GVTS: Global Virtual Time Fair Scheduling to Support Strict Fairness on Many Cores

Changdae Kim, Seungbeom Choi, Jaehyuk Huh, *Member, IEEE,*

Abstract—Proportional fairness in CPU scheduling has been widely adopted to fairly distribute CPU shares corresponding to their weights. With the emergence of cloud environments, the proportionally fair scheduling has been extended to groups of threads or nested groups to support virtual machines or containers. Such proportional fairness has been supported by popular schedulers, such as Linux Completely Fair Scheduler (CFS) through virtual time scheduling. However, CFS, with a distributed runqueue per CPU, implements the virtual time scheduling *locally*. Across different queues, the virtual times of threads are not strictly maintained to avoid potential scalability bottlenecks. The uneven fluctuation of CPU shares caused by the limitations of CFS not only violates the fairness support for CPU assignments, but also significantly increases the tail latencies of latency-sensitive applications. To mitigate the limitations of CFS, this paper proposes a *global virtual-time fair scheduler (GVTS)*, which enforces global virtual time fairness for threads and thread groups, even if they run across many physical cores. The new scheduler employs the hierarchical enforcement of target virtual time to enhance the scalability of schedulers, which is aware of the topology of CPU organization. We implemented GVTS in Linux kernel 4.6.4 with several optimizations to provide global virtual time efficiently. Our experimental results show that GVTS can almost eliminate the fairness violation of CFS for both non-grouped and grouped executions. Furthermore, GVTS can curtail the tail latency when latency-sensitive applications are co-running with batch tasks.

Index Terms—Proportional Fairness, CPU Scheduling, Group Fairness, Tail Latency

1 Introduction

PROPORTIONAL fairness in CPU scheduling mandates that the CPU shares of threads must be proportional to their assigned weights. It has been widely adopted by general purpose systems, as the de facto fairness support. With the popularity of system consolidation for clouds, proportional fairness has been further extended to enable weighted fairness among groups of threads or nested groups. Such extension of weighted fairness for thread groups is essential to support fair CPU assignments for containers or virtual machines served for different clients.

In such cloud-based computing models, the requirements for fairness support have become stricter than conventional native systems. Each user must receive the CPU share mandated by a service-level agreement (SLA). The user's container or virtual machine commonly consists of multiple threads or virtual CPUs, and thus the CPU share must be specified collectively at the group-level. However, in such clouds, heterogeneous workloads from different users share a physical system, with fluctuating loads. Furthermore, for latency-sensitive server workloads, fairness violation often causes significant increases of tail latencies, degrading the quality-of-service.

To implement the proportional fairness, a common mechanism is to maintain *virtual time* for each thread or group. The popular Completely Fair Scheduler (CFS) is designed to provide proportional fairness by scheduling based on such virtual time [1]. For each runqueue, CFS schedules runnable threads in a manner to equalize their virtual time progresses. CFS has also been extended to provide group fairness. A group with its own virtual time is inserted into the runequeue as a scheduling entity, and the group entity has a separate runqueue containing its member threads. In the hierarchical design, a group is selected, and then a thread from the group is selected based on the virtual time status.

However, CFS with per-CPU runqueues, implements the virtual time scheduling *locally*. Across different queues, the virtual times of threads are not strictly equalized to avoid potential scalability bottlenecks for enforcing *global* virtual time scheduling. Instead, a simpler load balancing mechanism distributes threads across multiple CPUs, and it provides approximate fairness for threads and thread groups running across multiple CPUs.

This paper investigates the limitation of the local virtual time scheduling in CFS. When the number of active threads is not always a multiple of physical cores, the proportional fairness is not strictly supported by the current CFS implementation. Such fairness violation gets worse with weighted fairness supports for groups. Furthermore, the CPU share of a thread may fluctuate significantly, depending on the decision of the load balancing mechanism. Such unfair fluctuation of CPU shares not only violates the SLA for CPU assignments, but also significantly increases the tail latencies of latency-sensitive applications.

To mitigate the limitations of CFS, this paper proposes a *Global Virtual Time Fair Scheduler (GVTS)*, which enforces global virtual time fairness for threads and thread groups, even if they run across many physical cores. The new scheduler employs topology-aware enforcement of target

• *Changdae Kim, Seungbeom Choi and Jaehyuk Huh are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, 305-701.*
E-mail: cdkim@calab.kaist.ac.kr, sbchoi@calab.kaist.ac.kr and jhhuh@kaist.ac.kr

virtual time to support the scalability of schedulers. Using the scalable global virtual time accounting, GVTS provides consistent CPU shares for threads even if they migrate across different CPUs.

