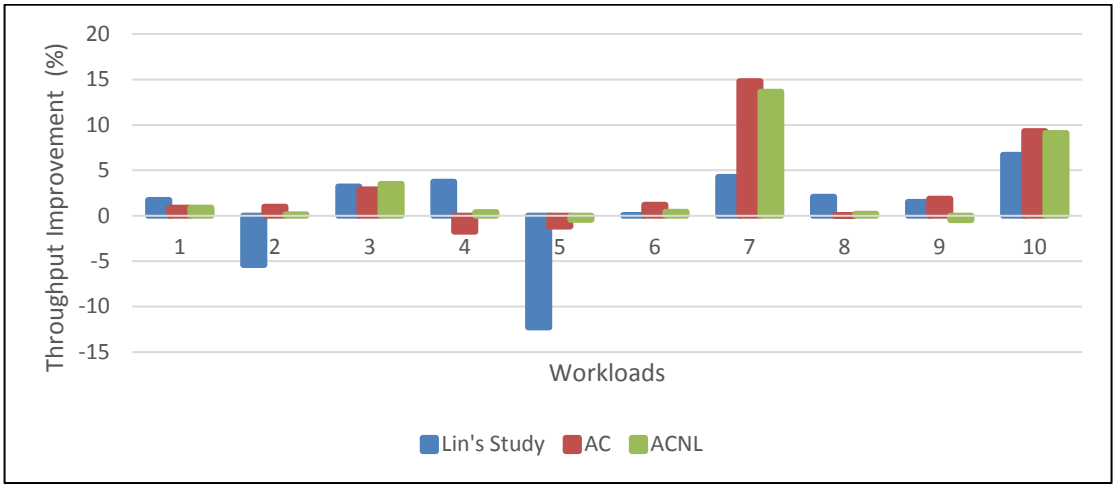

Figure 6.31. Improvement for all workloads on a 8-thread SMT with a 32-entry WIB

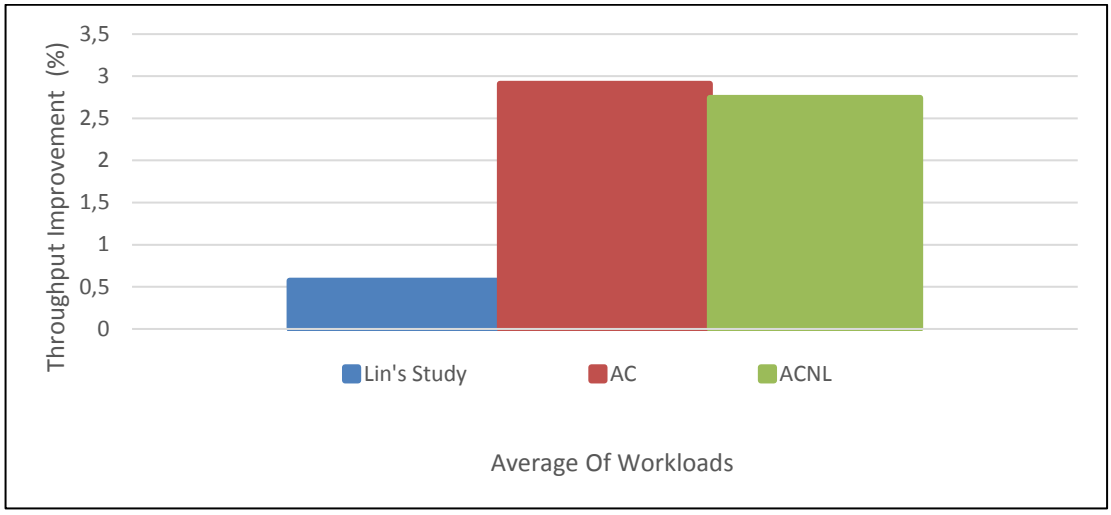

Figure 6.32. Average improvement for all workloads on a 8-thread SMT with a 32-entry WIB

In figure 6.33, we can see the improvement percentage of the system results of each workload on a 4-thread SMT with a 16-entry WIB system settings. We can see Lin's study has five workloads that have a negative impact on the overall system performance. While our proposed algorithm AC has two workloads that have a negative effect, ACNL has three workloads with a negative effect in this setting. In figure 6.34, it is shown that Lin's study has a negative impact on the system on average while our proposed algorithm has a positive impact by 7 % improvement over the baseline system.

