Preemptive Multi-Queue Fair Queuing

Yong Zhao[1], Kun Suo[1], Xiaofeng Wu[1], Jia Rao[1], Song Wu[2] and Hai Jin[2]
[1]The University of Texas at Arlington, USA
[2]National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology
Email: {yong.zhao, kun.suo, xiaofeng.wu jia.rao}@uta.edu, {wusong, hjin}@hust.edu.cn

ABSTRACT

Fair queuing (FQ) algorithms have been widely adopted in computer systems to share resources among multiple users. Modern operating systems and hypervisors use variants of FQ algorithms to implement the critical OS resource management – the thread scheduler. While the existing FQ algorithms enforce fair CPU allocation on a per-core basis, there lacks an algorithm to fairly allocate CPU on multiple cores. This common deficiency in state-of-the-art multicore schedulers causes unfair CPU allocations to parallel programs using blocking synchronization, leading to severe performance degradation. Parallel threads that frequently block due to synchronization exhibit *deceptive idleness* and are penalized by the thread scheduler. To this end, we propose a preemptive multi-queue fair queuing (P-MQFQ) algorithm that uses a centralized queue to fairly dispatch threads from different programs based on their received CPU bandwidth from multiple cores. We demonstrate that P-MQFQ can be approximated by augmenting the existing load balancing in the OS without requiring to implement the centralized queue or undermining scalability. We implement P-MQFQ in Linux and Xen, respectively, and show significantly improved utilization and performance for parallel programs.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Scheduling**; • **Computer systems organization → Multicore architectures**; **Cloud computing**; • **Mathematics of computing →** *Queueing theory*.

KEYWORDS

Performance, Scheduling, Multicore, Cloud Computing.

ACM Reference Format:
Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu and Hai Jin. 2019. Preemptive Multi-Queue Fair Queuing. In *Proceedings of The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19).* ACM, 12 pages. https://doi.org/10.1145/3307681.3326605

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPDC '19, June 22–29, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3326605

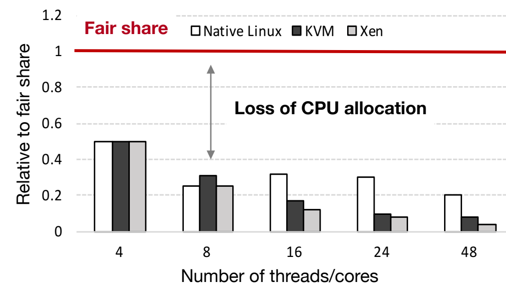

Figure 1: The deficiency in state-of-the-art multiprocessor schedulers. Multithreaded programs with blocking synchronization suffer unfair CPU allocation.

1 INTRODUCTION

The prevalence of shared services, such as multi-tenant clouds [2, 19, 50], shared storage [24, 43], and multi-user clusters [12, 30], has led to a plethora of studies on fair and predictable resource allocation of shared resources. An important method for achieving resource fairness is using a fair queuing (FQ) scheduler, which allows competing tenants to take turns to use a shared resource. While FQ is a packet scheduling technique originally designed for sharing a network link between multiple flows, it has since been extended to managing various types of resources [43, 46], scheduling flows with variable packet lengths [4, 5, 13, 28], and supporting fair queuing on multiple links [6].

Modern operating systems (OSes) and hypervisors employ variants of FQ algorithms in the thread scheduler or the virtual CPU (vCPU) scheduler. In this context, an FQ scheduler allocates processor bandwidth, i.e., CPU cycles, to competing threads [1]. The time quantum each thread receives in a round corresponds to a packet serviced in the original FQ algorithm. On a single-core system, FQ schedulers effectively guarantee fair CPU allocation among active threads while allowing some threads to use more than their share if otherwise the core would become idle. This ensures that the scheduler is *work conserving*. However, on multicore systems, existing FQ schedulers fail to provide necessary resource isolation between competing applications, causing unfairness and performance unpredictability.

Figure 1 demonstrates the severity of unfairness in state-of-the-art multicore schedulers. We placed two multi-threaded applications to share the same set of cores, with the number of threads in each application matching the number of cores. We measured the aggregate CPU allocation to each application as a whole to evaluate

[1]We use threads and vCPUs interchangeably.

the fairness of a multiprocessor scheduler. In the tests of hypervisor schedulers, the two applications ran in separate virtual machines (VMs) with the same number of vCPUs. Ideally, if *max-min* fairness is enforced, each application should receive a fair share of the total capacity of all available cores if the aggregate demand of all its threads exceeds the fair share. We empirically confirmed that both applications were able to consume almost all CPU when running in solo. The program under test was *streamcluster* from the PARSEC benchmark suite [45]. It employs blocking synchronization, which puts threads to sleep if they fail to enter the critical section. The colocated application was a synthetic benchmark with persistent CPU demand (see Section 4.2 for details). It could be either a parallel program with busy-waiting (spinning) synchronization or a multiprogramming workload. As shown in Figure 1, *streamcluster* suffered unfair CPU allocation under all three schedulers and the unfairness was aggravated as the number of threads/cores increased.

This previously unknown deficiency of multicore schedulers can cause significant performance slowdown and high variability to parallel applications in multi-tenant systems. As a consequence, due to concerns of poor service quality, leading public cloud providers, such as Amazon AWS, Microsoft Azure and Google Compute Engine, do not allow CPU multiplexing among symmetric multiprocessing (SMP) VMs. Such a conservative strategy diminishes the benefits of workload consolidation, resulting in low CPU utilization and high user cost. This deficiency can also hamper the adoption of the emerging serverless computing model, in which thousands of short-lived, possibly inter-dependent and/or chained containers are multiplexed on multicore systems.

The discovered unfairness in multicore scheduling is the result of the complex interplay between parallel workloads and OS thread schedulers. On the one hand, parallel programs rely on simultaneous access to CPU to make collective progress among multiple threads and otherwise suffer substantial performance slowdown if critical threads holding important locks are preempted. The remaining threads who are waiting on the synchronization cannot make progress, either performing futile spinning or being put to sleep (block). On the other hand, multicore schedulers enforce fair CPU allocation on a per-core basis and are usually work-conserving. Therefore, threads that are idling due to synchronization forfeit their CPU shares, leading to unfair allocation between a parallel program and other competing programs. OS Load balancing could further aggravate this problem. Frequently idling threads, which show low CPU load, are gradually moved onto a few cores as consolidating fragmented load helps improve load balance. If CPU stacking occurs, sibling threads belonging to the same application compete with each other, introducing more idleness.

The culprit of such harmful interactions is twofold – 1) parallel programs exhibit *deceptive idleness* (DI) under contention, failing to expose their actual CPU demand to the OS scheduler; 2) there is no mechanism to fairly schedule inter-dependent threads on multiple CPUs. Therefore, parallel programs are unfairly penalized for being idle but not appropriately compensated. To address these issues, we extend the FQ algorithm for sharing a single network link to thread scheduling on multiple cores. We propose preemptive multi-queue fair queuing (P-MQFQ), a close approximation of the idealized generalized processor sharing (GPS) service discipline for multiple CPUs. P-MQFQ assumes a centralized queue to dispatch threads to multiple CPUs such that competing programs as a whole receive a fair share of the aggregated capacity of multiple CPUs. To tackle deceptive idleness, P-MQFQ allows threads from underserved programs to preempt currently running threads from other programs. As such, programs experiencing deceptive idleness are temporarily prioritized to catch up with those who have exceeded their fair shares.

Maintaining a centralized thread queue imposes a major scalability bottleneck in multicore scheduling. To implement P-MQFQ in real systems, we augment state-of-the-art multicore schedulers that use independent local queues with three queue operations: Migrate, Ppreempt and Switch to approximate global fairness on multiple queues. We have implemented P-MQFQ in Linux completely fair scheduler (CFS), the default scheduler for native Linux and KVM, and Xen's credit scheduler. Experimental results show that P-MQFQ effectively addresses deceptive idleness and improves fairness in multicore scheduling. Parallel programs with blocking synchronization are able to utilize close to, but never exceed their fair shares. Our results also show significant improvement over three representative multicore optimizations.