We implemented GVTS in Linux kernel 4.6.4 with several optimizations to provide global virtual time efficiently. Our experimental results show that GVTS can almost eliminate the fairness violation of CFS for both non-grouped and grouped configurations. Furthermore, GVTS can effectively curtail the tail latency when latency-sensitive applications are co-running with batch tasks.

The followings are the new contributions of this paper:

- This study identifies the limitation of the current local virtual time scheduling in CFS. It shows that proportional fairness is violated when the number of threads is not a multiple of physical CPUs or CPU utilizations fluctuate.
- It proposes a new global virtual time scheduling. To efficiently support virtual time globally, it proposes a topology-aware balancing mechanism, which is aware of CPU interconnection topology.
- It improves proportional fairness support for thread groups, so that virtual machines and containers in clouds are provisioned as the SLA mandates.
- It investigates several optimizations to reduce unnecessary thread migrations which may incur by global enforcement of virtual time.

The rest of the paper is organized as follows. Section 2 discusses the limitations of CFS with its local virtual time tracking. Section 3 presents the design of global virtual time scheduling, and Section 4 discusses its implementation issues. Section 5 presents the experimental results. Section 6 discusses the related work and Section 7 concludes this paper.

2 Motivation

2.1 Virtual Time Based Proportional Fairness

Proportional fairness is a widely adopted definition of fairness in CPU scheduling. It was first proposed in the context of network flow control [2], and later adopted for CPU scheduling [3], [4], [5]. To use a proportional fair scheduler, all threads are assigned with their corresponding weights. The weight of a thread is the relative amount of CPU share it is entitled to receive. *A scheduling is fair if all threads receive their CPU shares in proportion to their weights.*

In real systems, multiple threads are assigned to CPUs in a time-sharing manner, and CPU share is represented as the received CPU time. Let $share_i(t_1, t_2)$ be the CPU time thread i receives between time t_1 and t_2. If there are n runnable threads between time t_1 and t_2, and their weights are $w_1, w_2, ...w_n$, the following equation represents the condition of proportional fairness, where *#CPUs* is the number of CPUs in the system.

$$share_i(t_1, t_2) = \frac{w_i}{\sum\limits_{j=1..n} w_j} \times (t_2 - t_1) \times \#CPUs$$

In addition to per-thread scheduling supports, threads can be grouped to provide group-level scheduling weights. A container can contain multiple threads with their own weights specifying proportional fairness within the container. The container itself has its own weight to specify its weight across different containers. Such thread grouping can be nested. A thread group can contain multiple inner thread groups. With thread groups, schedulers should support inter-group proportional fairness as well as intra-group proportional fairness. Let W_k be the weight of group k and $SHARE_k(t_1, t_2)$ be the sum of the CPU time group k's threads receive. For brevity, suppose that groups are not nested and every thread belongs to a group. The following equation represents the condition of proportional fairness for N groups and their threads.

$$SHARE_k(t_1, t_2) = \frac{W_k}{\sum\limits_{l=1..N} W_l} \times (t_2 - t_1) \times \#CPUs$$

$$share_{i_k}(t_1, t_2) = \frac{w_i}{\sum\limits_{j \in group_k} w_j} \times SHARE_k(t_1, t_2)$$

However, schedulers cannot provide perfect proportional fairness due to two reasons. First, in real systems, CPU time cannot be divided infinitesimally. The minimum scheduling quantum is restricted to be a multiple of timer interrupt interval, which is 1ms∼10ms on most systems, unless threads voluntarily yield their running CPUs. For this reason, *Lag Time* is defined as follows to present the difference between the ideal CPU time (share) and actual received CPU time of a thread between time t_1 and t_2 [6].

$$Lag_i(t_1, t_2) = share_i(t_1, t_2) - receivedCPUtime_i(t_1, t_2)$$

Second, if a thread has a very large weight, perfectly fair scheduling may not be achievable. For example, suppose that there are two CPUs and the weight of one thread is larger than the sum of the weights of all the other threads. By the definition of proportional fairness, the thread with the largest weight should receive more than a half of the total CPU share in the two-CPU system. Since a thread cannot consume more than one CPU at once, the thread with the largest weight cannot receive the share mandated by its weight. This phenomenon has been called *infeasible weight* problem [5], [7], [8]. If the weight of a thread satisfies the following condition, no scheduler can maintain perfect proportional fairness.

$$\frac{w_i}{\sum\limits_{j=1..n} w_j} > \frac{1}{\#CPUs}$$

To realize proportional fairness in CPU scheduling, several methods have been proposed, as we will explain in section 6.1. Among them, *virtual time* based proportional fair scheduling is widely used, and the Linux kernel scheduler adopts it [1], [3], [4], [5].