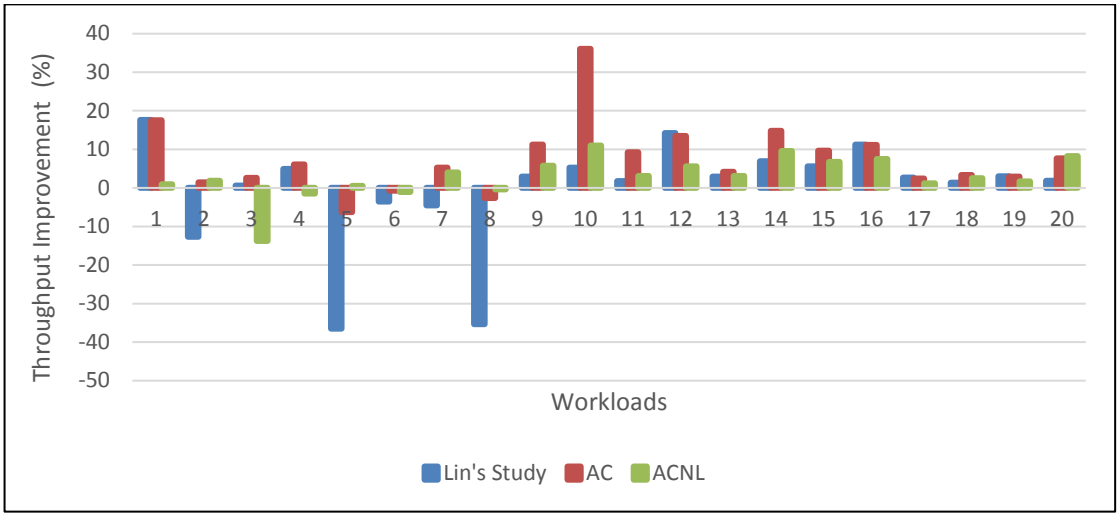

Figure 6.33. Improvement for all workloads on a 4-thread SMT with a 16-entry WIB

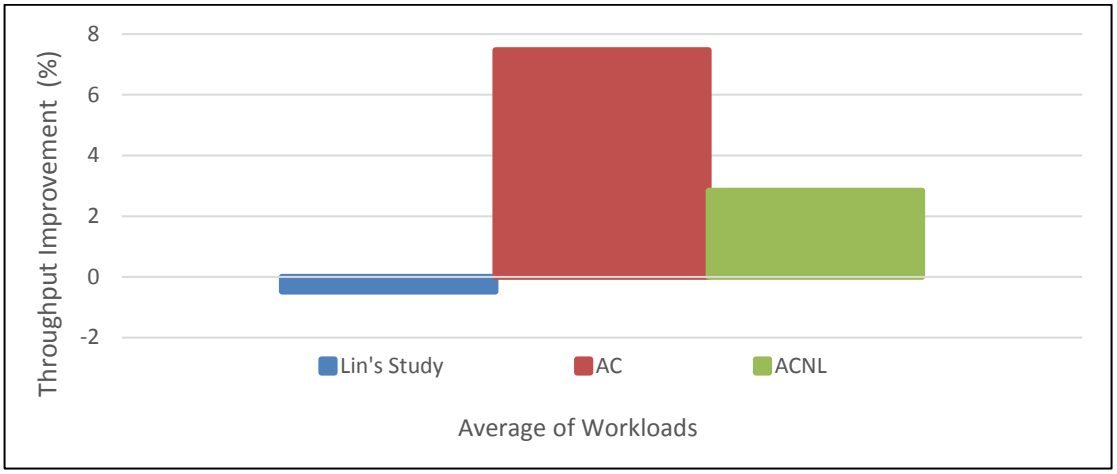

Figure 6.34. Average improvement for all workloads on a 4-thread SMT with a 16-entry WIB

We see the test results on the improvement percentage of the system for all workloads on a 8-thread SMT with a 16-entry WIB system settings in Figure 6.35. It is seen that Lin's study has one workload that has a performance degradation on the system. In figure 6.36, our proposed algorithms have higher performance improvement than that of Lin's study. ACNL algorithm has over 12 % improvement in the overall system performance.