2 BACKGROUND AND PROBLEM ANALYSIS

Classical fair queuing (FQ) algorithms are concerned with sharing a single network link among a set of flows [39]. Each flow consists of a sequence of packets that need to be transmitted through the shared link. Generalized processor sharing (GPS) is a reference model for fair queuing disciplines. A GPS server operates at a fixed rate r and can transmit multiple flows simultaneously. A flow is backlogged at time t_1 if a positive amount of that flow's requests is queued at time t_1. Then the idealized GPS model guarantees that during any interval $[t_1, t_2]$, in which the set of backlogged flows $\mathcal{F}$ is unchanged, each backlogged flow i receives a minimum service rate r_i according to its weight ϕ_i:

$$r_i = \frac{\phi_i}{\sum_{j \in \mathcal{F}} \phi_j} r.$$

To approximate GPS in realistic systems that can only transmit one flow at a time, a number of packet-based GPS (PGPS) approximations have been developed [4, 5, 13, 18, 20, 21, 24, 32, 39, 40, 42, 52, 53]. PGPS transmits packet by packet in its entirety and only serves one packet at a time. PGPS approximates bandwidth allocation in GPS by serving packets in the increasing order of their finish time F under GPS. PGPS uses a notion of *virtual time* to track the progress of packet transmission in GPS. Virtual time measures the number of bits has been transmitted on a per-flow basis. System virtual time and a flow's virtual time advance at a rate of $\frac{r}{\sum_{j \in \mathcal{F}} \phi_j}$ and $\frac{\phi_i}{\sum_{j \in \mathcal{F}} \phi_j} r$, respectively. If there is only one backlogged flow, its virtual time advances at the rate of server capacity r and is identical to wall-clock time. When multiple flows are being serviced, flow virtual time is slower than real time and reflects its progress under the idealized GPS discipline. System virtual time stops advancing when the server is idle.

Virtual time-based PGPS implementation assigns each packet k from flow i a virtual start tag S_i^k and a virtual finish tag F_i^k when the packet arrives. Assume that server capacity is normalized to

1 and all flow weights sum to 1. The finish tag of a packet can be calculated based on its start tag:

$$F_i^k = S_i^k + \frac{L_i^k}{\phi_i}, \quad (1)$$

where L_i^k is size of the packet. The start tag is the maximum of system virtual time $v(t)$ at packet arrival time t and the finish tag of the last packet from the same flow:

$$S_i^k = \max\{F_i^{k-1}, v(t)\}. \quad (2)$$

The update of start tag ensures that inactive flows would not lag arbitrarily behind active flows in virtual time such that they could penalize active flows for utilizing the bandwidth left idle by inactive flows.

2.1 Start-time Fair Queuing

One obstacle to implement virtual time-based PGPS is the computation of system virtual time $v(t)$, which requires the simulation of a bit-by-bit GPS server. This simulation is computationally expensive to perform at each packet dispatch. Start-time fair queuing (SFQ) [20] dispatches packets in the increasing order of start tags instead of finish tags. Ties are broken arbitrarily. Further, $v(t)$ is defined as the start tag of the packet in service at time t. SFQ offers two advantages [20]: 1) the packet size does not need to be known a priori and SFQ is able to handle variable server rates; 2) the computation of $v(t)$ is inexpensive as it only requires to examine the packets in service.

2.2 Fair-share CPU Scheduling

Similar to fair queuing in shared network, fair-share CPU scheduling aims to fairly allocate CPU bandwidth to competing threads. The time quantum each thread receives each time it runs on CPU is equivalent to dispatching a packet from a flow. Due to the advantages we previously discussed, SFQ and its variants are widely adopted in fair-share CPU scheduling. The default Linux completely fair scheduler (CFS) implements SFQ for fair sharing each individual CPU. CFS maintains a per-thread *virtual runtime* (vruntime) for each thread and tracks the minimum vruntime on a CPU. A thread's vruntime is updated each time it finishes a time quantum and the advancement is calculated based on the length of the time quantum and the thread's weight. CFS schedules threads based on the increasing order of their vruntimes. The minimum vruntime is defined as the maximum of the current minimum vruntime and the vruntime of the current running (in service) thread. It is updated each time a thread finishes a time quantum. When a thread wakes up from idling, its vruntime is set to the maximum of the current minimum vruntime and its vruntime before sleep.

In a multiprocessor (or multicore) [2] system, the operating system (OS) runs multiple copies of the fair-share CPU scheduling algorithm, one on each CPU, and relies on load balancing to evenly distribute threads over CPUs. Ideally, if all threads are runnable (backlogged) all the time and there are an equal number of threads on each CPU, fairly allocating CPU on a per-CPU basis leads to global fairness. However, system load often fluctuates over time and threads need to be migrated across CPUs to balance the load. Load imbalance undermines global fairness among threads as threads on CPU with higher load receive less CPU than those on CPUs with less load even they have the same weight. The fundamental problem is that weight is only significant to threads on the same CPU and affects the allocation on a particular CPU.

Unfortunately, thread migration is expensive as it requires double run queue locking to move a thread from the source to the destination CPU and it also undermines cache locality. Therefore, load balancing is performed infrequently and largely based on heuristics. For example, Linux performs load balancing on two occasions: 1) when a core becomes idle, it pulls threads from the busiest CPU; 2) the OS periodically moves threads from the busiest CPU to the least loaded CPU. The busyness of a CPU is measured by the number of runnable threads during a specified period. The busyness measure decays over time with recently runnable threads weighing more than older threads.

[2]We use multiprocessor and multicore interchangeably throughout this paper to refer to multiple CPU queues.

2.3 Deceptive Idleness

Despite fair-share scheduling on individual CPUs, parallel programs are susceptible to unfair CPU allocation on multiprocessors. A critical component of parallel programs is synchronization, which serializes the execution of threads in the critical section through locking. Threads could either spin on the lock or block if they failed to acquire the lock. Futile spinning wastes CPU cycles and can inflict priority inversions [7]. Therefore, blocking and the hybrid spin-then-block synchronization, which eventually puts lock waiter threads to sleep, is widely adopted in parallel libraries. For example, the default implementations of mutex locks, barrier and semaphore in Pthread use blocking. Similar to an issue in disk scheduling [23], parallel threads using blocking synchronization exhibit *deceptive idleness* (DI) when a thread on the critical path is preempted. The critical thread could be a lock holder or a designated waiter to acquire the lock, which refers to the well-studied lock-holder preemption (LHP) [17] and lock-waiter preemption (LWP) problems [1, 37], respectively. However, it is not well understood why these problems cause cascading performance degradation.

Recall that SFQ-based fair sharing is work-conserving and does not penalize threads that consume resources that are otherwise left idle. Since system virtual time $v(t)$ on each CPU advances according to the start tag of the current running thread, the finish tag F_i of an idle thread is guaranteed to be smaller than $v(t)$ when it wakes up. Therefore, according to equation 2, an idle thread will align its start tag with $v(t)$ when waking up. This will allow the continuously running thread, which consumes more than its fair share, an equal opportunity to compete with the waking thread. As this pattern repeats, a frequently idling thread forfeits its share but is not compensated. Since in multiprocessor scheduling, CPUs independently enforce fair allocation on local queues, deceptive idleness costs parallel programs a significant proportion of their share of CPU. The heuristic-based load balancing further aggravates the issue. A CPU with deceptively idling threads appears to be lightly loaded and pulls threads from heavily loaded CPUs. If sibling threads belonging to a parallel program are stacked on the same CPU, intra-program CPU competition leads to severe serialization and more idleness.

2.4 Multi-Queue Fair Queuing

Multi-queue fair queuing is concerned with sharing the aggregated capacity of multiple links among flows [6, 24]. A notable work extends SFQ to dispatch concurrent requests to utilize multiple links and controls the number of requests dispatched from different flows to enforce fair sharing [24]. The core problem is how to define system virtual time $v(t)$. Assume that there are D queues. Min-SFQ(D) [24] defines $v(t)$ as the minimum start tag of any outstanding requests, which include queued and dispatched yet completed requests. SFQ(D) defines $v(t)$ as the maximum start tag of any dispatched yet completed requests.

