Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2021) 29: 831 – 844
© TÜBİTAK
doi:10.3906/elk-2005-50

Dynamic issue queue capping for simultaneous multithreaded processors

Merve YILDIZ GÜNEY[1,*], **Büşra KURU**[1], **Sercan SARI**[1], **İsa Ahmet GÜNEY**[2], **Gürhan KÜÇÜK**[1]

[1]Department of Computer Engineering, Faculty of Engineering, Yeditepe University, İstanbul, Turkey
[2]Department of Computer Engineering, Faculty of Engineering, Doğuş University, İstanbul, Turkey

Received: 18.05.2020 • **Accepted/Published Online:** 10.08.2020 • **Final Version:** 30.03.2021

Abstract: A simultaneous multithreaded (SMT) processor mixes multiple instruction streams in its superscalar out-of-order execution core for higher throughput. To achieve this, a superscalar processor is modified in such a way that some of its resources are duplicated and the rest is shared among multiple threads. The issue queue (IQ), which holds all waiting instructions until they become ready and scheduled for execution, is among these shared resources. A baseline unmanaged IQ can give an unexpectedly low performance since a hungry thread can tie up most of the IQ entries. This type of scenario is also worse in terms of the fairness metric since some of the threads may experience starvation. Earlier studies propose both static and a limited type of dynamic capping of the IQ entries for regulating IQ traffic and providing better SMT throughput and fairness. In this study, we propose an efficiency-based dynamic capping (EDC) algorithm that calculates an efficiency metric for each thread and allocates the IQ entries for maximizing the throughput and the fairness metrics. EDC gives 3.6% better throughput and 3.9% better fairness results compared to the current state-of-the-art algorithms, on the average.

Key words: Simultaneous multithreaded processor, issue queue, resource capping, throughput, fairness

1. Introduction

A superscalar out-of-order (OoO) processor has a pipelined datapath that allows the completion of multiple instructions of a single thread per clock cycle. Such architectures are highly common among all types of commercial processors due to their inherent ability to exploit instruction level parallelism of processes and improve performance. The average number of instructions per clock cycle (IPC) is the major metric to measure the processor performance. Various datapath resources may sit idle during the execution of instructions from a single thread. For instance, when the processor is running the operating system code, most of the floating-point ALUs are not utilized. Similarly, when a thread is in its computation-intensive code region, the cache structure might be underutilized. A simultaneous multithreaded (SMT) processor, which is a modified version of a traditional superscalar processor, is introduced to solve this resource underutilization problem by allowing instructions of more than one thread to coexist and execute on the same datapath. Once we reduce the degree of resource underutilization, we can also expect an immediate throughput boost as its positive side-effect.

When a superscalar processor is modified to handle an incoming SMT traffic, it utilizes dedicated architectural register files, reorder buffers (ROB), and rename tables for each running thread while sharing enlarged versions of the existing physical register files, the branch predictor, caches, the fetch queue, and

*Correspondence: myildiz@cse.yeditepe.edu.tr the issue queue (IQ). Although the dedicated structures do not impose much of a research challenge, shared structures require special research attention. There are plenty of studies in the literature that achieve to improve performance by analyzing shared structures such as caches [1, 2], write buffer entries [3], and register files [4–6]. In this study, we focus on one of the crucial shared datapath structures (i.e. the IQ) that schedules ready instructions to the OoO execution core of an SMT processor.

The IQ is a centralized structure that accepts instructions from multiple threads. There are two major components of this structure: 1) the wake-up, and 2) the selection logic. The wake-up logic is responsible for waking up instructions whose source operands become valid by setting their corresponding ready flag within the IQ. When a ready flag is set, its corresponding instruction suddenly becomes a candidate for execution. Then, in an n-way SMT processor, the selection logic selects at most n ready instructions for execution among all those candidate instructions. There are various scheduling algorithms proposed and studied for the selection logic. Generally, the best algorithm for the hardware realization is a simple positional-based algorithm, in which the hardware scans IQ entries from the beginning to the end until it locates and schedules n ready instructions. The selection algorithm must complete the instruction selection task as fast as possible since it is in the critical path of the processor datapath. For that reason, the load balancing task on the number of instructions from each thread is usually targeted by the earlier fetch stage with the help of a fetch policy [7–11].

A performance bottleneck occurs when some threads insert too many instructions into the IQ and disallow others from exploiting instruction level parallelism by having a larger window. Tullsen et al. examine several fetch policies to prevent IQ clogging [7]. They examine both utilizing different round-robin schemes as well as prioritizing threads based on some novel metrics such as least unresolved branches (BRCOUNT), least data cache misses (MISSCOUNT), fewest instructions in the IQ (ICOUNT), and the youngest instruction at the head of the ROB (IQPSN).

