QoSMT: Supporting Precise Performance Control for Simultaneous multithreading Architecture

Xin Jin[1,2], Yaoyang Zhou[1,3], Bowen Huang[1], Zihao Yu[1,3], Xusheng Zhan[4], Huizhe Wang[1,3], Sa Wang[1,3], Ningmei Yu[2], Ninghui Sun[1,3], Yungang Bao[1,3*]
[1]State Key Laboratory of Computer Architecture, ICT, CAS
[2]Xi'an University of Technology
[3]University of Chinese Academy of Sciences
[4]Huawei Technologies Co., Ltd.

ABSTRACT

Simultaneous multithreading (SMT) technology improves CPU throughput, but also causes unpredictable performance fluctuations for co-located workloads. Although recent major SMT processors have adopted some techniques to promote hardware support for quality-of-service (QoS), achieving both precise performance control and high throughput on SMT architectures is still a challenging open problem.

In this paper, we perform some comprehensive experiments on real SMT systems and cycle-accurate simulators. From these experiments, we observe that almost all in-core resources may suffer from severe contention as workloads vary. We consider this observation as the fundamental reason leading to the challenging problem above. Thus, we introduce QoSMT, a novel hardware scheme that leverages a closed-loop controlling mechanism to enforce precise performance control for specific targets, e.g. achieving 85%, 90% or 95% of the performance of a workload running alone respectively. We implement a prototype on GEM5 simulator. Experimental results show that the control error is only 1.4%, 0.5% and 3.6%.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*.

KEYWORDS

SMT Interference, Data Center, QoS, Performance Predictability

ACM Reference Format:
X. Jin, Y. Zhou, B. Huang, Z. Yu, X. Zhan, H. Wang and S. Wang, N. Yu, N. Sun, Y. Bao. 2019. QoSMT: Supporting Precise Performance Control for Simultaneous multithreading Architecture. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330364

*Xin Jin and Yaoyanng Zhou contributed equally to this research.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330364

1 INTRODUCTION

Simultaneous multithreading (SMT) technology is widely adopted in contemporary general-purpose processors. It improves the throughput of a processor by issuing instructions from multiple threads to achieve better utilization of in-core resources such as instruction queue (IQ), reorder buffer (ROB) and load-store queue (LSQ).

However, SMT may result in unpredictable performance variations when multiple applications run simultaneously on an SMT processor. Recent work has shown that applications may suffer varying performance degradation by as high as 70% due to SMT induced interference [33]. To confirm this, we conduct some experiments on an Intel i7-4770 server with Hyper-Threading [21]. As shown in Figure 1, SMT may hurt the performance of an application (e.g. perlbench) by 1.1X-2.1X when it is co-located with different applications.

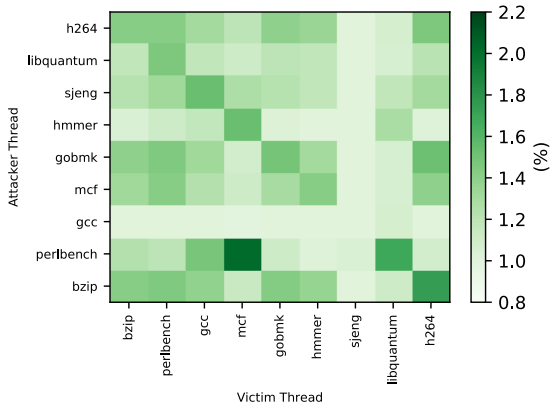

Figure 1: Performance variations of Spec2006 workloads

Recognizing the SMT-induced interference problem, Intel adopts a static partitioning approach to segregate two threads on shared pipeline resources such as IQ and ROB [21]. But it is inflexible for dynamic resource adjustment and may degrade both threads' performance due to reduced pipeline resources. IBM's POWER series processors allow to assign different priorities to workloads. And they provide different instruction fetch rates to guarantee the performance of high priority workloads [28]. But this approach makes other workloads hard to utilize in-core resources, against the original motivation of SMT. Although there is previous literature [9] [10]on hardware design for guarantee of quality-of-service (QoS) on SMT processors, most of their designs require profiling in-advance.

However, due to the limitation of privacy data protection and high operation and maintenance cost, the data center is reluctant to accept the off-line profiling.

Industrial companies have been struggling with the unpredictable performance caused by SMT. For instance, Microsoft even disables Hyper-Threading on Xeon servers in many data centers[23]. They hope it can help to guarantee predictable performance for better user experience. However, this will waste tremendous computing resources. Alibaba solicits solutions to perform load balancing among tens of thousands of Intel Xeon servers with Hyper-Threading enabled. This is because unpredictable performance on a single server brought by SMT can make load balancing very difficult in large scale data centers.[1]

In this paper, we ask the following question: ***Can we obtain guaranteed performance of a high-priority workload on an SMT core, meanwhile achieving reasonable overall throughput?*** Specifically, multiple workloads are allowed to co-run on an SMT core, but at least one workload with higher priority such as a latency-sensitive workload should strictly satisfy its performance target. This target can be described as, for example, "guaranteeing 90% of the performance of the workload's solo execution".

We introduce QoSMT, a novel hardware mechanism that is able to guarantee real-time performance requirements for higher priority workloads under SMT-enabled environments without profiling in advance. The key idea is to quantitatively measure SMT-induced performance degradation at run time, locate critical interference caused by lower priority workloads, and make up the performance loss by dynamic resource adjustment. To understand this, consider multiple workloads co-running on an SMT core and a high-priority workload among them* such as web search. The high-priority workload requires achieving at least 90% of the performance of its solo mode(the workload is executed alone on a processor)in term of IPC (instructions per cycle). To this end, we divide the execution into a series of epoches each of which contains tens of thousands cycles. During each epoch, QoSMT will first predict the execution time of the workload's solo mode (T_{solo}) online, calculate the performance loss between the measured execution time (T_{share}) and T_{solo}. If $\frac{T_{solo}}{T_{share}} < 0.9$, an interference detector will identify the microarchitecture resource contributing most to the performance loss. At last, to eliminate performance interference, a controlling unit will dynamically adjust the resource allocation until the performance requirement is satisfied. To realize this, we need to address these challenges:

(i) How to identify critical in-core resources causing performance loss?

There are many microarchitecture resources shared in an SMT core including IQ, ROB, LSQ, instruction L1 cache, data L1 cache, L2 cache and so forth. As shown in Figure 2, every resource could suffer severe contention. At a high level, we define events of severe interference for each resource, collect stall cycles caused by these events and rank resources according to stall cycles as well as their criticality (see 2.2).

(ii) How to precisely quantify a workload's performance loss due to SMT?

When multiple workloads are co-running on an SMT core, it is almost impossible using traditional performance counters to obtain a workload's performance in the solo mode because it is hard to decouple the SMT induced interference from these performance counters. However, performance in solo mode is necessary for calculating performance loss. To address this challenge, we propose a shadow solo-cycle accounting (SSCA) methodology, which monitors all shared in-core resources on the fly and counts the stall cycles caused by other workloads (see 4.2).

(iii) How to perform timely resource adjustment to meet a workload's performace requirements?