The distinction between the two algorithms is important to address deceptive idleness. 1) Min-SFQ(D) advances $v(t)$ according to the lagging flow that cannot fully utilize its share. For example, a slow flow with no concurrency but always backlogged determines $v(t)$. Thus, it always has precedence over faster flows that use excessive resources. However, Min-SFQ(D) can cause starvation to fast flows if the slow flow issues a burst of requests. 2) SFQ(D) advances $v(t)$ according to the start tag of last dispatched request in backlogged flow. Therefore, it allows multiple queues to be fully utilized without penalizing backlogged flows or compensating lagging flows. Four-tag start-time fair queuing (FSFQ(D)) [24] combines the benefits of Min-SFQ(D) and SFQ(D) by maintaining four tags, the adjusted start and adjusted finish tags in Min-SFQ(D) and start and finisht tags in SFQ(D), for each request. Request scheduling is still based on start tags under SFQ(D) but ties are broken according to adjusted start tags under Min-SFQ(D). This compensates a lagging flow by giving it precedence in breaking ties.

Unfortunately, none of these algorithms is able to address deceptive idleness in parallel programs. Assume D queues (CPUs). When deceptive idleness occurs and there is only one critical thread active in a parallel program, in the worst case, there could be $D-1$ threads dispatched from another backlogged program that are ahead of the blocked threads in the parallel program. Therefore, at each critical section of length l_c, a parallel program loses $(D-1)l_{max}$ utilization to another backlogged program after finishes Dl_{nc} cycles, where l_{max} is the time quantum and l_{nc} is the length of the non-critical section. Under Min-SFQ(D), which always treats the parallel program as a lagging flow, the parallel program receives at best $Dl_{nc} + l_{max}$ cycles on D CPUs in each round while the competing backlogged program receives $(D-1)l_{max}$ cycles. We use half of the total cycles allocated to both programs as the fair share and measure fairness using the *absolute relative lag* $|\frac{S_{\text{fair}}-S_{\text{parallel}}}{S_{\text{fair}}}|$, where S_{parallel} is the parallel program's CPU allocation. Therefore, we have

$$lag = \left|\frac{l_{max} - l_{nc}}{l_{max} + l_{nc}} - \frac{2l_{max}}{D(l_{max} + l_{nc})}\right|. \quad (3)$$

One can observe that when D is large and $l_{nc} << l_{max}$, $lag \to 1$. It suggests that parallel programs with fine-grained synchronization could suffer starvation on large-scale multicore systems due to deceptive idleness. Starvation is aggravated under SFQ(D) and FSFQ(D) as parallel threads forfeit CPU shares after they wake up.

3 PREEMPTIVE MULTI-QUEUE FAIR QUEUING

Our analysis finds that the keys to address deceptive idleness are 1) deriving a global dispatch order of threads on multiple CPUs so as to enforce fairness at the program level; 2) devising a multi-queue fair queuing algorithm that allows threads from a lagging program to be timely scheduled on CPU. In what follows, we elaborate on the design of preemptive multi-queue fair queuing (P-MQFQ) to meet these goals and present a practical implementation of P-MQFQ in state-of-the-art multicore schedulers.

Figure 2 (a) shows the P-MQFQ model for multiprocessor scheduling. The objective is to schedule requests from different programs (i.e., P1 and P2) such that the aggregated CPU time received by all threads of each program is proportional to program weights. If processors are available, programs with multiple threads can have multiple requests dispatched. A request represents the CPU demand from one thread. The request service time is either the maximum time quantum a thread can run or the actual runtime if the thread blocks before its time quantum expires. A thread immediately re-submits a request for another time quantum if its current quantum expires or waits until waking up to submit a request if it was blocked. Similar to the existing MQFQ algorithms, P-MQFQ maintains a centralized request dispatch queue, where requests are scheduled in the increasing order of their start tags. The centralized queue also tracks system virtual time $v(t)$. The algorithm is defined as follows:

1. System virtual time $v(t)$ is defined as the maximum start tag of all requests in service. Per-program virtual time $v_i(t)$ is defined as the maximum start tag of requests in service that belong to a particular program. Defining virtual time according to requests in service allows lagging programs to preempt threads from aggressive programs.
2. If a program is backlogged, i.e., there is at least one thread from the program having a request running on a CPU or queued, a request's start tag is the maximum of the program's last finish tag and the program's virtual time. Otherwise, i.e., the program has been idling or is newly launched, request start tag is aligned with the system virtual time.
3. On arrival, if a request's start tag is smaller than the system virtual time, it indicates that another program receives more service than this program. The newly arrived request preempts the request with the largest start tag in service. Note that the preempted request is guaranteed to be from a different program as only such requests can advance system virtual time beyond the new request's program virtual time.
4. After the request with the largest start tag in service is preempted, the system virtual time $v(t)$ is updated to the second largest start tag in service. The finish tag of the preempted request is $S_i^k + \frac{l_r}{\phi_i}$, where S_i^k is the start tag, l_r is the amount of time the request has been running on CPU before being preempted and ϕ_i is the weight of the program.

The use of two types of virtual time, i.e., system virtual time and per-program virtual time, allows P-MQFQ to track the amount of service received by each program and prioritize lagging programs. Most importantly, request preemption guarantees that threads blocked

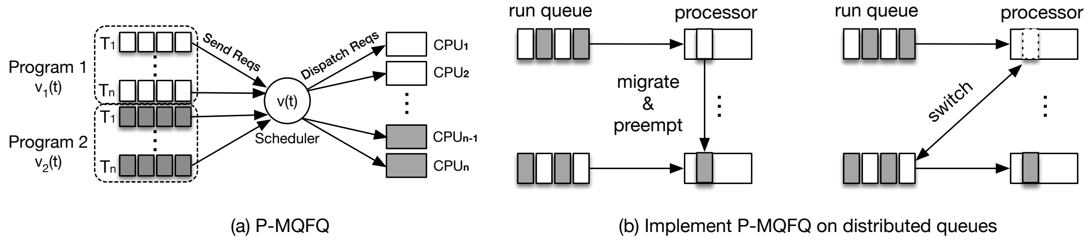

Figure 2: (a) The idealized P-MQFQ model with a centralized request queue. (b) A practical implementation of P-MQFQ on distributed queues by augmenting multicore schedulers with three run-queue operations.

due to synchronization can have requests immediately scheduled after waking up if they belong to a lagging program. This prevents deceptive idleness from happening.

Fairness Analysis

As shown in [20], during any interval $[t_1, t_2]$, the difference between the amount of service received by two backlogged flows f and g under start-time fair queuing (SFQ) is given as:

$$\left| \frac{w_f(t_1,t_2)}{\phi_f} - \frac{w_g(t_1,t_2)}{\phi_g} \right| \leq \frac{l_f^{max}}{\phi_f} + \frac{l_g^{max}}{\phi_g}, \tag{4}$$

where ϕ_f and ϕ_g are the weights of flows f and g, and l_f^{max} and l_g^{max} are their maximum packet sizes, respectively. In CPU fair scheduling, l_f^{max} refers to the maximum time thread f is allowed to execute on CPU. In SFQ, system virtual time $v(t)$, which is used to track the progress under the idealized GPS model, is defined as the start tag of the packet (request) in service (running on CPU) at time t. Inaccuracies arise in using start tags to track the progress under GPS during interval $[t_1, t_2]$ if there exist packets that arrived before t_1 and completed before t_2 or arrived before t_2 and completed after t_2. The inaccuracy is at most $\frac{l_{max}}{\phi}$ for any flow. Therefore, the theorem in (4) suggests that unfairness is maximum when one flow receives maximum service while the other minimum service. Since there is only one processor in a uniprocessor system, the unfairness is bounded by the service equivalent to serving two maximum packets (requests). This unfairness bound extends to $(D+1)*(\frac{l_f^{max}}{\phi_f} + \frac{l_g^{max}}{\phi_g})$ in a D-processor system under SFQ(D) [24] when a flow with no concurrency lags behind aggressive flows that can fully utilize D processors. Since there are now D processors, there exist at most D requests that do not completely fall in the interval $[t_1, t_2]$ but either started or completed in the interval. This leads to a service difference of $(D+1)*(\frac{l_f^{max}}{\phi_f} + \frac{l_g^{max}}{\phi_g})$, in which one additional term $\frac{l_f^{max}}{\phi_f} + \frac{l_g^{max}}{\phi_g}$ is added to account for the unfairness of SFQ on a single CPU.