Some research in the literature focuses on ceasing fetch from threads which are likely to cause IQ clogs. Tullsen and Brown state that threads with long-latency load instructions are likely to clog the IQ due to instructions depending on these long-latency loads and propose blocking threads from fetching instructions until they complete their long-latency loads [8]. Eyerman and Eeckhout argue that not all long-latency loads are the same, and allowing threads to fetch more instructions can improve their performance if a thread can overlap the penalties of long-latency loads by inserting independent load instructions [9]. El-Moursy and Albonesi propose preventing threads from fetching instructions as long as the number of load instructions in the pipeline which resulted in an L1 data cache miss is above a certain threshold [11].

Apart from policies that attack the problem by preventing threads from fetching, some work focus on distributing the resources among threads. Cazorla et al. allocate shared resources to threads by classifying them as slow or fast based on the existence of an L2 miss [10]. Threads are also classified as active and inactive for each resource type according to their recent utilization of that resource. They allocate resources only to active threads, and the amount is determined based on the number of fast and slow threads that are present in the system.

Choi and Yeung choose to utilize the throughput of the system as a feedback metric and propose a hill-climbing algorithm that dynamically manipulates how much IQ entries are allocated to each thread [12]. Bitirgen et al. criticize their work for not using any learning models [13]. In another work, Wang et al. propose an efficiency metric that indicates how much work is done by each thread compared to their allocated resources and periodically allocates additional resources to the most efficient thread [14]. Another study proposes a method to improve thread coscheduling by utilizing a probabilistic model for prediction [15].

The idea of capping IQ entries was first proposed as a static capping method [16]. The authors of the paper study various cap values, and for a 32-entry IQ on an 8-instruction wide 4-threaded SMT processor, they claim that the static cap value of 12 is the magic number giving the optimal performance. The authors also propose a limited type of dynamic algorithm that assigns either a cap value of 12 for fast threads or a cap value of 5 for slow threads [17]. In that method, the average number of blocked instructions for each thread is calculated within a 256-cycle period, and then, a cap value of 5 is applied when that average number of blocked instructions of a thread is larger than 28, assuming that the corresponding thread is slow. Otherwise, the thread is assumed to be a fast thread, and it is given a larger cap value of 12. In a later study, the authors also propose an autonomous IQ distribution control, in which the cap value can be either increased or decreased according to the overall issue rate of the system [18].

To better motivate our work, we run both the baseline configuration with an unmanaged IQ and the static capping configuration with a fixed capping of 12 IQ entries on the M-Sim simulator with the identical SMT settings [19]. Figures 1 to 4 compare the average IQ occupancy traffic on both baseline and static capping configurations for two different workloads containing a set of selected benchmarks from the SPEC2000 suite. In these figures, the epoch length is selected to be 256 K cycles. Figures 1 and 2 depict a scenario with the $<$*mgrid,mcf,mgrid,mcf*$>$ 4-thread workload, in which the baseline configuration gives 4.5% better performance over the static capping method. Here, the important thing is that there is almost no ALU conflict (only 0.7 ALU conflicts per cycle) among threads in the baseline configuration. An ALU conflict instance occurs when a ready instruction cannot be scheduled to its corresponding ALU since that ALU is already busy executing another instruction. With the static configuration, average ALU conflicts per cycle drop to 0.5; however, high-performance *mgrid* threads cannot allocate enough IQ entries to show their best performance. In Figure 1, we see that *mgrid* threads may need up to 20 IQ entries on the average to show higher performance results, and the baseline configuration gives them such opportunity.

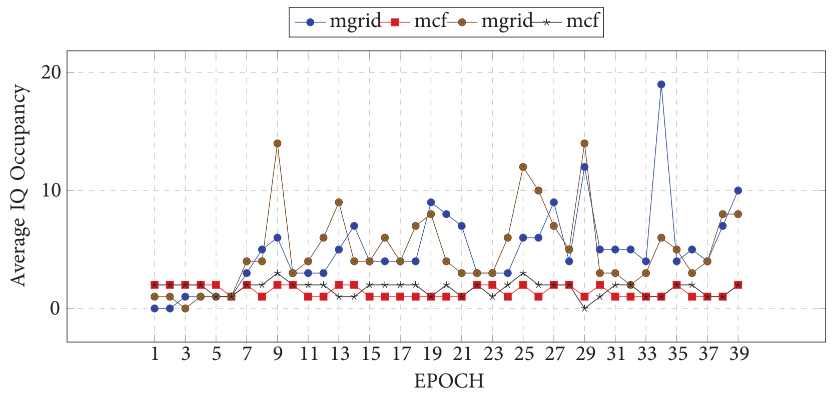

Figure 1. Average IQ occupancy traffic for <mgrid,mcf,mgrid,mcf> workload on the baseline configuration.