Hardware-software codesign is needed to address this issue: First, we leverage a mechanism similar to Intel's model-specific register(MSR) [19] to allow workloads to convey their performance requirements to the underlying hardware. Second, we add control logic to shared resources to enable dynamic resource adjustment. Finally, we design an online algorithm, which takes IPC_{solo}, IPC_{share} and status of critical resources as inputs and then tells how many resources should be adjusted. Thus, these operations form a closed-loop controlling mechanism that consists of monitoring, decision and adjustment. (see §4.3)

To show the feasibility of our design, we have implemented QoSMT on GEM5 simulator [3]. Experimental results on SPECCPU 2006 show that QoSMT is able to achieve performance control for different performance targets (i.e., 85%, 90% and 95% of IPC_{solo}) with an average error of 1.4%, 0.5% and 3.6%.

To summarize, we make the following contributions:

- We demonstrate a thorough analysis of interference induced by SMT from micro-architecture level and reveal two findings that provide important insights about supporting performance guarantee on SMT processors.
- We propose a methodology that enables precise performance control with high utilization, i.e. QoSMT which leverages a shadow solo-cycle accounting framework and closed-loop controlling algorithms.
- We implement a prototype of QoSMT on GEM5 simulator. Through comprehensive experiments, we demonstrate the effectiveness of QoSMT for guaranteeing specific performance target for a given workload while improving SMT resource utilization.

2 BACKGROUND

In this section, we briefly introduce pros and cons of SMT, and then illustrate resources causing contention on SMT. Finally, we present current techniques of eliminating contention adopted by three commodity processors, i.e., Intel's SkyLake, IBM's POWER8 and AMD's Zen.

2.1 SMT's Pros and Cons

To improve processors' throughput, Tullsen et al. proposed simultaneous multithreading (SMT) that allows multiple logical threads to run on a single physical processor to better utilize pipeline resources. In modern commodity multicore processors, each physical core supports two to eight logical threads. Intel and AMD's processors[2] usually have two logical threads while IBM's POWER8 [29]supports eight logical threads.

*It is worth noting that in practice usually only one high-priority workload is scheduled on an SMT core.

Many studies demonstrate the efficacy of SMT. For instance, in light of Madonna et.al's evaluation [20]on IBM POWER8 processors with SMT enabled, running eight instances of 433.milc from SPECCPU2006 on four dual-thread physical cores can approach the performance of running them on eight different physical cores.

However, resource sharing within a physical core may cause performance degradation for logical threads. As illustrated in previous literature [33], a logical thread may suffer performance degradation by up to 70% when its demanding resources are occupied by other threads.

2.2 Resource Contention on SMT

In general, there are many shared resources in SMT processors, including: **fetch unit, instruction queue (IQ), reorder buffer (ROB), load/store queue (LSQ), L1 I-cache/D-cache, and L2 cache and so forth**. A shared resource may encounter a "stall event" due to resource contention, which will prevent instructions from being executed in pipelines. To better understand the SMT-induced interference, We divide these shared resources into three categories according to previous studies [11].

Front-end resources: Instruction fetch unit is the major shared front-end resource. Fetch unit works like a time-sharing scheduler that can fetch one or more instructions during each time slice. In SMT processors, all logical threads share these time slices with a scheduling policy such as round-robin policy. Thus, a fetch stall event may happen for one logical thread when time slices are occupied by other threads.

Back-end resources: The shared back-end resources include ROB, IQ, and LSQ. A stall event of back-end resources happens when a load or store instruction suffers a long cache miss and finally causes dispatching stall because of back-end resources getting exhausted without commit-ready instructions.

Cache resources: All threads share the same cache hierarchy. Unfortunately, shared cache contention can result in significant performance degradation. To address this issue, Intel's processors recently support Cache Allocation Technology (CAT) [18] for L3 cache. But L1 and L2 caches shared by multiple threads still suffer severe contention.

2.3 Commodity Processors' Efforts

As illustrated in Table 1, many commodity processors have devoted efforts to alleviating the SMT interference problem.

Intel Skylake and AMD ZEN attempt to guarantee fairness for two logical threads through statically partitioning back-end resources such as ROB and LSQ. However, static back-end resource partitioning is insufficient to completely eliminate interference. To confirm this argument, we conducted experiments on a server with Intel i7-4770 processors that support Hyper-Threading feature [19] and static partitioning. As shown in Figure 1, when co-located with other applications on a physical core, almost all applications suffer from performance loss, by even up to 2.1X (e.g. perlbench).

IBM POWER8 focuses more on controlling front-end resource and adopts an aggressive policy to support differentiated instruction fetch rates to enforce performance guarantee. But the performance target of Power8 is coarse. It only marks a thread as high or low priority, rather than precisely control the performance like achieving "90% of the performance of solo mode".

Table 1: Commodity processors' efforts to eliminating performance interference caused by SMT.

		Skylake	Power 8	ZEN
Front-end	Fetch	Unknown	Unknown	Unknown
Back-end	ROB	Partition	Shared	Partition
	IQ	Shared	Shared	Partition
	LQ	Partition	Shared	Shared
	SQ	Partition	Shared	Partition
Cache	L1-I	Shared	Shared	Shared
	L1-D	Shared	Shared	Shared
	L2	Shared	Shared	Shared

Since there is little literature about the impact of the three categories of shared resources on performance variations, it seems that processor vendors do not reach a consensus on how to address the challenge. Thus, it is worthwhile to conduct a comprehensive investigation.

3 OBSERVATIONS ON SMT INTERFERENCE

In this section, we present some more in-depth analysis of resources contention by cycle-accurate simulations (see experimental setup in Section 5.1). Specifically, we perform two sets of experiments.

3.1 Static Partitioning

Intel's processors adopt static partitioning for back-end resources, but still exhibit severe performance fluctuations. Since the instruction fetch unit of front-end resources employs a fair round-robin policy, which results in the same effect of static partitioning, a possible reason is because cache resources such as L1/L2 caches are all shared. To test the assumption, we perform experiments to compare three management polices for back-end and cache resources as follows:

(1) **All-Shared**: Both back-end and cache resources are shared by all logical thread.
(2) **Back-end Static Partitioning (Back-end-SP)**: This is an Intel-like policy. The whole back-end resources are equally partitioned for each logical thread, but L1/L2 cache are still shared.
(3) **Complete Static Partitioning (Complete-SP)**: Both back-end and cache resources are equally partitioned for each logical thread.

We implement all these policies on a GEM5 based SMT simulator and run seven workloads of SPECCPU 2006 that are randomly selected on the simulator. For each workload, we run it with other six workload and then measure its averaged IPC and performance variation. As shown in Figure 3, the **All-Shared** policy exhibits worst while the **Complete-SP** policy performs best in terms of performance variation. For instance, the variation of *zeusmp*'s normalized IPC decreases by an order of magnitude, from 0.1 to 0.01. For the **Back-end-SP** policy, its performance variation is non-trivial,

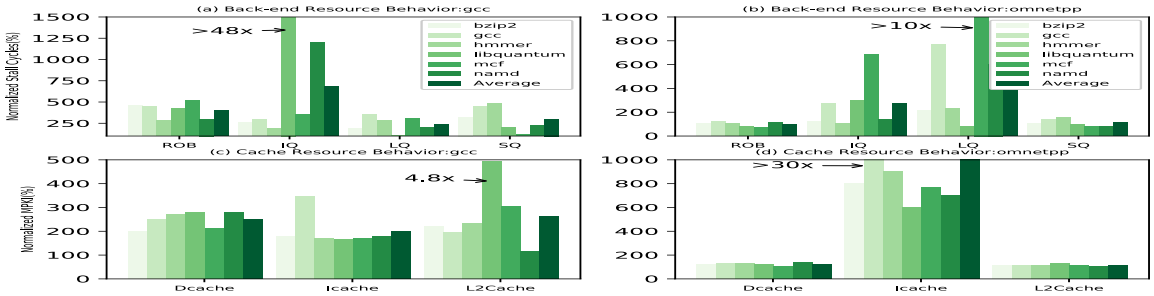

Figure 2: Interference on shared resources: gcc & omnetpp running with other workloads.