The key difference between P-MQFQ and SFQ-based approaches is that P-MQFQ allows threads from a lagging program to preempt a running thread at any time. If preemption had no cost, packet (request) scheduling can be made bit-by-bit (cycle-by-cycle). As such, in uniprocessor systems, the unfairness bound reduces to $\frac{1}{\phi_f} + \frac{1}{\phi_g}$, assuming that 1 is the resolution of virtual time. Unlike SFQ(D) whose unfairness bound scales with the number of processors D, ideally, P-MQFQ retains the same constant unfairness bound in multiprocessor systems as that in uniprocessor systems. When the program-level virtual time of a lagging program falls behind, threads from this program can even preempt threads from other programs that are dispatched but not started on CPU. Note that this is not possible in uniprocessor systems because dispatched threads must have a smaller start tag than any queued threads. In real systems, CPU schedulers enforce a minimum time slice l_{min} to prevent too frequent context switching. Therefore, in the worst case, when a lagging program intends to preempt, there could be at most D threads that have just started on CPU and are each guaranteed to run for l_{min}. Following the proof in [24], the unfairness in P-MQFQ is bounded by $(D+1)*(\frac{l_{min}}{\phi_f} + \frac{l_{min}}{\phi_g})$.

3.1 Approximating P-MQFQ on Distributed Queues

P-MQFQ relies on a centralized queue to derive a global notion of virtual time and requires tracking per-program virtual time. However, state-of-the-art multicore schedulers employ a distributed queue architecture because it scales well with a large number of CPUs. Each CPU maintains per-CPU virtual time and independently enforces fair sharing on local queues. To approximate P-MQFQ without requiring to maintain the system and per-program virtual time, we augment multicore schedulers with distributed queues with three run queue operations: Migrate, Ppreempt and Switch.

With distributed queues, there lacks a notion of global system virtual time. Per-CPU system virtual time progresses independently. Therefore, request start tags on different queues do not reflect the amount of service received by requests; in other words, start tags are not comparable across queues. Recall that P-MQFQ preempts the in-service request with the largest start tag. It is equivalent to finding the last dispatched request from the program that received the most service. We relax the requirement for finding the global maximum of all start tags. Instead, as shown by the left figure in Figure 2 (b), after a request is completed (i.e., its time slice expires), the thread who submitted this request preempts a thread that has received more service on another CPU queue. We also ensure that the preempted thread belongs to a different program. Since per-CPU system virtual time progresses as request start tag increases, we consider the current running thread on a queue with the fastest virtual time progression to have received the most service. If such a thread is found on a queue, the thread with an expired time quantum is Migrated to the queue and Preempts the running thread.

Per-program virtual time tracks the aggregate service time of all threads in a program on multiple queues. During synchronization, per-program virtual time advances slowly as only the critical thread is active. This ensures that the critical thread has a small start tag and is always timely scheduled to avoid lock holder or waiter preemption. However, tracking per-program virtual time across multiple queues will impose a significant scalability bottleneck. We approximate the effect of per-program virtual time by ensuring that the critical thread is timely scheduled. As shown by the right figure in Figure 2 (b), if a thread is blocked (dotted box) before its time quantum expires, it iterates over queues to look for a runnable (queued) sibling thread from the same program. If found, P-MQFQ will switch these two threads and allows the selected runnable thread to use the remaining time quantum left by the blocked thread. The Switch operation moves the two threads but retains the virtual time on the original queues. The selected runnable thread will inherit the virtual time of the blocked thread and vice versa. This ensures that thread switching does not undermine fairness and programs can continuously receive service during synchronization.

3.2 Implementation

Native Linux and KVM Since the completely fair scheduler (CFS) in Linux is an implementation of SFQ on a per-CPU basis, P-MQFQ naturally extends to CFS. In CFS, each CPU (queue) independently maintains per-queue system virtual time (minimum vruntime), which is the vruntime of the current running thread. Minimum vruntime is updated at each timer interrupt (by default every 1ms in Linux) with the vruntime of the current running thread. P-MQFQ tracks the progression of per-queue minimum vruntime in a 3ms time window and considers the queue with the largest vruntime advancement as the one receiving most service. When a thread is blocked, it scans all queues to look for a runnable thread with the same parent and switch to the thread. This method finds sibling threads belonging to the same program in native Linux and sibling vCPUs in KVM.

Xen In Xen's credit scheduler, CPU allocation is measured by *credits*. As a vCPU consumes CPU, credits are debited and the balance determines the vCPU's priority. vCPUs with non-negative credit balance are assigned with the normal `UNDER` priority while those with negative balance are given a lower `OVER` priority. Xen refills vCPUs' credits at the beginning of each accounting period (every 30ms). Each time a vCPU is allocated the amount of credits that lasts for a time quantum (30ms in Xen). If a vCPU cannot use up its credits in an accounting period, the unused credits are discarded, which is intended to prevent vCPUs from accumulating credits. Although the credit scheduler does not use the notion of virtual time, the consumption of credits is equivalent to the progression of virtual time. To implement P-MQFQ, we record the discarded credits on each CPU in each accounting period. The CPU with the least discarded credits is considered to have received the most service and the running vCPU on this CPU is preempted. The implementation of the Switch operation is similar to that in Linux.

4 EVALUATION

In this section, we present an evaluation of P-MQFQ in both a bare-metal Linux environment and two representative virtualized environments (KVM and Xen). We first study the effectiveness of P-MQFQ in addressing deceptive idleness in parallel workloads with blocking synchronization (§ 4.2). We then show that P-MQFQ can significantly improve the performance of parallel workloads with different types of synchronization (§ 4.3). Finally, we study the overall system efficiency under P-MQFQ when multiple parallel applications are each scheduled by P-MQFQ (§ 4.4).

4.1 Experimental Setup

All experiments were performed on two servers. One is a DELL PowerEdge T420 server with two six-core Intel Xeon E5-2420 processors (24 cores with hyperthreading enabled) and 32GB memory. Another is a DELL PowerEdge R830 with four twelve-core Intel Xeon E5-4640 processors (48 cores with hyperthreading disabled) and 256GB memory. The settings were intended to demonstrate the effectiveness of P-MQFQ at different scales as well as with or without hyperthreading. Linux 4.1.39 was used as the native Linux OS, the host OS in KVM, and the guest OS in KVM and Xen. Xen 4.5.0 was used in the Xen test. For the tests in KVM and Xen, two VMs were used to run the parallel programs under test and the interfering workloads. All results were the average of 5 runs.

Workloads We selected the PARSEC [45] and NASA parallel benchmarks [3] as the parallel workloads under test. PARSEC is a shared memory parallel benchmark suite with various blocking synchronization primitives such as mutex locks, condition variables and barriers. We compiled PARSEC using pthreads and used the native input size. NASA parallel benchmarks include 9 parallel programs. We used the OpenMP implementation of benchmarks with the class C input size. We set the environment variable `OMP_WAIT_POLICY` to `INACTIVE` to enable blocking synchronization.

Scheduling strategies We compare the performance of P-MQFQ with the baseline multicore schedulers and three representative scheduling strategies:

- `Linux CFS` and `Xen credit scheduler` were used as the baseline schedulers in native Linux, KVM and Xen, respectively.
- `Pin + {Linux, KVM and Xen}`: To avoid the severe CPU stacking issue due to deceptive idleness, we pinned threads and vCPUs to individual cores to disable OS load balancing. This strategy is often employed in production systems to improve performance predictability and preserve locality.
- `Relaxed-Co`: we implemented the VMware's relaxed co-scheduling in native Linux, KVM and Xen. Relaxed-Co monitors the execution skew of each vCPU (thread) and stops the vCPU that makes significantly more progress than the slowest vCPU. When a VM (program)'s leading vCPU (thread) is stopped, the hypervisor switches it with its slowest sibling vCPU to boost the lagging vCPU.
- `Gleaner [14]`: In multi-tenant systems, CPU multiplexing causes suboptimal scheduling and fragmented CPU allocation in parallel programs. `Gleaner` consolidates fragmented CPU allocation into a few dedicated CPUs. Although CPU consolidation does not provide enough concurrency to user-level threads, it avoids expensive trapping to the hypervisor due to idling and harmful competition with co-running applications.