Figures 3 and 4 depict an opposite case with the $<$*bzip2,applu,parser,art*$>$ 4-thread workload, in which the static capping method gives more than 32% better performance over the baseline configuration. Here, the important thing is that there is an average of 4.5 ALU conflicts per cycle among threads in the baseline configuration. With the static capping strategy, average ALU conflicts per cycle drops to 3.5, meanwhile, all threads receive a sufficient number of IQ entries to show their best-possible performance with such limited

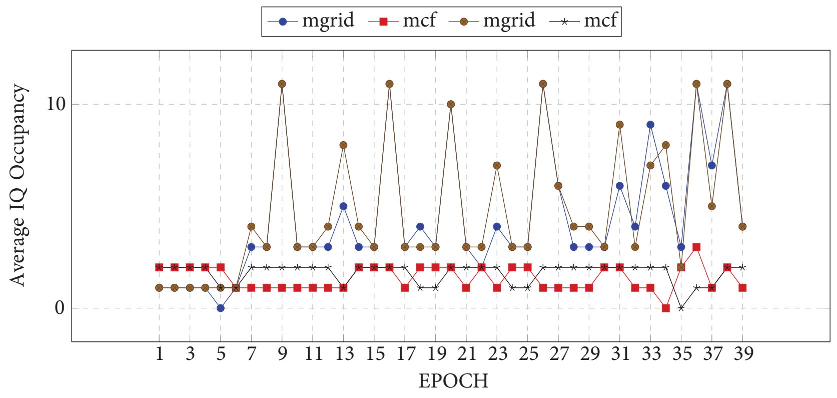

Figure 2. Average IQ occupancy traffic for <mgrid,mcf,mgrid,mcf> workload on the configuration with the static capping method.

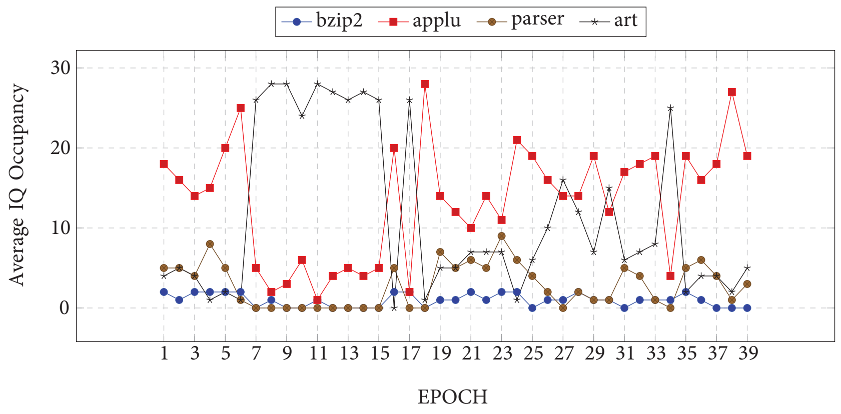

Figure 3. Average IQ occupancy traffic for <bzip2,applu,parser,art> workload on the baseline configuration.

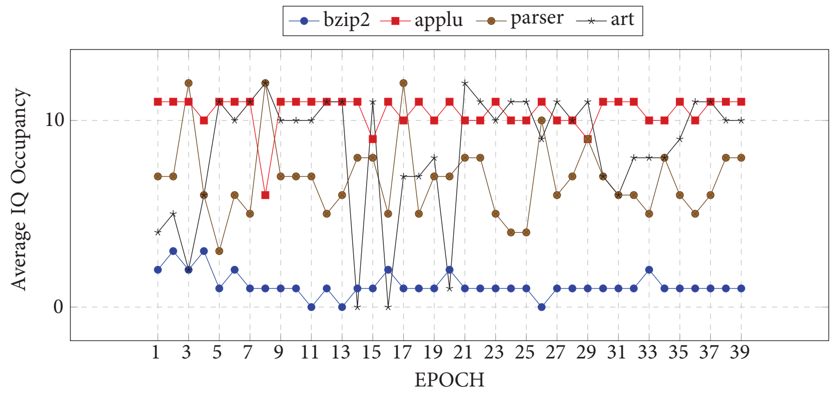

Figure 4. Average IQ occupancy traffic for <bzip2,applu,parser,art> workload on the configuration with the static capping method.

resources. In Figure 3, we see that the *art* thread receives almost all IQ entries between epochs 7 and 16, and that creates long periods of starvation on the other remaining threads. As a result, the baseline configuration performs much worse than the static capping method on this workload.

To summarize, we empirically show that there are workloads and execution intervals where the baseline allocation method and static capping outperform one another. Furthermore, we show that some threads may require a higher number of IQ entries than a fixed upper limit defined by the system, and in some cases, we may benefit from a higher overall throughput by allocating a high number of entries to a single thread without significantly lowering the performances of other threads.

We motivate this study regarding the above observations. Devising a new allocation method which avoids starvation and low fairness by allocating IQ entries based on performance, while not limiting the allocation options by imposing a fixed upper bound can help us improve both throughput and fairness metrics compared to the baseline and static capping configurations.

Our contributions in this paper are as follows:

- We show that both the baseline configuration and the static capping method cannot be always effective in all kinds of workload mixtures.
- We propose an adaptive capping mechanism to achieve a more fair thread performance and higher SMT throughput.
- Our proposed mechanism uses dynamic capping values instead of relying on fixed thresholds and ensures that all IQ entries are always utilized.
- Our proposed mechanism is insensitive to epoch durations and can provide a stable performance even without using fine-grain decision intervals.