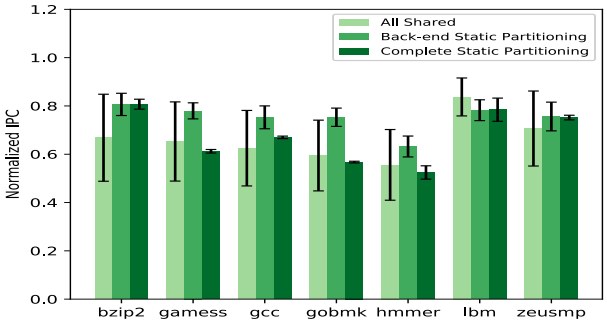

Figure 3: The impact of static partitioning on performance.

which means that L1/L2 cache contention should not be neglected if we want to achieve precise performance control.

However, although the **Complete-SP** policy is beneficial for decreasing performance variation, it is still insufficient for precise performance control due to two reasons. A fixed quota allocation policy does not guarantee a fixed predictable performance for any given workload. Different workloads may suffer from different performance degradation. For instance, even with **Complete-SP** policy that each workload gets the same amount of resources, bzip2's normalized IPC decreases by 20% while the normalized IPC of *gobmk* and *hmmer* decrease by more than 40%. In addition, static partitioning reduces the amount of available resource capacity for each logical thread, which can result in large performance degradation. Take *gobmk* as an example, its normalized IPC decreases by almost 30%, compared to the **Back-end-SP** policy. Therefore, dynamic resource partitioning is necessary to achieve our goal of precise performance control.

3.2 Demand of Dynamic Resource Partitioning

We further investigate which resources may require dynamic partitioning. We co-run *gcc* and *omnetpp* with six other workloads randomly selected from SPECCPU 2006.

Figure 2 shows the behavior of different shared resources. The upper two figures depict normalized stall cycles caused by stall events of back-end resources, while the bottom two figures illustrate the normalized MPKI (miss per kilo-instruction) of cache resources. According to the figures, our findings are as follows:

- ***Finding 1.*** Almost each component suffers from severe interference and potentially becomes a bottleneck.

For example, for *gcc*, the stall cycles of LQ increase modestly by only 3 times while the stall cycles of IQ increase by more than 48 times, indicating that LQ is not a critical resources for gcc. But LQ's stall cycles increase sharply by about 10 times when *omnetpp* co-runs with *mcf*.

- ***Finding 2.*** Even one pair of workloads can result in contention on multiple components.

For example, the workload pair of *omnet* and *hmmer* suffer from the contention of not only ROB but also ICache. The case of *omnet* and *mcf* is even worse, incurring significant contention of many components including IQ, LQ, SQ, ICache and LLC.

These findings suggest that resource requirements of workload mixtures change significantly and it is difficult to meet the resource requirements of each logical thread through pre-fixed static resource partitioning. In order to guarantee the performance of a targeted thread, a flexible dynamic resource partitioning for both back-end and cache resources is needed.

4 DESIGN OF QOSMT

In this paper, we propose QoSMT that aims to guarantee performance of high-priority threads (HPT) while achieving reasonable overall throughput on SMT processors. QoSMT consists of two modules: contention detection module (CDM) and policy enforcement module (PEM). CDM is responsible for collecting and recording contention data on the three categories of shared resources. Meanwhile, PEM periodically takes the contention data as input to perform dynamic resource allocation.

There are three main challenges in implementing the two modules described above: (1) CDM needs to identify the most severely interfered resources; (2) CDM needs to predict performance in solo–mode with SMT enabled; (3) PEM should be able to do efficient dynamic controlling in response to the information gathered above.

Figure 4 shows the overview design of QoSMT. Among all the components, ①,③,⑤ are responsible for both contention detection and policy enforcement; ②,④,⑦ belongs to CDM; ⑥ are paths for information gathering.

Figure 4 also illustrates the QoSMT process of dynamically guaranteeing HPT performance through a real fragment from an evaluation where HPT:astars is co-running with LPT:cactusADM. The whole process is divided into four steps:

Step1: At beginning, the user or administrator specifies a performance target for the HPT by configuring the MSR register.

Step2: CDM keeps identifying various interference events with the help of CMT(cache miss table) and counts the stall cycles of each interference event in the CRT(contention resource table)⑦.

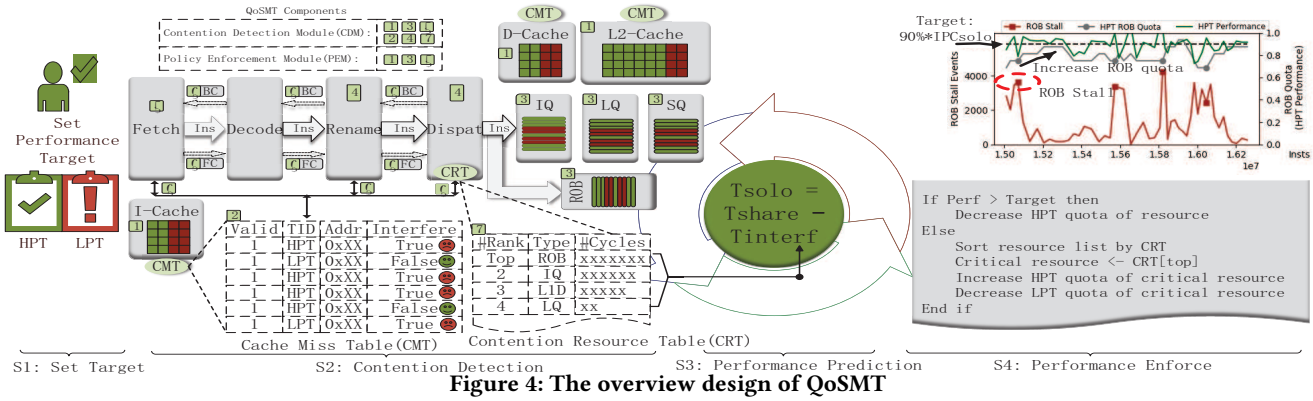

Figure 4: The overview design of QoSMT

Step3: Based on the number of stall cycles counted in CRT in Step2, the HPT's solo performance will be predicted.

Step4: If the current HPT performance fails to reach the target, PEM will conduct dynamic resource allocation operation. PEM will reallocate the resource of TOP rank in CRT from LPT to HPT. For example, t1 in the figure shows that the current performance does not meet the requirements, and ROB is the performance bottleneck. So HPT gets more ROB resources, increasing the performance .