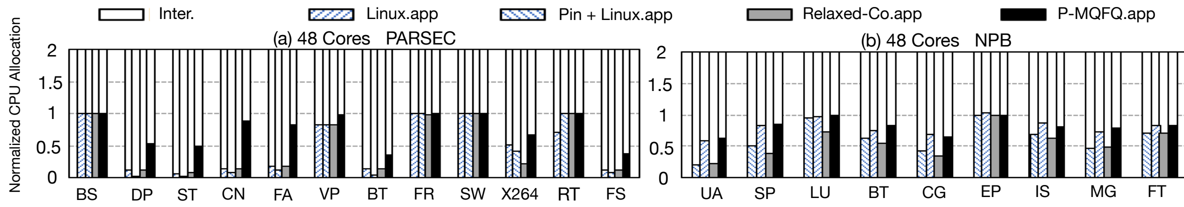

Figure 3: Normalized CPU allocation on physical machines for PARSEC (a) and NPB (b).

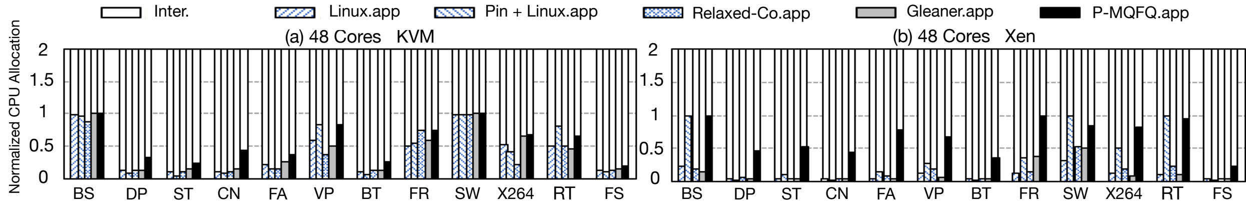

Figure 4: Normalized CPU allocation on KVM (a) and Xen (b) for PARSEC.

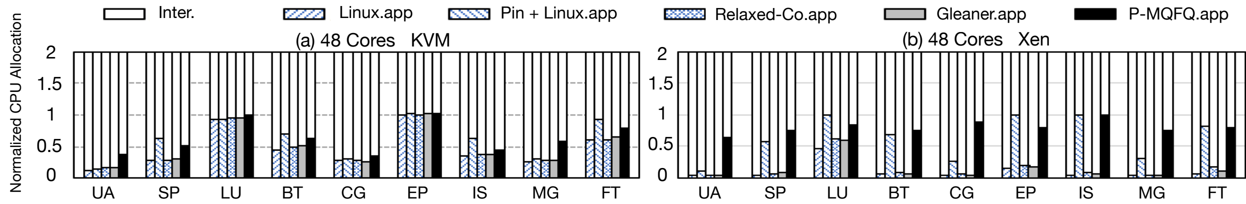

Figure 5: Normalized CPU allocation on KVM (a) and Xen (b) for NPB.

4.2 Addressing Deceptive Idleness

In this section, we evaluate the effectiveness of P-MQFQ in addressing deceptive idleness and improving the fairness of CPU allocation. Similar to the experiments in Figure 1, we collocated parallel workloads with a synthetic benchmark that had persistent CPU demand. The synthetic benchmark consisted of the same number of CPU hogs as the number of CPUs and was intended to create a contentious scenario to cause unfair CPU allocation. Figures 3, 4 and 5 show the normalized CPU allocation to the foreground parallel programs and the background interfering workload. The stacked bars show the allocation to the parallel applications (e.g., `P-MQFQ.app`) and the interfering workload (e.g., `Inter.`) under various approaches. A normalized allocation of 1 refers to fair allocation while a value less than 1 indicates that parallel programs receive less than the fair share. From these figures, we have the following observations:

First, compared with the baselines, P-MQFQ significantly increased the CPU allocation to most parallel workloads. P-MQFQ is most effective for programs with fine-grained synchronization. For example, among PARSEC benchmarks, *dedup*, *streamcluster* and *canneal* benefited most from P-MQFQ. According to the equation 3, the shorter the non-critical section (l_{nc}), i.e., more frequent synchronization, the higher degree of unfairness. In comparison, the improvement on CPU allocation was less in NPB benchmarks, which have much longer non-critical sections and less frequent synchronization. This observation was consistent both in the physical and virtualized environments.

Second, most benchmarks suffered more from deceptive idleness as the number of CPUs scaled but P-MQFQ also had diminishing gains [3]. Intuitively, deceptive idleness is aggravated with a larger number of threads as more threads would be idling during synchronization. As shown in equation 3, the degree of unfairness increases with the number of CPUs D. P-MQFQ mitigated deceptive idleness by prioritizing the critical thread to avoid unnecessary idleness. However, P-MQFQ was unable to entirely eliminate idleness at scale, in which the critical section weighed more to the non-critical section. We empirically confirmed that fine-grained programs, such as *streamcluster* and *canneal*, were unable to utilize their fair shares even in solo mode. For example, *streamcluster* only utilized around 1400% (equivalent to the capacity of 14 CPUs) CPU on the 48-core machine with 48 threads. This limits the gain of P-MQFQ at scale.

Third, different multicore schedulers suffered differently from deceptive idleness, which also affected the effectiveness of P-MQFQ. Xen employs a longer time quantum l_{max} (i.e., 30ms) than that in CFS (i.e., 6ms). Therefore, parallel programs' CPU allocation was much lower in baseline Xen compared to Linux and KVM. This does not affect the effectiveness of P-MQFQ as thread preemption in P-MQFQ effectively mitigate deceptive idleness, regardless of

[3] We have tested with 12, 24, and 48 cores. Due to limited space, only the results on 48 cores are shown.

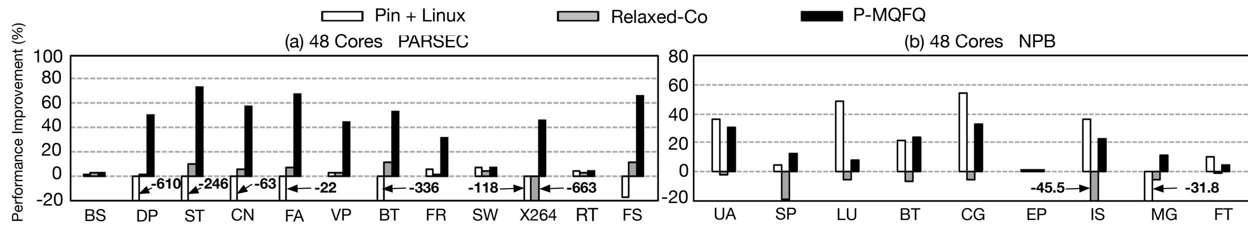

Figure 6: Performance improvement of PARSEC (a) and NPB (b) on physical machine.

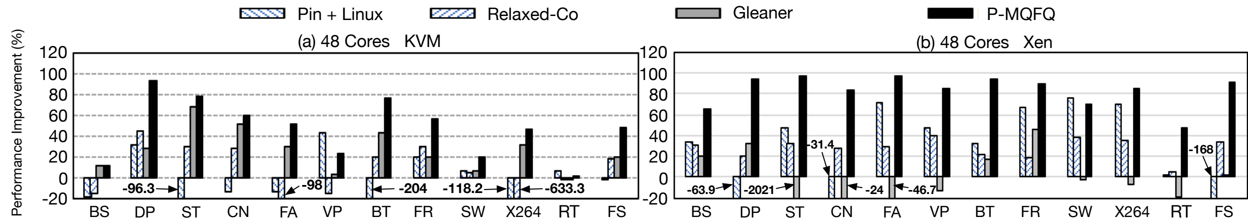

Figure 7: Performance improvement of PARSEC on KVM (a) and Xen (b).