The rest of the paper is organized as follows. Section 2 describes the proposed mechanism that aims for better throughput and fairness results followed by our simulation methodology in Section 3. Section 4 presents our simulation results. Finally, we conclude our paper in Section 5.

2. The proposed design

The authors of the earlier static IQ capping paper also propose a dynamic method in which the IQ cap of each thread is set according to its dynamically determined thread category of either slow or fast [17]. We also focus on an alternative dynamic IQ capping method which is known as autonomous IQ distribution control [18]. Before explaining our proposed efficiency-based dynamic IQ capping method, we would like to review these earlier approaches.

2.1. Limited dynamic IQ capping

We call this earlier method as the limited dynamic IQ capping (LDC) method since it only allows two cap offsets (5 for slow threads and 12 for fast threads for a 32-entry IQ on a 4-threaded SMT processor) rather than allowing a range of cap values.

The LDC method, which is shown in Algorithm 1, requires the knowledge of the average number of blocked instructions for each thread in clock cycle granularity. The mechanism keeps a running count of the number of instructions which are dispatched with operands not in ready state in BAD (i.e. blocked at dispatch) vector and completed but not committed in the BAC (i.e. blocked at commit) vector. The average number of blocked instructions is calculated by dividing the total blocked instruction count by the epoch length E. Next, the algorithm checks whether the average number of blocked instructions is less than a predefined threshold

T for each thread. If it is less than the predefined threshold, the algorithm updates the IQ cap value of that thread as 12. Otherwise, the method assumes that the number of blocked instructions of that thread is too high, and it classifies the thread as slow and sets its cap value to 5.

Algorithm 1 The limited dynamic capping algorithm

```
if $EPOCH$ ends then
    for each thread $i$ do
        $BAD[i] \leftarrow BAD[i]/E$
        $BAC[i] \leftarrow BAC[i]/E$
        if $(BAD[i] + BAC[i]) \leq T$ then
            $CAP[i] \leftarrow 12$
        else
            $CAP[i] \leftarrow 5$
        $BAD[i] \leftarrow 0$
        $BAC[i] \leftarrow 0$
```

Although this algorithm is presented as a dynamic IQ capping algorithm, the predefined threshold value restricts the number of possible IQ cap values to just two constants. Eventually, threads receive either 5 or 12 IQ entries, and this can cause an inefficient allocation of IQ entries among threads. Additionally, in some cases, the total cap can be smaller than the IQ size. For instance, if all threads are selected as slow, the total cap of the system is restricted to 20 entries for a 4-threaded SMT processor, and 12 entries sit idle. Such allocations may cause underutilization of resources.

2.2. Autonomous IQ distrubution control

The autonomous IQ distribution control (ADC) method focuses on a single IQ cap value that can be dynamically moved every 1000 cycles [18]. While this mechanism needs no a priori information about system environments or workloads, it applies the same cap value to the all threads. The algorithm, which is shown in Algorithm 2, counts the number of issued instructions, and if that number is larger than the number of issued instructions of the previous period by a certain threshold rate T, it increases the cap value by $DELTA$ amount. Otherwise, if the number of issued instructions in the previous period is larger than the number of issued instructions of the current period by the same threshold rate T, then the cap value is decremented by $DELTA$ amount. We believe that applying the same cap value to all threads may not be as efficient as dedicating an individual cap value to each thread for considering their instantaneous demands.

Algorithm 2 Autonomous IQ distrubution control Algorithm

```
if $EPOCH$ ends then
    for each thread $i$ do
        $total\_issue \leftarrow total\_issue + issue\_count[i]$
        $issue\_count[i] \leftarrow 0$
    if $(total\_issue - prev\_total\_issue) \geq (T * prev\_total\_issue)$ then
        $CAP \leftarrow CAP + DELTA$
    else
        if $(prev\_total\_issue - total\_issue) \geq (T * prev\_total\_issue)$ then
            $CAP \leftarrow CAP - DELTA$
    $prev\_total\_issue \leftarrow total\_issue$
```

2.3. The efficiency-based dynamic capping algorithm

In our proposed efficiency-based dynamic capping (EDC) algorithm, we utilize an efficiency metric similar to the committed instructions per resource entry ($CIPRE$) metric of an earlier study [14]. First, we collect the average number of ROB entries of each thread in the R vector for the current epoch. Then, we collect the number of committed instructions in the *cc* (i.e. commit count) field of each thread T for the same epoch. Meanwhile, we also remember the previous commit count in the *pcc* field of the same thread, which is collected from the previous epoch. Details of our proposed algorithm, efficiency-based dynamic capping algorithm, are shown in Algorithm 3.