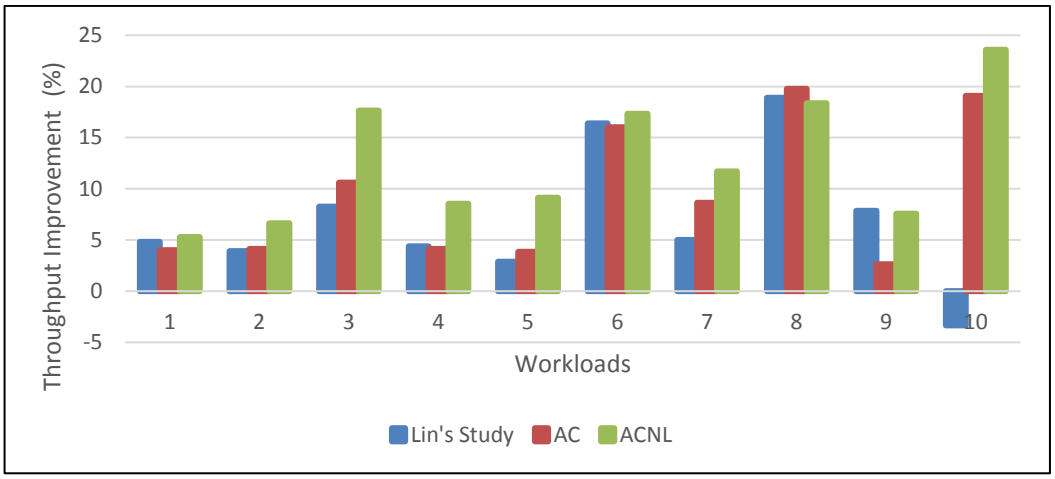

Figure 6.35. Improvement for all workloads on a 8-thread SMT with a 16-entry WIB

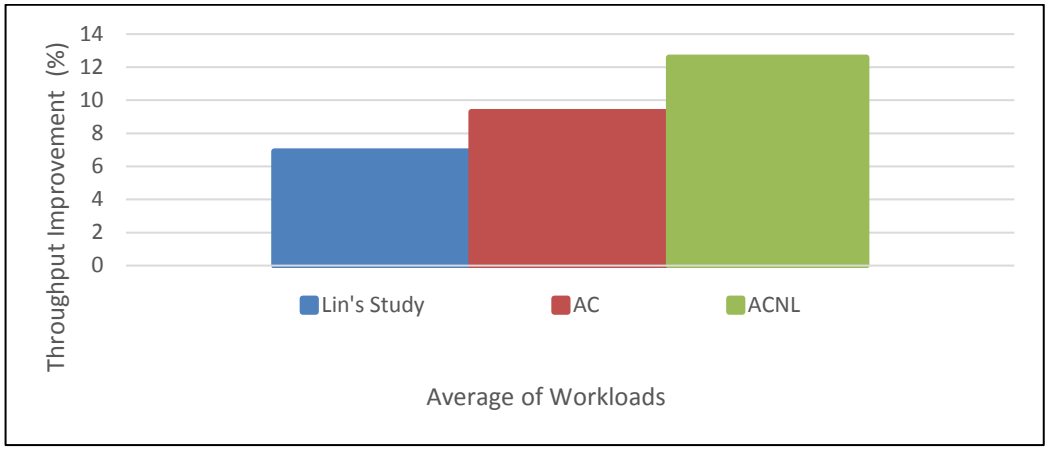

Figure 6.36. Average improvement for all workloads on a 8-thread SMT with a 16-entry WIB

As a result of the tests we carried out, we can say that our proposed algorithms provide better results than that of Lin's study for all settings that we studied. In a 4-thread settings, for both WIB sizes of 16 and 32, Lin's study has a negative improvement over the baseline. For a 8-thread and WIB size of 16 setting, Lin's study has the highest improvement among its results while our proposed algorithms still provide better improvement. While our adaptive capping algorithm *AC* shows better improvement on a 8-thread SMT with a 32-entry WIB and a 4-thread SMT with a 16-entry WIB, *ACNL* provides a better improvement on a 4-thread SMT with a 32-entry WIB and a 8-thread SMT with a 16-entry WIB.