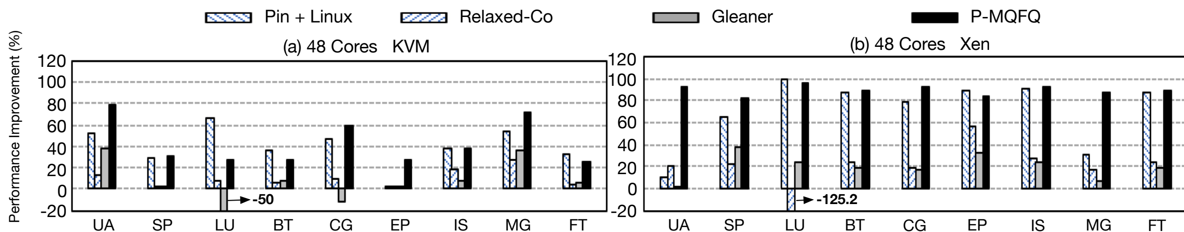

Figure 8: Performance improvement of NPB on KVM (a) and Xen (b).

the length of the time quantum. P-MQFQ was able to achieve a similar level of utilization improvement in Xen compared to that in Linux. In contrast, P-MQFQ was less effective in KVM. Although both native Linux and KVM use CFS as the scheduler, in KVM, the scheduling entity is the vCPU. KVM employs hardware assisted virtualization (e.g., Intel VT-x) to virtualize vCPUs and thus incurs higher overhead for vCPUs migration. Significantly, a vCPU is not immediately eligible for migration after it is preempted because of a write barrier to enforce consistency across cores. In KVM, it takes more than 1ms before P-MQFQ can migrate a critical thread after the thread is preempted, thereby unable to timely schedule the critical thread. As a result, P-MQFQ achieved lower CPU allocation in KVM than that in Linux and Xen.

Fourth, P-MQFQ achieved a higher CPU allocation than the three representative scheduling strategies `Pin`, `Relaxed-co` and `Gleaner` in all tests except for *EP* from NPB, which is embarrassingly parallel and uses no synchronization. The key in P-MQFQ to increasing CPU utilization is to eliminate idleness as much as possible. To achieve this goal, P-MQFQ allows a preempted thread a chance to continue running by preempting another over-serving threads and moves threads across CPUs to avoid forfeiting allocated time quantum. Neither of the three approaches devise both optimizations. `Pin` avoids CPU stacking due to deceptive idleness but does not address the preemption of critical threads; `Relaxed-co` could not guarantee the simultaneous progress of all threads and thus is still susceptible to idleness-induced CPU stacking; thread consolidation in `Gleaner` does not expose enough parallelism to the user-level thread, thereby aggravating serialization at the critical section.

4.3 Improving Performance

For programs with blocking synchronization, scheduling inefficiencies manifest as excessive idleness. Thus, it is expected that P-MQFQ leads to performance improvement as it can eliminate much of the idleness and improve CPU utilization. Figures 6, 7 and 8 show the execution time of PARSEC and NPB benchmarks due to different scheduling strategies in native Linux, KVM and Xen, respectively. Performance is normalized to that in the baselines, i.e., vanilla Linux, KVM and Xen. From these figures, we can see that P-MQFQ improved the performance of all benchmarks compared to that in the baselines. The performance improvement was up to 97% and 96% for PARSEC and NPB benchmarks, respectively.

In contrast, there were benchmarks suffering performance degradation under other approaches. For example, *streamcluster* had a degradation of 400% under `Pin` on the 24-core physical machine since `Pin` cannot mitigate the deceptive idleness; `Relaxed-co` degraded *x264* by as much as 663% on the 48-core physical machine due to the loss of 25% CPU allocation; `Gleaner` inflicted a 2021% slowdown to *streamcluster* on the 48-core Xen machine. The reason

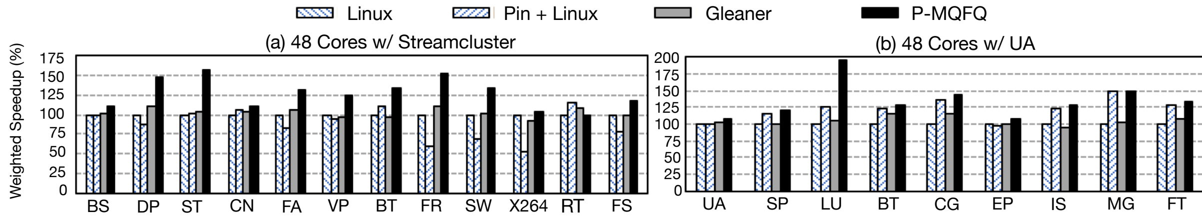

Figure 9: Weighted speedup of PARSEC (a) and NPB (b) on physical machine.

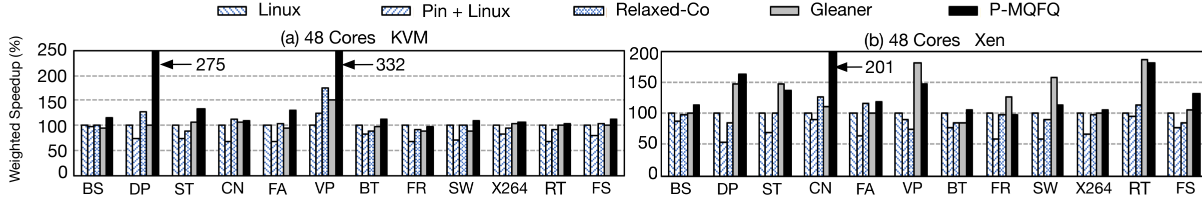

Figure 10: Weighted speedup of PARSEC and streamcluster on KVM (a) and Xen (b).

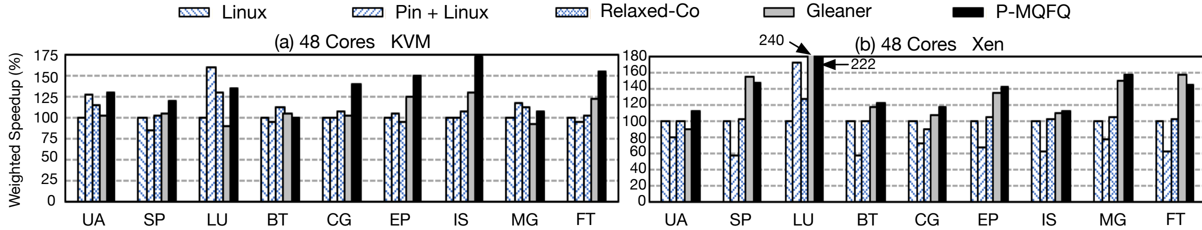

Figure 11: Weighted speedup of NPB and UA on KVM (a) and Xen (b).

is that Xen hypervisor employed the comparison of vCPU priority to decide the load balance. Therefore, consolidating more threads onto a single vCPU in Xen would make the stacking problem more serious and thus performance is further degraded.

4.4 System Efficiency

The results presented so far focused on improving the utilization and performance of the foreground parallel programs. P-MQFQ did not take effect for the background workloads. We are also interested in evaluating P-MQFQ in managing multiple parallel workloads, each is actively scheduled by P-MQFQ. We collocated two blocking parallel workloads to share the same set of CPUs and used the geometric mean of individual programs' speedups (weighted speedup) to measure the overall system efficiency. The higher the weighted speedup, the higher the system efficiency. A weighted speedup of 1 indicates the same performance as the baseline system. The foreground and background workloads were both repeated at least five times to ensure their execution completely overlapped with each other. We selected *streamcluster* as the background workload for the PARSEC benchmarks and *UA* for the NBP benchmarks.

Figures 9, 10 and 11 show the weighted speedup of PARSEC and NPB benchmarks due to different scheduling strategies in native Linux, KVM and Xen, respectively. Our results indicated that P-MQFQ improved the system-wide weighted speedup by as much as 332%. In a physical environment, it achieved an average weighted speedup of 13% and 27.7% for PARSEC and NBP benchmarks, respectively. The overall performance improvement for both the foreground and background benchmarks were higher in virtualized environments (KVM and Xen). An examination of the foreground and background workloads revealed that the gain in system-wide weighted speedup was due to the performance improvement in both applications. Furthermore, both foreground and background applications had improved CPU utilizations.

Compared to P-MQFQ, the pinning mechanism either had marginal improvement on the weighted speedup or hurt the overall system efficiency. For example, pinning degraded the weighted speedup considerably for `x264` on physical machine, `ferret` on KVM and `dedup` on Xen. `Relaxed-Co` achieved better performance than pinning, but still hurt overall system efficiency when running `streamcluster` and `facesim`. `Gleaner` performed better in some cases than P-MQFQ such as `ferret` and `x264` on 24 cores KVM. Overall, P-MQFQ significantly outperformed these approaches in the average speedup over all workloads combinations.