4.1 Critical Resource Detection Mechanism

In order to guide dynamic resources allocation, we first need to detect the true interference events caused by SMT among various of pipeline stall events. And then find out the most severely interfered resources (critical resources). As defined in earlier sections 2.2, we need to detect front-end contention, back-end contention and cache interference. For the sake of logical sequence, we explain the three detection schemes in reverse order.

4.1.1 *Cache Interference Detection.*

The main difficulty in detecting cache interference is:

- How to distinguish whether a HPT cache request miss is caused by low priority thread (LPT)'s eviction or by HPT itself;

To solve the problem, we devise a shadow tag ① mechanism. The shadow tag maintains a private LRU stack for HPT. For each cache access, only the access requests issued by HPT will access shadow tag. When an HPT cache miss occurs, but if it hits in the shadow tag, CDM treats this cache miss as an interference event.

To record the detection results, we design a cache miss table(CMT)②. CMT has three parts, corresponding to I-cache, D-Cache and L2-Cache respectively. Each part has the same number of entries as the number of MSHRs, which is the maximum number of outstanding miss allowed. CMT is synchronized with MSHR allocation and deallocation. Whenever an MSHR is allocated, its corresponding valid bit in CMT is set, the ID of the logical thread that initiates the cache access will be recorded in the TID field. If a shadow tag hits, we also need to set the interference bit. Whenever an MSHR is released, its corresponding valid bit and interference bit in CMT should be cleared.

4.1.2 *Back-end Contention Detection.*

There are two main difficulties in detecting back-end contention:

- How to distinguish whether a back-end stall is caused by LPT occupying shared resources or by HPT itself;
- How to trace the root cause of back-end stall under SMT.

To solve the first problem, we bring the idea of shadow tag to other components, such as shadow ROB, shadow IQ, shadow LQ and shadow SQ ③. We use the shadow queue to emulate HPT's use of the original queue when it would run in solo mode (with the whole physical core monopolized). When the real queue for HPT running in SMT-mode is not full, behaviors of a shadow queue are actually consistent with those of the real queue. When the real queue for HPT is full, the shadow queue's entries continue to be allocated until it reaches the max size of original queue. Once the shadow queue is exhausted, the subsequent stall events should also occur in solo mode, which are caused by HPT itself. As a consequence, by comparing the full state of real queue and shadow queue, we can tell whether SMT induced contention is happening.

Below we will take shadow ROB as an example to illustrate the maintenance of shadow queue. The shadow ROB can be implemented as a counter, and is incremented whenever an HPT instruction is inserted into ROB. When the ROB blocks HPT, CDM keeps incrementing the shadow ROB counter. When the real ROB is full but the shadow ROB is not yet, the ROB contention is happening. When both of the two queues reach the max size, we will not increment the stall cycles in CRT. Shadow ROB counter is decremented since an HPT instruction is retired from ROB. The other three shadow queue counters are maintained similarly.

Upon a long-latency load miss, the processor back-end will stall because of the ROB, IQ, or LSQ getting exhausted[14]. To trace the root cause of back-end stall, shadow queues need to cooperate with CMT. Figure 5 illustrates how to use CMT and shadow queue to detect contention on back-end resources by taking IQ as an example. When HPT is to dispatch instructions and IQ is full, it will read CMT to check whether this blocking is caused by cache interference. Then it will see whether there are outstanding misses or floating point instructions in computation. If neither of situations happens, HPT should not be blocked. So this stall is regarded as a contention in IQ. If there exists such a long latency instruction, we check the shadow IQ counter for further decision. If the shadow IQ counter < maxIQsize, which suggests that the IQ Full event is caused by another thread's contention on IQ, this incident should be regarded as a SMT contention. If the shadow IQ counter = maxIQsize, it is the thread itself filled up the IQ, not to blame SMT.

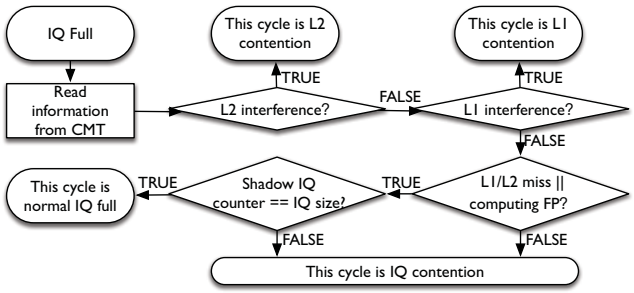

Figure 5: Back-end contention detection.

Table 2: Conditions of Fetch Contention

Status of HPT	Contention Conditions
I-Cache Miss by LPT	I-Cache Contention
I-Cache Miss by HPT	Not Contention
Back-end Stall by LPT	Back-end Contention
Back-end Stall by HPT	Not Contention
Yield	Front-end Contention

4.1.3 *Front-end Contention Detection.*

To detect contention at front end ⑤, we classify the scenarios in which fetch can not supply instructions into three categories. (1) I-Cache miss, (2) LPT preempts HPT's timeslice, (3) Stalls at back-end resources.

In Table 2 we show our contention determination scheme based on HPT's status. Next we give an explanation of Table 2. Since whether LPT getting a fetch opportunity interferes HPT depends on the status of HPT, the first column of matrix shows HPT's status. Back-end stall by LPT means that HPT has back-end stalls, which is caused by LPT. Yield means that in this cycle HPT has neither I-cache miss nor back-end stall, but LPT takes the fetch opportunity. Other states' name explains themselves. The second column shows whether this cycle should be judged as a contention and what kind of contention it is when HPT does take the fetch opportunity. For example, the third row tells that when HPT's back-end is blocked by LPT, this cycle in the fetch phase should be regarded as Back-end Contention.

4.2 Performance Prediction Mechanism

To ensure precise performance control, we need to precisely predict performance in solo-mode as a baseline.

4.2.1 *Gathering Contention Information.*

In order to quantify the contention on each shared resource, we design a Critical Resource Table (CRT) to record contention cycles for each stall event by three steps. First, each CDM distributed at each pipeline stage will separately generate contention judgement signals by checking whether HPT is stalled due to contention based on the mechanism mentioned above. Then the signals will be forwarded to the dispatch stage. Finally, the values of the corresponding contention cycle entries of the CRT are updated according to the contention judgement signals at dispatch stage.

To gather contention judgement signals detected in distributed CDMs, we add some signals along the paths marked by ⑥. In particular, FC (front-end contention) means that contention is detected at fetch/rename stage, and passed to later stages. BC (back-end contention) means that contention is detected at rename/dispatch stage, and passed to earlier stages. Fetch stage obtains back-end contention information by this path. Signals from CMT to pipeline stages convey cache interference and miss information. They are used in 4.1.2 and 4.1.3.

For example, when the CDM at rename stage detects a ROB stall event of HPT caused by LPT, it will forward the judgement signals to the dispatch stage and will increment the contention cycle value corresponding to ROB contention entry in the CRT.