Algorithm 3 The Efficiency-Based Dynamic Capping Algorithm (EDC)

```
if $EPOCH\ ends$ then
    for $each\ thread\ i$ do
        if $R[i] = 0$ then
            $EFF[i] \leftarrow 0$
        else
            $EFF[i] \leftarrow (T[i].cc - T[i].pcc[i])/R[i]$
        $T[i].pcc \leftarrow T[i].cc$
        $R[i] \leftarrow 0$
    $(MINidx,\ MAXidx) \leftarrow findMINMAX(EFF)$
    $CAP[MAXidx] \leftarrow CAP[MAXidx] + DELTA$
    $CAP[MINidx] \leftarrow CAP[MINidx] - DELTA$
```

To calculate the efficiency for a certain thread in a certain epoch, we divide the difference between committed instructions in consecutive epochs by the average occupancy of ROB entries and store the result in the efficiency vector EFF. The efficiency value represents how well a thread performs with the given amount of resources. If it is high, it shows that a thread can commit a large number of instructions per ROB entry, and this makes the thread an efficient thread. Otherwise, we can consider the thread as an inefficient one.

In EDC, we consider the efficiency of all threads and we try to allocate IQ entries among the threads according to their instantaneous demands. After the calculation of the efficiency vector, we determine two threads: 1) a thread with the minimum efficiency value whose cap value is not the lower cap limit, and 2) a thread with the maximum efficiency value whose cap value is not the upper cap limit. In this study, we selected the lower and upper cap limit as 5 and 12, respectively, and initially all caps are set to 8. In Algorithm 2, the *findMINMAX* function accepts the EFF vector and returns indices of suitable threads with minimum and maximum efficiency. Here, $DELTA$ stands for how many IQ entries efficient and inefficient threads can exchange at a single decision step. In our tests, we only utilized the $DELTA$ value of 1, since the gap between the lower and the upper limits are chosen to be small. The main idea of our algorithm is to adjust the optimum sharing of IQ entries among threads that have the maximum and the minimum efficiency.

To clarify the key points of EDC, we give an example of a 2-threaded environment. Assume that we have threads $T1$ and $T2$. At the end of an epoch, we run our algorithm and calculate the individual efficiency value of each thread. Next, if we see that $T1$ works more efficiently than $T2$, to balance between threads, we reduce the cap value of $T2$ by $DELTA$ entries while increasing the cap value of $T1$ by $DELTA$ entries. By doing so, we provide an adaptive capping mechanism to achieve a more stable and fair resource sharing mechanism among threads. Rather than giving a static value of 5 or 12 to threads or giving the same cap value to all of the threads, we give them a chance to receive cap values that are more suitable to their instant needs. By doing that, while we eliminate unfair resource sharing, we also provide an SMT processor with better throughput.

3. Experimental methodology

3.1. Simulator and workloads

The simulator used in this research is M-Sim [19]. It is an extended version of the well-known SimpleScalar simulator offering both single- and multithreaded simulations. M-Sim enables us to collect cycle-accurate processor statistics, to implement our proposed algorithm, and to accurately compare its results with those of other algorithms in the literature. However, simulators are too slow to run any workload from its beginning to its end. Therefore, we run our simulations for 200M instructions on each workload after skipping first 100M instructions. Table 1 gives the fixed parameters of the simulated processor. To compare our proposed design with the current state of the art, we chose a similar processor configuration and similar simulation regions described in the original study [17].

SPEC CPU2000 and CPU2006 benchmark suites are used to evaluate our proposed design. We selected 20 4-threaded workloads showing various characteristics in favor of the LDC, ADC, or the EDC algorithms. These 20 combinations of workload mixtures are shown in Table 2.

Table 1. Simulation parameters.

Parameter	Description
Superscalar width	8
Number of threads	4
IFQ/ROB/LSQ/IQ size	32/128/48/32 entries
Function unit: Number of units/ completion latency/issue latency	integer ALU: 6/1/1 integer mult: 1/3/1 Integer Div: 1/12/12 Floating point Add: 4/2/1 Floating point Mul: 1/4/1 Floating point Div: 1/20/19 Floating point Sqrt: 1/24/24
Level-1 instruction cache	512 sets, 64-byte blocks, 2-way associativity, LRU
Level-1 data cache	256 sets, 64-byte blocks, 4-way associativity, LRU
Level-2 unified cache	512 sets, 64-byte blocks, 16-way associativity, LRU
Write buffers	16 64-byte entries
Branch predictor	2048-entry bimodal
Physical registers	256 integer and floating point

3.2. Metrics

Prevalent metrics for a multithreaded environment are throughput and fairness [20]. The throughput is defined as the sum of IPC of each thread, where n indicates the number of threads per mix in the system, as shown in

Table 2. Workload mixtures.

Mix	Benchmarks
1	libquantum, milc, sjeng, zeusmp
2	libquantum, omnetpp, sjeng, zeusmp
3	mesa, mcf, parser, twolf
4	hmmer, milc, omnetpp, zeusmp
5	hmmer, libquantum, namd, zeusmp
6	mcf, milc, omnetpp, sjeng
7	mcf, milc, namd, sjeng
8	mcf, milc, omnetpp, zeusmp
9	mcf, milc, sjeng, zeusmp
10	libquantum, milc, namd, sjeng
11	libquantum, milc, omnetpp, sjeng
12	gcc, gcc, gcc, gcc
13	art, gcc, vortex, mesa
14	bzip2, art, mesa, twolf
15	libquantum, namd, omnetpp, sjeng
16	mgrid, mcf, mgrid, mcf
17	milc, omnetpp, sjeng, zeusmp
18	bzip2, applu, parser, art
19	milc, namd, omnetpp, sjeng
20	mcf, namd, omnetpp, sjeng