4.5 Proportional Fair Sharing

Next, we evaluate the effectiveness of P-MQFQ in enforcing proportional fair sharing. We set larger weights for the foreground parallel program and use two metrics: program execution time and absolute relative lag, to quantify the benefit of P-MQFQ compared to other approaches. Table 1 compares P-MQFQ with `PIN+{Linux,`

	Pin + Linux (1:2)	Pin + Linux (1:4)	Pin + KVM (1:2)	Pin + KVM (1:4)	Pin + Xen (1:2)	Pin + Xen (1:4)
# of apps (maximum improvement)	Parsec: 7 (51%) NPB: 3 (12%)	Parsec: 7 (55%) NPB: 3 (14.2%)	Parsec: 4 (58%) NPB: 3 (35.9%)	Parsec: 3 (42.9%) NPB: 2 (61%)	Parsec: 9 (91%) NPB: 6 (92%)	Parsec: 9 (93%) NPB: 6 (89%)
Average performance improvement across all apps	Parsec: 14.8% NPB: -8.4%	Parsec: 16.3% NPB: -3.7%	Parsec: 6.9% NPB: 0%	Parsec: 6.8% NPB: -4.7%	Parsec: 67% NPB: -1%	Parsec: 68% NPB: -0.9%
Average fairness improvement across all apps	Parsec: 20% NPB: 18%	Parsec: 30% NPB: 19.6%	Parsec: 31.8% NPB: 4.3%	Parsec: 26% NPB: 10.6%	Parsec: 29.2% NPB: 32.5%	Parsec: 27.4% NPB: 41.4%

Table 1: Comparison of P-MQFQ and `PIN+{Linux, KVM and Xen}` with two weight settings.

`KVM and Xen}`. Due to limited space, we do not show the comparison with vanilla Linux and other approaches. Overall, P-MQFQ significantly outperformed these approaches in all benchmarks as bigger weights did not prevent CPU stacking. In contrast, bigger weights together with CPU pinning (e.g., `PIN+Linux`) effectively addressed deceptive idleness as the parallel thread in the critical section is less likely to be preempted. As shown in Table 1, P-MQFQ still outperformed PIN-based approaches in most cases. The first row in Table 1 lists the number of benchmarks in PARSEC and NPB in which P-MQFQ outperformed PIN-based approaches and the maximum improvement over PIN. Two weight settings: 1:2 and 1:4 (larger weights for parallel programs) were tested. One can observe that 1) PARSEC benchmarks benefited more from P-MQFQ as they suffer most from deceptive idleness due to fine-grained synchronization; 2) on average, P-MQFQ improved the performance of PARSEC benchmarks but incurred slight degradation to NPB benchmarks; 3) P-MQFQ achieved CPU allocations closer to the proportional fair share in both benchmarks. Among the three platforms, P-MQFQ attained the least fairness improvement for NPB benchmarks in `PIN+KVM`. KVM employs hardware-assisted CPU virtualization, e.g., Intel's VT-x, which causes expensive virtual CPU context switching due to blocking synchronization. Such expensive blocking hinders a parallel program from scaling its CPU utilization even with a larger weight. Compared to PARSEC, NPB benchmarks employ coarse-grained synchronization and suffer less from deceptive idleness, thereby leaving less room for CPU utilization improvement.

5 RELATED WORK

Packet Scheduling. Fair queuing algorithms have been extensively studied in the literature. Most algorithms use the generalized processor sharing discipline [28] as the reference model. Since GPS uses an idealized fluid model and cannot be implemented in real systems, various packet-based approximations of GPS have been developed [4–6, 13, 18, 20, 21, 24, 32, 39, 40, 42, 52, 53]. Among these algorithms, weighted fair queuing (WFQ) [13] also known as Packet-by-Packet Generalized Processor Sharing (PGPS) [39] has been considered to be the best approximation to the GPS with respect to the accuracy. WFQ will schedule the packets in the increasing order of their departure times in the GPS. However, WF^2Q [5] demonstrates that WFQ can be far ahead of GPS in terms of number of bits served for a session which could cause large discrepancies between the services provided by WFQ and GPS. In order to mitigate this unfairness of WFQ, WF^2Q only consideres the packets which have already started receiving service in GPS system. For integrated services networks (e.g., video and audio applications), start-time fair queueing [20] or SFQ showed it was better suited than WFQ to provide fairness over servers with time varying capacity. SFQ associated each packet with start tag and finish tag and scheduled these packets in the increasing order of start tag. Furthermore, SFQ changed the definition of system virtual time $v(t)$ to be the start tag of the packet in the service time t. SFQ(D) [24] is adaption of start-time fair queueing (SFQ) for servers with a configurable degree of internal concurrency and it makes it possible to share a server among multiple request flows. Hierarchical packet fair queueing [4] or H-PFQ proposed the WF^2Q+ algorithm which extended WF^2Q but with a lower complexity to provide the bounded-delay and fairness for hierarchical link sharing and traffic management between different service classes. All these fair sharing algorithms only provide methods for proportionally sharing a single server among competing flows. Unfortunately, they do not address the problem of fair sharing on multiple servers. MSFQ [6] extends WFQ to support multiple servers. It dispatches packets whenever there is an idle server and according to the order of packet completion under GPS. An important limitation of these packet-based fair queuing algorithms is that fairness is only guaranteed between backlogged flows and those that exhibit deceptive idleness are penalized under these algorithms.

CPU Fair Scheduling. Proportional share resource management provides a flexible and useful abstraction for multiplexing scarce resources such as the CPU, memory and disk among multiple users. The basic idea is to associate each user a weight and allocate resources to users in proportion to their weights. Many proportional fair share algorithms have been proposed to allocate CPU bandwidth [11, 14–16, 20, 22, 25–27, 29, 31, 33–36, 38, 41, 43, 44, 46–49, 51].

In uniprocessor systems, early CPU schedulers [22, 26] are based on the concept of task priority and failed to precisely control CPU allocation in proportion to user weights. Lottery scheduling [46] is a randomized resource allocation mechanism based on the notion of a ticket. Compared to the traditional priority-based schedulers, lottery scheduling approximates proportional fair sharing in the long-term scheduling. Stride scheduling [47] further improves over lottery scheduling with significantly improved accuracy over relative throughput rates and less response time variability. The core allocation mechanism used by lottery and stride scheduling is based on WFQ or SFQ algorithms for packet scheduling. Hierarchical CPU scheduler [20] implemented the SFQ algorithm in multithreaded scheduling and could support hard and soft real-time, as well as best effort applications in a multimedia operating systems. Borrowed Virtual Time Scheduling [15] or BVT aimed to provide low-latency for real-time and interactive applications by introducing a latency parameter into SFQ algorithm. Virtual time round robin [34] combined the round robin and WFQ fair queueing algorithm to guarantee proportional allocation CPU resource among a set of clients.

In multiprocessor systems, GPS-based fair sharing algorithms can result in unbounded unfairness. Since a thread can only run on a single processor at a time, certain weight assignments are infeasible in multiprocessor environments, allowing some threads to lag arbitrarily behind other threads. To address this issue, a series of works [8–10] focus on adjusting infeasible weights to the closest feasible weights and employed variants of the SFQ algorithm to schedule threads under the new weight assignments. Surplus Fair Scheduling [8] or SFS schedules threads in the increasing order of service surplus, which is the difference between the amount of CPU service received by a thread and the service that the thread would have received under the generalized multiprocessor sharing. Deadline Fair Scheduling [9] ensures that the number of times a thread is scheduled on multiprocessors is proportional to its weight. Hierarchical multiprocessor scheduling (H-SMP) [10] employs a space scheduler to allocate an integer number of CPUs to each thread group and uses SFS to schedule individual threads within each group. While the approaches to address infeasible weights are complementary to our work, the thread schedulers in [8–10] do not take the dependency between threads of a parallel program into account, thereby unable to enforce program-level fairness. Take SFS for example, it tracks per-thread virtual time and allows threads to forfeit its CPU allocation as system virtual time advances. Since SFS and other similar approaches do not have a notion of program virtual time, they fail to compensate a parallel program that have threads suffering from deceptive idleness. In the worst case, a parallel program can lose concurrency (i.e., all threads stacking on a single core) and lags arbitrarily behind other programs. In contrast, P-MQFQ tracks per-program virtual time and allows threads from lagging programs to preempt threads from programs that received excessive CPU. This allows threads from a parallel program to be compensated if their sibling threads (not necessarily themselves) forfeited CPU.