There are two reasons for this design: (1) The PEM needs to use contention cycles as input. Since contention information is distributed at serval stages, a centralized CRT can avoid long wire delay of reading distributed contention values; (2) Because of the overlap of SMT-induced contention and stalling by itself, performing contention detection at one stage of pipeline could lead to inaccurate result that the amount of collected contention cycles may diverge from the actual amount caused by LPT. In addition, updating CRT at the dispatch stage can achieve a good tradeoff between design complexity and statistical accuracy. The earlier contention detection is performed in the pipeline, the less information and poorer statistical accuracy will be obtained. The later detection is performed in the pipeline, the more complicated the design will be.

4.2.2 *Predicting Performance in Solo-Mode.*

Referring to prior work [12], we design shadow solo-cycle accounting (SSCA) approach to estimate workloads' execution time in solo mode by $T_{solo} = T_{share} - T_{interf}$, where T_{share} is the execution time in SMT mode and the interference time, T_{interf}, is the sum of the contention stall cycles stored in the CRT.

ALGORITHM 1: Dynamic Controlling Policy

Input:
T_{solo}: Estimated solo execution time
T_{share}: Real execution time in SMT processor
$CurPerf$: Current HPT performance behavior
$ResourceList$: Enumeration of shared resources
$ContentionList$: Contentions cycles of resources in CRT
$CriticalResouces$: Resources affect HPT most
$ExpectedTarget$: User's expected target
Output: $CurPerf = \frac{T_{solo}}{T_{share}}$

```
if CurPerf > ExpectedTarget then
    for resource ∈ ResourceList do
        Decrease HPT quota of resource;
        Increase LPT quota of resource;
    end
else
    Argsort ResourceList by ContentionList;
    CriticalResouces ← ResourceList[top : tail];
    for resource ∈ CriticalResouces do
        Increase HPT quota of resource;
        Decrease LPT quota of resource;
    end
end
```

4.3 Dynamic Controlling

QoSMT enables users to write the Expected Target Register, which is similar to MSR in Intel processors, to convey performance target requirements. To achieve the expected target, the PEM takes predicted T_{solo} and detected critical resources as input, and leverages resource allocation modules in fetch unit ⑤, ROB, IQ, LSQ ③ and caches ①, to perform dynamic resource adjustment. Next, we will first introduce the algorithm of the decision and controlling module, and then describe the grain of resource adjustment.

Algorithm 1 shows the procedure of dynamic controlling. We first calculate current performance target of HPT using solo execution time estimated in 4.2. If current performance is larger than expectation, then we allocate all kinds of resources from HPT to LPT to improve throughput. If not satisfied, the PEM first sorts *ResourceList* in CRT by corresponding contention cycles in decreasing order. Then PEM chooses the most critical resources from the sorted list, and reallocates resources accordingly.

The quantum of fetch scheduling is one cycle out of 16 cycles. When HPT needs more fetch quota, one more cycle is allocated from LPT to HPT, or vice versa. Both threads have at least one cycle out of 16 cycles. Assume that *size* is the capacity of ROB/IQ/LQ/SQ, each thread occupies at least $size/16$ entries and $size/16$ is the reallocation unit. For caches, the allocation unit is cache associativity, and each thread occupies at least one way. Initially, all these resources are equally partitioned.

5 EVALUATION

5.1 Experiment setup

We implement QoSMT on GEM5 simulator. Considering the support of SMT, we choose Alpha ISA. We use 24 benchmarks from SPEC2006 [17] as workloads. Due to compile or runtime errors, another 5 benchmarks are not used. To extract typical behaviors of workloads, we use SimPoint [25] to acquire checkpoints for each benchmark. We run different benchmarks from their checkpoints on SMT to achieve workload co-location.

To obtain typical workload pairs among these benchmarks, we refer to the balanced random method [32] to select 48 pairs of benchmarks for experiments. Then we run QoSMT with 85%, 90% and 95% as users' expected target. But due to space limitation, we show only 24 pairs without loss of generality. They are selected by first ranking the 48 pairs according to HPT performance, then sampled with a fixed step of 2. They are used through the remaining parts of the evaluation.

Table 3 shows the GEM5 configurations. Since both Intel and AMD's processors have two logical threads, we also primarily investigate the configuration of two threads. We assume that fetch buffers are private with round-robin policy by default, but the functional units are shared.

For convenience, we use some abbreviations to refer all the policies. FR means fetch round-robin. FD means fetch dynamically. BS means back-end sharing. BP means back-end static partitioning. CS means cache sharing. CP means cache static partitioning. Cazorla is a state-of-the-art mechanism presented by Cazorla,e.g.,[6]. QoSMT, our methodology, will dynamically adjust all resources.

Table 3: Configuration List

resource	configuration
Width	8
Fetch buffer size	64 per thread
ROB entries	224
IQ size	96
LQ entries	72
LQ entries	56
Physical Int registers	256
Physical FP registers	200
L1 D-Cache	32KB, 4-way, LRU, 8 MSHRs
L1 I-Cache	32KB, 4-way, LRU, 8 MSHRs
L2 Cache	2MB, 8-way, LRU, 32 MSHRs
Cache latencies	L1(4), L2(40)

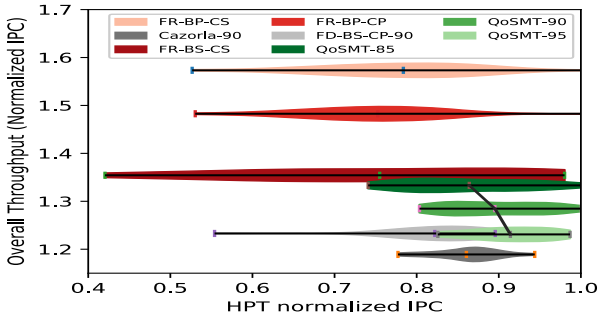

Figure 6: Target IPC-Throughput

5.2 Evaluation of Different Performance Target

To verify QoSMT's capability of guaranteeing performance target, we run the selected pairs of workloads with HPT performance targets of 85%, 90% and 95%, respectively. Figure 6 shows the distributions of HPT normalized IPC(IPCshare/IPCreal)[†] from the selected pairs with different policies. The segment length of each policy on X-axis implies the variation of the performance of HPT. The shorter the segment is, the more predictable performance the policy provides. The centroid of each segment indicates the average performance of HPT. FR-BS-CS is a policy that lets all resources suffer from full contention, which performs the worst predictability for HPT. FR-BP-CS simulates the policy adopted by Intel, but it may suffer from interference in cache. FR-BP-CP supports cache partitioning based on FR-BP-CS. Its distribution is a little more concentrated than FR-BP-CS, yielding a little bit better predictability. However, the centroid moves to the left, suggesting performance loss. It is worth noting that all of these conclusions are consistent with the observations in §3. The centroid of three green segments (QoSMT-85, QoSMT-90 and QoSMT-95) show that QoSMT can let HPT finally achieve average normalized IPC of 86.4%, 89.5%, 91.4% respectively. The segment length of Cazorla-90 is almost the same as that of QoSMT-90, indicating that they have similar performance stability. However, the average normalized IPC of Cazorla-90 is less than 90%, indicating that the ability of Cazorla-90 to satisfied target is less than QoSMT's. From this view, QoSMT provides best performance predictability than all other policies. This is because

[†]IPCreal is the real IPC of HPT running in solo mode that we get it offline. IPCshare is the IPC of HPT running in QoSMT