Equation (1) below.

$$Throughput = \sum_{i}^{n} IPC_i \tag{1}$$

Harmonic IPC is a popular metric to calculate the degree of fairness among the threads (see Equation (2)). Even if one of the running threads experiences a high performance drop compared to its standalone performance, the harmonic IPC (or fairness) drops drastically.

$$Fairness = \frac{n}{\sum_{i}^{n} \frac{StandaloneIPC_i}{IPC_i}} \tag{2}$$

In this paper, these two metrics are used to compare our proposed algorithm of efficiency-based dynamic capping to its competitors [17]. The metric shown in Equation (3) indicates the throughput improvement percentage averaged over the studied mixtures. Here m denotes the number of workloads in our simulation. We also measure the improvement percentage of fairness similarly.

$$Improvement\% = \sum_{j}^{m} \frac{Throughput_{EDC} - Throughput_{competitor}}{Throughput_{competitor}} \tag{3}$$

4. Results

In this section, we compare four processor configurations (i.e. baseline, LDC, ADC, and EDC) in terms of throughput and fairness metrics followed by a sensitivity study on the epoch length. The baseline configuration represents an SMT processor with an unmanaged issue queue structure. The results indicate that EDC provides the highest average throughput and fairness compared to its competitors.

4.1. Throughput

Figure 5 shows the throughput comparison of the baseline, the LDC, ADC, and the EDC algorithms. In the first four workloads, the LDC performs better than the EDC. However, in the fourth workload, the baseline and ADC configurations are the winners. The baseline configuration performs slightly better than both LDC and EDC algorithms in workloads 9, 11, and 16, as well. This phenomenon occurs when the ALU requirements of the running threads do not conflict too much. When running threads are harmonious, the baseline configuration becomes the best configuration for performance, since it does not limit the IQ requirement of any of the running threads. However, when the average number of ALU conflicts per cycle increases, the baseline configuration receives a considerable performance hit. Workloads 5, 13, 14, and 18 show such example behavior.

In Figure 5, LDC algorithm performs worst on the rightmost workloads. It simply cannot capture the IQ requirements of the running threads well (almost 20% worse performance compared to the baseline configuration). ADC algorithm, on the other hand, performs worse than LDC and EDC algorithms in workloads 3, 5, 8, 12, 13, 14, and 18. However, it is never the worst performing algorithm in any of the workloads. Our EDC algorithm and its efficiency-based adaptive nature change cap values of individual threads such as that it even slightly beats the baseline and ADC configurations. In the end, the EDC performs 3.6% better than the ADC, 3.7% better than the LDC and 5.8% better than the baseline case across all simulated workloads, on the average.

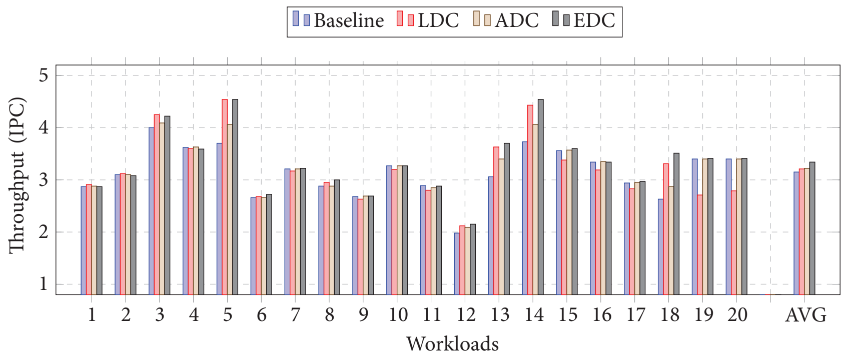

Figure 5. Throughput comparison of the baseline, LDC, and EDC mechanisms.

4.2. Fairness

Generally, fairness and throughput metrics are two conflicting metrics. For instance, we can give all the resources to one of the best performing threads, and achieve top performance. In that case, however, since the other threads start to starve, the fairness value goes to almost flat zero.

Figure 6 shows fairness comparison of the baseline, LDC, ADC, and the EDC algorithms. In workload 5, we get an unusual result: although dynamic approaches beat the baseline configuration by a large margin in terms of throughput, they also beat it in terms of fairness, by even a larger margin, as well. Similar results are also observed in workloads 3, 12, 13, 14, and 18. This is again probably due to a high number of ALU conflicts per cycle while running the baseline case. Since there is no capping involved in the baseline configuration, too many ALU conflicts result in low performance on all running threads further lowering the fairness value. In the dynamic approaches, IQ capping helps to reduce such ALU conflicts, and as a result, all dynamic approaches provide better throughput and better fairness results compared to the baseline case. In contrast, in workload 11, the fairness of the baseline configuration is slightly higher than the fairness results of the dynamic approaches. Finally, two worst-performing workloads of LDC, workload 19 and 20, badly suffer in terms of fairness, as well.