7. CONCLUSION

An autonomous and dynamic capping algorithm for the waiting instruction buffer (WIB), which is one of the shared resources in SMT systems, is proposed in this thesis. In SMT systems, to achieve a higher throughput, threads compete for the common resources. Therefore, exploiting key datapath components and critical shared resources play an important role for high performance SMT systems [12]. In this paper, we show that an adaptive algorithm to control the allocation and distribution of the WIB significantly improves the overall system performance and fairness among threads. Moreover, it is shown that the proposed algorithm achieves the optimum performance on various workloads and different system settings. We propose two dynamic algorithms for capping WIB allocation. The first one is the dynamic capping technique in which we adjust the cap values of threads without exceeding the maximum number of WIB entries. The other algorithm we propose does not limit the number of capping that we set for the threads to make the threads with the highest utilization of resources keep allocating if they need more resource entries. Both algorithms have better performance than Lin's study, which we take as a reference study to improve. The reason that our algorithms have better performance is that Lin's study has too short sampling windows and global capping for all threads. Since Lin's study has short sampling windows it leads the inconsistent and unstable adjustments due to arbitrary changes in the behaviors of threads in short periods of execution. Moreover, using a global cap value for all threads does not give required adjustments that are needed to be applied for specific threads. Since all threads have the same cap value, starvation and over-use of resources have a negative impact on overall system performance. As can be seen in our test results, our proposed algorithm (AC) provides a 7.46 % improvement on a 4-thread system with an WIB size of 16 while Lin's study has a negative impact on the average. Our other proposed algorithm shows a 12.57 % improvement on an 8-thread system with an WIB size of 16 while Lin's study provides a 6.90 % improvement on the system. For the fairness results, our AC algorithm has greater results than the Lin's study for all test settings while the ACNL algorithm has a greater fairness result for only 8-thread system with an WIB size of 16.

REFERENCES

1. Zhang Y, Lin W. Autonomous issue queue distribution control for simultaneous multi-threading CPUs. J*ournal of Emerging Trends in Computing and Information Sciences. 2015*.

2. Sahba A, Zhang Y, Hays M, Lin WM. A real-time per-thread IQ-capping technique for simultaneous multi-threading (SMT) processors. *International Conference on Information Technology: New Generations.*; *2014*.

3. Tullsen DM, Brown JA. Handling long-latency loads in a simultaneous multithreading processor. *34th International Symposium on Microarchitecture*. 2001.

4. Cazorla FJ, Ramirez A, Valero M, Fernandez E. Dynamically controlled resource allocation in SMT processors. *37th International Symposium on Microarchitecture. 2004*.

5. Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM. Simultaneous multithreading: *IEEE Micro. 1997.*

6. Nagaraju T, Douglas C, Lin WM, Jhon E. Effective dispatching for simultaneous multi-threading (SMT) processors by capping per-thread resource utilization. *The Computing Science and Technology International Journal*. *2011;2(1):5-14*.

7. Wang H, Koren I, Krishna CM. An adaptive resource partitioning algorithm for SMT processors. *In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques on; 2008: ACM.*

8. Zhang Y, Douglas, C, Lin W. On maximizing resource utilization for simultaneous multi-threading (SMT) processors by instruction recalling. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) on; 2012*.

9. Zhang Y, Lin WM. Write buffer sharing control in SMT processors. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) on; 2012.*

10. Carrol S, Lin WM. Dynamic issue queue allocation based on reorder buffer instruction Status. *International Journal of Computer Systems. 2015;9(2):395-404.*

11. Sahba A, Sahba R, Lin WM. Improving IPC in simultaneous multi-threading (SMT) processors by capping IQ utilization according to dispatched memory instructions. *World Automation Congress. 2014.*

12. Zhang Y, Lin WM. Intelligent usage management of shared resources in simultaneous multi-threading processors. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) on; 2015: ACM.*

13. Vandierendonck H, SeznecManaging A. SMT resource usage through speculative instruction window weighting. *Transactions on Architecture and Code Optimization (TACO), Vol. 8, No. 3, November, 2008.*

14. Küçük G, Uslu G, Yesil C. History-based predictive instruction window weighting for SMT processors. *Springer International Publishing Switzerland. 2014.*

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