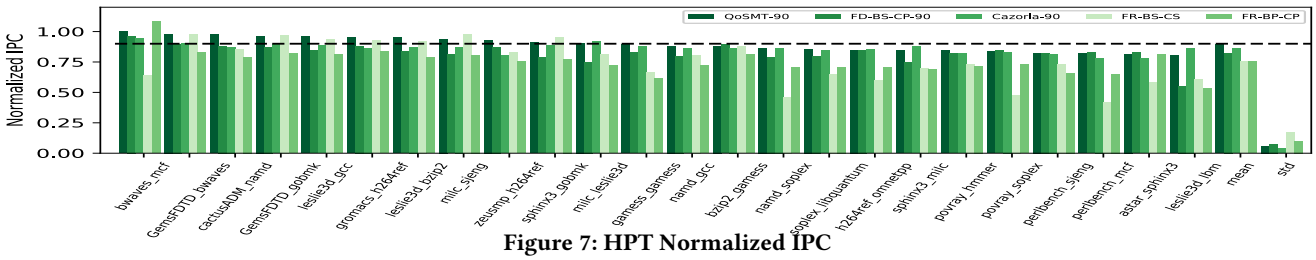

Figure 7: HPT Normalized IPC

QoSMT can dynamically control all types of resources, protecting HPT from contention on any resources.

FD-BS-CP-90 is very interesting. It simulates the policy of online monitoring and adjustment with performance target 90% on an IBM POWER8 machine. But it only obtain 82.2% average normalized IPC for HPT, even being outperformed by QoSMT-85. This is because FD-BS-CP-90 does not deal with back-end resource interference. It also has less predictability (longer segment) than QoSMT. These suggest that back-end resource control is necessary to performance guarantee. We will have further discussions about the details in §5.5.

To show that QoSMT does not hurt too much throughput with performance guarantee, we calculate the overall normalized IPC on Y-axis in Figure 6, which is the sum of normalized IPC of both HPT and LPT. FR-BP-CS provides the highest overall normalized IPC because of its balanced resource allocation. But as expected, it does not provide an adequate ability of guarantee performance target. Compared with policies that focus on guaranteeing performance, We can see that QoSMT-95 achieves similar throughput to FD-BS-CP-90. QoSMT-90 provides a better throughput than Cazorla-90, where LPT can still get average 40% normalized IPC. Because Cazorla-90 need a sampling phase dedicated for running in isolation the HPT, which causes performance loss for LPT. In addition, the coarse-grain resources partition of Cazorla-90 will also limit the LPT's performance. As for QoSMT, each operation of resource allocation is based on fine-grain statistics of critical resource. These suggest that QoSMT can achieve reasonable overall throughput through a timely performance prediction and a effective dynamic controlling mechanism, compared with state-of-the-art policies.

Note that our design is fully compatible with other policies, because QoSMT provides a tuning mechanism between overall throughput and HPT's performance. This is achieved by configuring the parameters of dynamic controlling mechanism introduced in §4.3. For example, if overall throughput is preferred, one can configure QoSMT similar to FR-BP-CS.

5.3 Further Analysis on 24 Pairs

To further understand the difference among these policies, Figure 7 lists the HPT performance of all the selected 24 pairs with the expected performance target of 90%. Each pair is named as "X_Y", where "X" is HPT and "Y" is LPT. All pairs are sorted by the HPT performance from left to right. Compared with Figure 6, Figure 7 implies two more conclusions.

First, QoSMT outperforms all other policies for most pairs. This is because QoSMT provides dynamic control over all resources. The performance of HPT varies between 85% - 95% with QoSMT.

Second, given an HPT, QoSMT can protect it from being interfered by any other workloads. This can be concluded from the observation that pairs with the same HPT are not far from each other in Figure 7, since pairs are already sorted by the HPT performance from left to right. For example, running with bwaves or gobmk, GemsFDTD still keeps similar performance with QoSMT.

There is a special case which bwaves achieves a performance greater than 100%. We find that there are large amount of floating point branches in bwaves. They are not only hard to predict whether it is taken or not, but also highly depend on prior floating point instructions[15]. Therefore, floating point branches in bwaves will be more likely to stay at the head of ROB. Then, the more instructions in ROB, the higher penalty a branch misprediction results in. Hence bwaves performs better when ROB is smaller.

5.4 Effect of Prediction Mechanism

As shown in Figure 6 and Figure 7, some HPT workloads fail to achieve the performance target. The main reason is that there is a modeling error in the prediction mechanism. To demonstrate the effect of prediction accuracy on achieving target, we show the relationship of "prediction error" versus "expected target gap" of the 24 pairs of QoSMT-90 in Figure 8, where the prediction error and expected target gap are calculated as IPCsolo_predicted - IPCsolo_real and IPCshare - 90%*IPCsolo_real respectively. From Figure 8, the slope of QoSMT-90 is 1, which means expected target gap is equal to its prediction error. The fewer the prediction error is, the fewer the expected target gap will be.

We analyze the two sources of prediction error.

- From the aspect of mirco-architecture, a program's behavior under SMT mode is totally different from that under solo mode. To obtain the cache behavior under solo mode as much precise as possible, QoSMT already uses shadow tag. However, it is very difficult to obtain the pipeline behavior under solo mode, due to the complexity of resource contention. Therefore, the shadow queue mechanism will still introduce certain amount of inaccuracy.
- The prediction method of QoSMT is inspired by PTA. PTA uses MLP correction to achieve higher accuracy [12]. However, we can not get an application's MLP without offline profiling, so this correctness mechanism is not adopted in QoSMT. This may be another source of inaccuracy of prediction.

As for QoSMT-95, because there is a minimum quota for both threads during resources allocation, QoSMT did not completely guarantee the target in some workloads. For example, L1-D cache have 4 ways, and the minimum quota is one way. In this case, at most 3 ways can be allocated to HPT, which causes performance

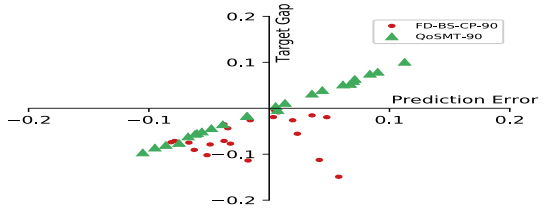

Figure 8: Relationship between prediction error and achieved performance

loss. But this it is not common, and occurs where extremely high performance is expected.

5.5 Effect of Dynamic Controlling

To have a better understanding about the need of dynamic controlling on both front-end and back-end resources, we plot FD-BS-CP-90 performance distribution in Figure 8, which only adopts fetch-rate reallocation. If a policy is effective, the expected target gap should be equal to its prediction error. As shown in Figure 8, the distribution of QoSMT-90 conforms to our hypothesis, while it is not the case for FD-BS-CP-90. That is, there are some situations, where FD-BS-CP-90 knows that the expected target has not been satisfied, but it does not have enough ability to guarantee the target performance. This implies the need of controlling back-end resources, which can help to enforce the performance requirements more effectively.