To sum up, in terms of fairness, the EDC performs 7.7% better than the LDC, 7.5% better than the baseline, and 3.9% better than the ADC method across all simulated workloads, on the average. Our results show that rather than using fixed cap values or giving the same cap value to all threads, it is more efficient to adjust the cap value according to the demands of threads. Average fairness results for the baseline configuration and the LDC go head to head.

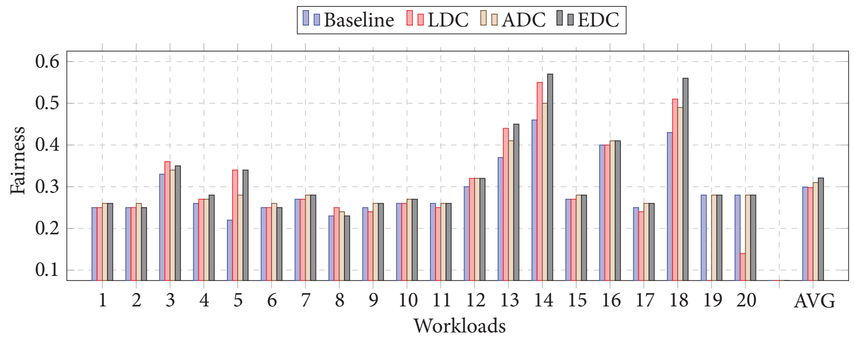

Figure 6. Fairness comparison of the baseline, LDC, and EDC mechanisms.

4.3. Sensitivity to epoch length

Finally, we study the effect of epoch length on our results. In our experiments, we study 256 cycles (which is taken from the LDC study in [17]) and 256K cycles of epoch lengths. Figure 7 shows the average results of throughput and fairness for both of these epoch lengths. Note that there is a very negligible difference in these results, and more surprisingly, the choice of a low-duty cycle (256K cycles) epoch length gives slightly better throughput and fairness. Remember that our EDC algorithm relies on the complex *findMINMAX* function, which may consume a considerable amount of energy if it is implemented on hardware and if it is run every 256 cycles. Therefore, this sensitivity analysis provides us an important result to prove the feasibility of our proposed mechanism in terms of its energy-efficiency.

Both LDC and ADC require very short decision intervals, which can create oscillations in the control logic. For instance, when we consider the ADC algorithm, it is highly possible that the issue rate of the processor may incline in one short period of time, and ADC immediately increases the cap size of all of the threads. In the end, such a decision may trigger a stall in the issue rate due to too many ALU conflicts, which is obviously the result of the decision taken in the previous period. Thus, due to a decline in the issue rate, ADC immediately decreases the cap size of all of the threads creating a loop, which forces ADC to reincrease the cap size in the next period. Similar oscillations can also happen in the LDC logic, since it utilizes even shorter decision intervals. EDC on the other hand, provides higher stability compared to its competitors since it can work with very large time intervals, which are immune to such resource oscillations.

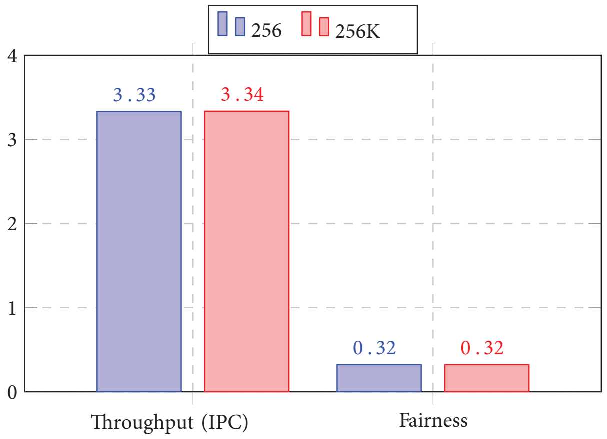

Figure 7. Results of the sensitivity study to 256 and 256K cycles of epoch lengths.

5. Conclusion

Simultaneous multithreaded (SMT) processors gained huge popularity recently, since they allow better processor resource utilization and better throughput. From the system perspective, these are two much-admired features. In this paper, we directly address the fairness problem in SMT processors, by applying an efficiency metric and capping the instruction queue (IQ) accordingly. In our proposed efficiency-based dynamic capping mechanism, which we call EDC, we periodically calculate the efficiency of each thread by dividing their committed number of instructions by the given average number of reorder buffer (ROB) entries in each period. At the end of a period, we take IQ entries from the thread with the minimum efficiency and give them to the thread with the maximum efficiency. This simple but effective dynamic method gives 3.6% better average throughput and 3.9% better average fairness results compared to its closest competitor, which is known as the autonomous IQ distribution control (ADC) method.

In the LDC and the ADC methods, the period length is set to 256 and 1000 cycles, respectively. These are indeed extremely fine-grain settings, and at the end of every 256 or 1000 cycles, a new capping decision is made. We show that our proposed method is insensitive to the period length, and even when we chose the period length of 256 K cycles, almost identical throughput and fairness results are still obtained. This is an important finding to show the energy-efficiency of our proposed method.