6 DISCUSSION

Spinning workloads face a different challenge in multi-tenant systems. Futile spinning due to the preemption of the critical thread wastes the fair CPU share of the spinning workload. Since spinning workloads do not block, they do not suffer from deceptive idleness and thereby are unable to benefit from P-MQFQ. Recent advances in processor design allow the OS to detect excessive spinning through hardware-based techniques, such as pause-loop exiting (PLE), and to forcibly stop (blocks) a spinning thread. PLE is especially useful in virtualized environments, where the hypervisor is oblivious of spinning inside VMs. As such, spinning workloads will suffer deceptive idleness when threads are involuntarily put to sleep and thus they also can benefit from P-MQFQ.

Overhead Approximating a centralized request dispatch queue requires synchronization between CPUs. Our implementation of P-MQFQ on multicore schedulers incurs two types of overhead. First, to identify the thread that received most service, P-MQFQ needs to monitor the progression of virtual time on all CPUs and find the queue with the largest virtual time progression. This requires traversing all CPUs every 3ms. To isolate the overhead due to this operation, we compared program performance with and without P-MQFQ in solo mode, in which no thread migration is performed. We found approximating global virtual time incurs negligible overhead, adding an average of 1.3%, 3.7% and 2.4% overhead to execution time in Linux, KVM and Xen respectively.

Second, frequent thread migrations undermine cache locality and require an expensive run queue operation in multicore schedulers – double run queue locking, which locks the source and destination CPUs before a thread migration is completed. When deceptive idleness occurs, P-MQFQ effectively improves CPU allocations to parallel programs, though inevitably incurs overhead. We measured the migration cost and cache miss ratio which are introduced by the three operations of P-MQFQ in all experiments, including those with the synthetic benchmark and with a realistic parallel workload, in Linux, KVM and Xen. Migration cost equals the total number of threads migration multiplied by time needed to move one thread from one core to another. Experimental results show that the migration cost of all threads in applications from PARSEC and NPB benchmarks range from 0.014s to 0.02s which is only 0.005% to 0.2% of total execution time. Cache miss ratio represents the total cache misses divided by the issued cache references. Experimental results show that cache miss ratio caused by P-MQFQ is less than default Linux by 0.2% to 2% which indicates P-MQFQ did not bring high memory cost through thread migration. It is also important to study which factor, improving utilization or expensive thread migration, weighs more in overall performance. We consider that the overhead of thread migration dominates overall performance if P-MQFQ outperforms `Pin` in CPU utilizations but results in less performance improvement. For example, P-MQFQ outperformed `Pin` in CPU utilization (10% vs. 2.5%) but achieved less performance improvement than `Pin` (26.4% vs. 66.9%) in KVM. In a physical environment, 44 out of 48 PARSEC tests and 33 out of 36 NPB tests show that the benefit of P-MQFQ outweighed its overhead. Similar observations were also made in KVM and Xen.

7 CONCLUSION

This paper identifies an important deficiency in state-of-the-art multicore schedulers that causes unfair CPU allocation to parallel programs with blocking synchronization and leads to severe performance degradation. This deficiency will seriously hamper CPU multiplexing in shared services, such as public clouds. We attribute the deficiency to the inability of existing schedulers to deal with deceptive idleness and the lack of multi-queue fair queuing in the context of thread scheduling. To this end, we proposed preemptive multi-queue fair queuing (P-MQFQ), a centralized algorithm that uses thread preemption to guarantee fair CPU allocation for multi-threaded programs on multiple CPUs. P-MQFQ can be approximated by augmenting distributed queue-based schedulers with three run queue operations. Results show that P-MQFQ improves utilization and performance compared to three representative scheduling strategies.

8 ACKNOWLEDGEMENT

We thank our shepherd Giuliano Casale and the anonymous reviewers for their insightful suggestions. This research is supported by National Science Foundation under grant CCF-1845706.

REFERENCES

[1] J. Ahn, C. H. Park, and J. Huh. Micro-sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (Micro-47)*, 2014.

[2] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91)*, 1991.

[4] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithm. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'96)*, 1996.

[5] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of 15th International Conference on Networking for Global Communications (INFOCOM'96)*, 1996.

[6] J. M. Blanquer and B. Ozden. Fair Queueing for Aggregated Multiple Links. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'01)*, 2001.

[7] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Operating System Review*, 13(2), 1979.

[8] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)*, 2000.

[9] A. Chandra, M. Adler, and P. Shenoy. Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *Proceedings of 7th IEEE Symposium on Real-time Technology and Applications (RTTAS'01)*, 2001.

[10] A. Chandra and P. Shenoy. Hierarchical Scheduling forSymmetric Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems.*, 19, 2008.

[11] L. Cheng, J. Rao, and F. C. M. Lau. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, 2016.

[12] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of Symposium on Communications Architecture and Protocals (SIGCOMM'89)*, 1989.

[14] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the Blockwaiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*, 2014.

[15] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'99)*, 1999.

[16] R. Essick. An Event-based Fair Share Scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, 1990.

[17] T. Friebel and S. Biemueller. How to Deal With Lock Holder Preemption. In *Xen Developer Summit*, 2008.

[18] S. J. Golestani. A Self-clocked Fair Queueing Scheme for Broadband Applications. In *Proceedings of 13th International Conference on Networking for Global Communications (INFOCOM'94)*, 1994.

[19] Google Cloud Platform. http://cloud.google.com/compute.

[20] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Scheduling Networks. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'96)*, 1996.

[21] A. G. Greenberg and N. Madras. How Fair is Fair Queueing. *Journal of ACM*, 39, 1992.

[22] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63, 1984.

[23] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[24] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for Storage Service Utility. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, 2004.

[25] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC'06)*, 2006.

[26] J. Kay and P. Lauder. The Fair Share Scheduler. *Communications of the ACM*, 31, 1988.

[27] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-Based Coordinated Scheduling for SMP VMs. In *Proceedings of 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013.

[28] L. Kleinrock. *Queueing System*. 1975.

[29] T. Li, D. Baumberger, and S. Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round Robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Computing (PPoPP'09)*, 2009.

[30] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible collocations. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (Micro-44)*, 2011.

[31] I. Molnar. Completely Fair Scheduler. In *Linux Journal*, 2009.

[32] J. Nagle. On Packet Switches with Infinite Storage. In *RFC 970, FACC Palo Alto*, 1985.

[33] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.

[34] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time-Round-Robin: An O(1) Proportional Share Scheduler . In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC'11)*, 2011.

[35] S. Orathai and K. S. Hyong. Is Co-Scheduling Too Expensive for SMP VMs? In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, 2011.

[36] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the IEEE Distributed Compute System*, 1982.

[37] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE'13)*, 2013.

[38] S. Panneerselvam and M. M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, 2012.

[39] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks: the Single-node Case. *ACM Transaction on Networking.*, 1, 1993.

[40] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks: the Multiple Node Case. *ACM Transaction on Networking.*, 2, 1994.

[41] J. Rao, K. Wang, X. Zhou, and C. Xu. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, 2013.

[42] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'95)*, 1995.

[43] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12))*, 2012.

[44] I. Stoica, H. Adbel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of Multimedia Computing and Networking*, 1996.

[45] The Princeton Application Repository for Shared-Memory Computers (PARSEC) . http://parsec.cs.princeton.edu/.

[46] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation (OSDI'94)*, 1994.

[47] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Determinstic Proportional-share Resource Management. In *MIT Technical Report*, 1995.

[48] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06))*, 2006.

[49] C. Weng, Z. Zhang, M. Li, and X. Lu. The Hybrid Scheduling Framework for Virtual Machine Systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE'09)*, 2009.

[50] Windows Azure Open Cloud Platform. http://www.windowsazure.com.

[51] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: Latency-aware Virtual Machine Scheduling via Differentiated-frequency CPU Slicing . In *Proceedings of the 21th International Symposium on High Performance Parallel and Distributed Computing (HPDC'12))*, 2012.

[52] H. Zhang and S. Keshav. Comparison of Rate-Based Service Dispilines. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'91)*, 1991.

[53] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'90)*, 1990.