5.6 Evaluation of Response Time

The latency-sensitive application requires quick response time for a performance guarantee policy. To demonstrate the effect of QoSMT dynamic control mechanism on response time, we did an evaluation of response time on the above 24 pairs. Figure 9 shows how long it takes QoSMT to guarantee performance to meet the expected target, where the expected target is 90% IPCsolo. As Figure 9 shows, QoSMT-90 has a significantly faster response time than Cazorla-90. For 90% of pairs, QoSMT only needs to spend less than 450000 cycles to meet the performance target. Taking 2Ghz cpu frequency as an example, the response time of 90% is 225us. For Cazorla-90, Its response time of 90% is 325us. Especially for cache-sensitive workloads, the Cazorla-90 has almost twice the QoSMT response time. We think there are two reasons for this result:1> In order to predict solo-mode performance, Cazorla-90 needs to sample HPT performance by exclusively running, and the sampling phase needs to maintain a certain period to filter the impact of cache interference on sampling accuracy. Thus it inevitably leads to a delay in response time. By using the shadow solo-cycle accounting methodology, QoSMT can timely get the predicted IPC without the delay of waiting for the sampling phase. 2> Cazorla adopts a coarse-grained resource allocation method, like Fetch Rate and Issue bandwidth. It doesn't adjust for critical resources. Through the combination of CMT and CRT, QoSMT can directly locate critical resources in a fine-grained manner and find performance bottlenecks. For example, for a cache sensitive workload, it would not be efficient to allocate more issue bandwidth, while CMT could timely calculate cache interferences, which would prompt QoSMT to allocate more cache resources, thus providing a faster response time.

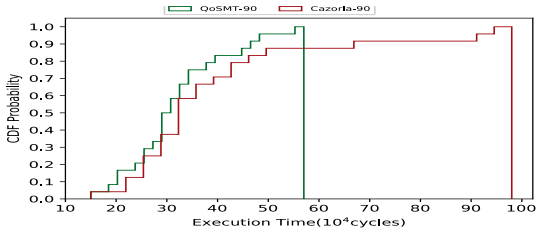

Figure 9: Response Time

5.7 Epoch length Choosing

Since QoSMT will adjust the resource allocation periodically, we need to address the problem how to choose a suitable epoch length to make QoSMT more effective. We have observed that, if the length is too short, QoSMT will not catch enough event to make a good decision about resource allocation. On the other hand, if the length is too long, QoSMT will not make timely decision to the contention variation. We choose 10 pairs of benchmarks with 90% of performance target, trying different epoch length among {1000, 2000, 5000, 10000, 20000, 40000} cycles. And 10000 cycles and 20000 cycles are better than others from the view of overall performance. Users can choose it according to actual needs. If the user is more concerned about latency, 10000 cycles is preferred. If the user want to achieve lower dynamic power, 20000 cycles is a better choice.

5.8 Overhead

There are three major concerns about adding QoSMT design into the architecture of general SMT processors:

- How many extra latency is introduced?
- How many hardware resources are required ?
- How much extra power consumption is introduced?

Latency. QoSMT does not modify the basic architecture of the pipeline logic. Based on our evaluation on GEM5, the added components do not introduce extra latency at all. As Figure 4 shows, CRT and CMT are located on non-critical path. For CDM, We need signals to gather contention information detected in distributed CDMs, We add these signals along the pipe line with refer to data forwarding signals. In addition, the memory requests will be sent to shadow tag and cache tag array simultaneously, they will individually work without any impact on each other.

Resources. Our design introduces some new components, but most of them are very cheap. The only costly component is the shadow tag. In order to ensure the accuracy of interference detecting, The size of shadow tags are as same as each level cache tag. Other components' overhead mainly depend on the number of entries in each table:

(1) The CMT requires n × r bits, where n is the number of bits per entry, r is the the maximum number of outstanding miss allowed. For our design, each entry includes 1 bit valid signal, 1 bit interference judgement signal, and 18 bit TID. For L1 cache, the maximum number of outstanding miss allowed is 8, for L2 cache, the number is 32, which is (1 + 1 + 18) × (8 + 8 + 32) = 960 bits for CMT;
(2) The CRT requires n × r bits, where n is the number of bits per resources contention stall counter and r is the number of contention resource, which is 64 × 8 = 512 bits;

(3) There are four shadow queue counters for ROB, IQ, LQ, and SQ. The shadow queue counters need $\log_2 ROB$ + $\log_2 IQ$ + $\log_2 LSQ$, which is 23 bits in our design totally.

We evaluate the area overhead of QoSMT using McPAT[22]. The area of the baseline chip based on ALPHA31264 is 47 mm^2 under a 40nm process technology. The area overhead of L2 Shadowtag, L1 Shadowtag, CMT, and CRT in terms of percent (%) are 0.7%, 0.02%, 0.0047%, 0.0045% respectively.

Power. We use Wattch[5] to evaluate the power consumption. The most power consumption component is L2 cache shadow tag, which is 1.8 W , while other components power are 1.5W in total. The sum power consumption accounts for 3.6% of total processors power, where the total power of processor is 90 W.

5.9 Equal-Silicon-Area Performance

The area overhead of the shadow tag can be replaced with a 46KB cache. To evaluate normalized performance to an equal-silicon-area non-QoSMT design, we disable shadow tag mechanism and increase the L1 Dcache capacity from 32KB to 128KB. The average normalized IPC of equal-silicon-area non-QoSMT designs with 90% target is 86%, which is worse than QoSMT90. To further evaluate the benefit of shadow tag on eliminating cache interference, we choose a cache sensitive workload hmmer in SPEC2006. When co-runs with a memory-intensive workload mcf, the normalized IPC of hmmer drops to 75%. Therefore, shadow tag combining with dynamic cache partitioning is a key component to eliminate cache interference. The area overhead is worth it.

6 RELATED WORK

6.1 Industry Design

As shown in Table 1, IBM POWER processors adopts more aggressive QoS support than Intel and AMD. Thus, we focus on IBM's design. Generally, IBM introduced two-level control mechanisms [4] since POWER5 to enable software to adjust instruction fetch rates for specific threads. [24] evaluates the effect of software controlling approach on POWER5 and POWER6. Their results show that the effect of software controlling is unstable across various microbenchmarks. While max priority can achieve 90%-100% HPT IPC, the co-located LPT only receives as low as 2% - 9% of performance in solo mode, indicating huge throughput/utilization loss.

6.2 Resources Contention and Modeling

Raasch et al. [26] revealed that industry-favored simple static partitioning policies are able to achieve good overall throughput, but they did not investigate the impact of static partitioning on guaranteeing performance of specific threads.

Cakarevic et al. [31] analyzed the impact of shared resources of UltraSPARC T2 that is a somewhat different fine-grain multithreading processor. But their results are unapplicable to mainstream SMT processors because UltraSPARC T2's architecture is pretty different from Intel and AMD's design.

Recent work SMiTe [33] uses a set of carefully designed microbenchmarks to perform offline sensitive analysis on various shared resources of SMT cores, and then proposes a decent job scheduler to avoid performance degradation of HPT.

6.3 Software Based Polices

Eyerman et al. [13] proposed a job scheduler based on sampling mechanism along with hardware modification to achieve better throughput on SMT processors. Feliu et al. proposed an improved design [16] without hardware modification, but still focused on overall throughput rather than performance guarantee.