Currently, our proposed algorithm shares all available IQ resources among all active threads. A future research direction would be to determine how many IQ entries each thread requires in order to reach their potential performance, and whether all IQ entries should be kept powered or not. In the case where the total number of IQ entries required by the threads is less than the number of physical entries in the processor, the system can temporarily power down idle entries in order to save power by both powering down the entries themselves, and by partially powering off the out-of-order instruction selection logic.

Finally, in a recent study, Margaritov et al. propose an SMT processor, in which the threads are not equally favored [21]. When there is a latency-sensitive thread in the mixture, it might be served more resources than the rest of the threads to satisfy its predetermined quality-of-service (QoS) threshold. We believe that similar arrangements can be easily integrated into our proposed algorithm to satisfy QoS of certain threads in SMT workloads.

Acknowledgment

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E866.

References

[1] Díaz J, Hidalgo JI, Fernández F, Garnica O, López S. Improving SMT performance: an application of genetic algorithms to configure resizable caches. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers; New York, USA; 2009. pp. 2029-2034.

[2] Özer E. Low-cost and power-efficient thread collision detection scheme for shared caches in a real-time multithreaded embedded processor. Turkish Journal of Electrical Engineering and Computer Sciences 2013; 21(3): 714-733.

[3] 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); New York, USA; 2013. pp. 243-249.

[4] Gungorer H, Kucuk G. Dynamic capping of physical register files in simultaneous multi-threading processors for performance. In: 32nd International Symposium (ISCIS 2018); Poznan, Poland; 2018. pp. 41-48.

[5] Sheikh MN, Lin WM. Dynamic capping of rename registers for SMT processors. Journal of Systems Architecture 2019; 99: 1-20. doi: 10.1016/j.sysarc.2019.101637

[6] Zhang Y, Lin WM. Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors. Microprocessors and Microsystems 2016; 45 (PB): 270-282. doi: 10.1016/j.micpro.2016.06.002

[7] Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: maximizing on-chip parallelism. In: 25 Years of the International Symposia on Computer Architecture (Selected Papers); Barcelona, Spain; 1998. pp. 533-544.

[8] Tullsen DM, Brown JA. Handling long-latency loads in a simultaneous multithreading processor. In: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture; Austin, TX, USA; 2001. pp. 318-327.

[9] Everman S, Eeckhout L. A memory-level parallelism aware fetch policy for SMT processors. In: 2007 IEEE 13th International Symposium on High Performance Computer Architecture; Scottsdale, AZ, USA; 2007. pp. 240-249.

[10] Cazorla FJ, Ramirez A, Valero M, Fernandez E. Dynamically Controlled Resource Allocation in SMT Processors. In: 37th International Symposium on Microarchitecture (MICRO-37); Portland, OR, USA; 2004. pp. 171-182.

[11] El-Moursy A, Albonesi DH. Front-end policies for improved issue efficiency in SMT processors. In: The Ninth International Symposium on High-Performance Computer Architecture; Anaheim, CA, USA; 2003. pp. 31-40.

[12] Choi S, Yeung D. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In: 33rd International Symposium on Computer Architecture (ISCA'06); Boston, MA, USA; 2006. pp. 239-251.

[13] Bitirgen R, Ipek E, Martinez JF. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In: 41st IEEE/ACM International Symposium on Microarchitecture; Lake Como Italy; 2008. pp.318-329.

[14] Wang H, Koren I, Krishna CM. An Adaptive Resource Partitioning Algorithm for SMT processors. In: International Conference on Parallel Architectures and Compilation Techniques (PACT); Toronto, ON, Canada; 2008. pp. 230-239.

[15] Eyerman S, Eeckhout L. Probabilistic job symbiosis modeling for SMT processor scheduling. ACM SIGARCH Computer Architecture News 2010; 38(1): 91-102.

[16] Nagaraju T, Douglas C, Lin VM, John E. Effective Dispatching for Simultaneous Multi-Threading (SMT) Processors by Capping PerThread Resource Utilization. The Computing Science and Technology International Journal 2011; 1(2): 5-14.

[17] Carroll S, Lin W. Dynamic issue queue allocation based on reorder buffer instruction status. International Journal of Computer Systems 2015; 2 (9): 395-404

[18] Zhang Y, Hays M, Lin W, John E. Autonomous control of issue queue utilization for simultaneous multi-threading CPUs. Journal of Emerging Trends in Computing and Information Sciences 2015; 6 (12): 736-744.

[19] Sharkey JJ, Ponomarev D, Ghose K. M-SIM: A Flexible, Multithreaded Architectural Simulation Environment. Technical Report CS-TR-05-DP01. 2005.

[20] Luo K, Gummaraju J, Franklin M. Balancing throughput and fairness in SMT processors. International Symposium on Performance Analysis of Systems and Software (ISPASS); Tuscon, AZ, USA; 2001. pp.164-171.

[21] Margaritov A, Gupta S, Gonzalez-Alberquilla R, Grot B. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA); New York, USA; 2019. pp. 15-27.