6.4 Hardware based policies

The closest work to QoSMT is the design proposed in [6, 8]. For their work, whether the prediction IPC is accurate depends on the severity of the interference of LPT at the sampling stage. They use warmup about 5K instructions procedure to filter the interference, however, it will cost millions of instructions to release L2 cache interference. We did evaluate their design with memory intensive benchmark as LPT, the result illustrated that the average performance decreased. In addition, QoSMT does not need a time period of tens of thousands of cycles for sampling, hence is more robust against frequent program phase changes. Transparent thread also aims to maximize HPT's performance with reasonable overall throughput [11]. Unlike QoSMT, it does not support precise performance control based on user-defined target. There are many studies [7][30][9][10][27] providing solutions on improving overall SMT throughput and fairness, but they did not take performance control into account. Eyerman et al. [12] proposed the per-thread cycle accounting (PTA) mechanism that is able to estimate a workload's solo performance in a co-running mode on SMT processors. QoSMT leverages this concept to perform performance prediction.

7 CONCLUSION

This paper presents QoSMT methodology, which enables precise performance guarantee for given performance targets on SMT core. Based on our investigation on the behavior of interference induced by SMT, we find that it is difficult to meet the variable resource requirements for a logical thread through static resource allocation. As a consequence, we design a critical in-core resources identification mechanism and a solo IPC predictor to direct a closed-loop mechanism to dynamically allocate the resources for target thread. We implemented a QoSMT prototype based on GEM5 simulator. As illustrated in our experiment, compared with the state-of-the-art, QoSMT are able to enforce a precise performance control for specific IPC targets.

8 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their valuable feedbacks and suggestions. We thank our group members, Wenbin Lv, Zhigang Liu, Tianni Xu, Zhiyuan Yan for their help on this work. Especially thank Lan Xiao for her silently concern. This work was supported in part by National Key R&D Program of China (2016YFB1000201), and the National Natural Science Foundation of China (Grant No. 61420106013 and 61702480), and Primary Research & Development Plan of Shaanxi Province(2019TSLGY08-03) and Youth Innovation Promotion Association of Chinese Academy of Sciences (2013073).

REFERENCES

[1] Alibaba. 2018. *Alibaba Innovative Research.* https://102.alibaba.com/fund/proposalAbout.htm.

[2] AMD. 2016. The Zen Core Architecture AMD. http://www.amd.com/en-gb/innovations/software-technologies/zen-cpu.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[4] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C. Y. Cher, and M. Valero. 2008. Software-Controlled Priority Characterization of POWER5 Processor. In *2008 International Symposium on Computer Architecture*. 415–426. https://doi.org/10.1109/ISCA.2008.8

[5] David M. Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture (ISCA 2000), June 10-14, 2000, Vancouver, BC, Canada*. 83–94.

[6] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. 2006. Predictable Performance in SMT Processors: Synergy Between the OS and SMTs. *IEEE Trans. Comput.* 55, 7 (July 2006), 785–799. https://doi.org/10.1109/TC.2006.108

[7] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. 2004. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 171–182. https://doi.org/10.1109/MICRO.2004.17

[8] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernandez. 2004. QoS for high-performance SMT processors in embedded systems. *IEEE Micro* 24, 4 (July 2004), 24–31. https://doi.org/10.1109/MM.2004.37

[9] Seungryul Choi and Donald Yeung. 2006. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. *SIGARCH Comput. Archit. News* 34, 2 (May 2006), 239–251. https://doi.org/10.1145/1150019.1136507

[10] Seungryul Choi and Donald Yeung. 2009. Hill-climbing SMT Processor Resource Distribution. *ACM Trans. Comput. Syst.* 27, 1, Article 1 (Feb. 2009), 47 pages. https://doi.org/10.1145/1482619.1482620

[11] Gautham K. Dorai and Donald Yeung. 2002. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 30–. http://dl.acm.org/citation.cfm?id=645989.674324

[12] Stijn Eyerman and Lieven Eeckhout. 2009. Per-thread Cycle Accounting in SMT Processors. *SIGPLAN Not.* 44, 3 (March 2009), 133–144. https://doi.org/10.1145/1508284.1508260

[13] Stijn Eyerman and Lieven Eeckhout. 2010. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. *SIGPLAN Not.* 45, 3 (March 2010), 91–102. https://doi.org/10.1145/1735971.1736033

[14] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A performance counter architecture for computing accurate CPI components. (2006), 175–184. https://doi.org/10.1145/1168857.1168880

[15] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. 2006. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2006, March 19-21, 2006, Austin, Texas, USA, Proceedings*. 48–58.

[16] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. 2016. Symbiotic job scheduling on the IBM POWER8. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 669–680. https://doi.org/10.1109/HPCA.2016.7446103

[17] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[18] INTEL. 2005. *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family.* https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[19] INTEL. 2016. 64-ia-32-architectures-software-developer-vol-3b-part-2-manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[20] S. Jasmine Madonna, Satish Kumar Sadasivam, and Prathiba Kumar. 2015. *Bandwidth-Aware Resource Optimization for SMT Processors.* Springer International Publishing, Cham, 49–59.

[21] D. Koufaty and D. T. Marr. 2003. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro* 23, 2 (March 2003), 56–65. https://doi.org/10.1109/MM.2003.1196115

[22] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*. 469–480. https://doi.org/10.1145/1669112.1669172

[23] Microsoft. 2017. *Azure SQL Database.* https://azure.microsoft.com/en-us/pricing/details/sql-database/elastic/.

[24] A. Morari, C. Boneti, F. J. Cazorla, R. Gioiosa, C. Y. Cher, A. Buyuktosunoglu, P. Bose, and M. Valero. 2013. SMT Malleability in IBM POWER5 and POWER6 Processors. *IEEE Trans. Comput.* 62, 4 (April 2013), 813–826. https://doi.org/10.1109/TC.2012.34

[25] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (June 2003), 318–319. https://doi.org/10.1145/885651.781076

[26] Steven E. Raasch and Steven K. Reinhardt. 2003. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*. IEEE Computer Society, Washington, DC, USA, 15–. http://dl.acm.org/citation.cfm?id=942806.943858

[27] Joseph Sharkey, Deniz Balkan, and Dmitry Ponomarev. 2006. Adaptive Reorder Buffers for SMT Processors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*. ACM, New York, NY, USA, 244–253. https://doi.org/10.1145/1152154.1152192

[28] Balaram Sinharoy, Ronald N Kalla, Joel M Tendler, Richard J Eickemeyer, and Jody B Joyner. 2005. POWER5 system microarchitecture. *IBM journal of research and development* 49, 4.5 (2005), 505–521.

[29] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.

[30] Hans Vandierendonck and André Seznec. 2011. Managing SMT Resource Usage Through Speculative Instruction Window Weighting. *ACM Trans. Archit. Code Optim.* 8, 3, Article 12 (Oct. 2011), 20 pages. https://doi.org/10.1145/2019608.2019611

[31] Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2009. Characterizing the Resource-sharing Levels in the UltraSPARC T2 Processor. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 481–492. https://doi.org/10.1145/1669112.1669173

[32] Ricardo A Velásquez, Pierre Michaud, and André Seznec. 2013. Selecting benchmark combinations for the evaluation of multicore throughput. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 173–182.

[33] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 406–418. https://doi.org/10.1109/MICRO.2014.53