Virtual time is defined as the received CPU time divided by the weight of a thread. With the notations above, *virtual time* is represented as follows.

$$vtime = receivedCPUtime_i / w_i$$

Then, *virtual time fair scheduling* maintains the virtual times of all threads as similar as possible. If the virtual times of two threads are equal, their received CPU times are exactly proportional to their weights. Thus, proportional fairness can be maintained by virtual time based scheduling.

2.2 Local Virtual Time Fair Scheduling in CFS

Completely Fair Scheduler (CFS) [1] has been the mainline Linux scheduler since 2007. It provides thread-level and group-level proportional fairness based on virtual time fair scheduling. However, virtual time based scheduling is used for only local scheduling within a CPU as CFS maintains a separate runqueue for each CPU. For global scheduling across cores, a load balancing mechanism is used to improve the scalability.

At each CPU, CFS maintains a runqueue for the CPU, and threads are inserted to the queue as scheduling entities. CFS maintains *virtual time* of all threads, and always picks the thread with the lowest virtual time. CFS runs the picked thread for a while and updates its virtual time with the CPU time received by the thread. After CFS compares the updated virtual time to those of the other threads, if the current one is not the lowest one anymore, CFS picks the next thread to run. This mechanism forces the virtual times of all threads to make approximately equal progresses and proportional fairness is maintained within the runqueue. To implement the runqueue, CFS uses a red-black tree for its efficient time complexity. By using virtual time as the key of the red-black tree, picking the lowest virtual time thread can be done in $O(1)$ time, and re-inserting the previous thread to the tree is done in $O(logN)$ time.

To support group-based proportional fairness, CFS implements a hierarchical scheduling mechanism. A group of threads is considered as a unit for scheduling, being inserted as a scheduling entity to the CPU's runqueue. Each group maintains its own runqueue for the threads in the group. Since a group also has its weight, the virtual time of the group is defined similarly to threads. If CFS picks a scheduling entity from the CPU's runqueue and the selected entity is a thread group, CFS picks a scheduling entity from the group's runqueue. Since CFS allows nested groups, this selection process is repeated until a leaf thread is selected. This scheduling makes the virtual times of groups progress in equal paces to support inter-group proportional fairness. In addition, since CFS makes the virtual times of threads for each group progress similarly, intra-group proportional fairness is also maintained.

However, virtual time fair scheduling of CFS is restricted to each CPU, enforcing strict virtual time maintenance only within the runqueue of a CPU. CFS does not support accurate virtual time accounting globally across multiple CPUs to avoid a potential scalability problem of tracking accurate global virtual time across multiple runqueues. When a thread migrates to another CPU for load balancing, its accurate virtual time is not transferred to the new CPU, maintaining only approximate relative information regarding the virtual time of the migrated thread.

2.3 Inaccuracy of Virtual Time Tracking with CFS

Without strict virtual time accounting across CPUs, CFS uses its load balancing mechanism for global scheduling. The *load* of a thread is defined as the thread's weight multiplied by the thread's CPU utilization, and the *load* of a CPU is defined as the sum of thread loads in the CPU's runqueue. Since the CPU utilization of a thread represents the time portion when the thread is in a runnable state, *load* of a CPU represents the expected sum of its own active threads' weight. Thus, if the loads of two CPUs are equal, for a given time, the amounts of total virtual time increase per CPU are equal. CFS periodically compares CPU loads and migrates threads to balance the loads. This makes local fair scheduling at each CPU leads to approximate global fair scheduling across CPU cores.

To support global proportional fairness for groups, CFS uses *hierarchical load balancing*. The load of a group is defined as the group's weight multiplied by the group's CPU utilization. The *hierarchical load* of a thread is the portion of group's load contributed by the thread. The sum of hierarchical loads of all the threads in the group is the group's load. The same core mechanism is used as *load balancing*, but each thread uses its *hierarchical load* instead of *load*.

The load balancing mechanism provides high scalability as the runqueue for each CPU is managed independently. However, load balancing does not guarantee accurate proportional fairness, since loads can be balanced only at thread granularity. For example, if there are two CPUs and three threads with equal loads, loads cannot be balanced accurately between CPUs. In such situations, local virtual time fair scheduling with load balancing does not lead to globally fair scheduling.

Although such approximate fair scheduling of CFS has been effective enough for private systems, the recent trends of consolidation in clouds require more strict proportional fairness support. In such consolidated systems with multiple heterogeneous applications which have fluctuating CPU utilizations, the imperfect fairness support can cause inconsistent CPU shares among threads, incurring severe variances in latencies for server workloads. Such problems exacerbate as multiple clients share a physical system either by containers and virtual machines. The next section quantitatively analyzes the limitation of CFS.

2.4 Impact of the Fairness Limitations in CFS

In this section, we show the impact of unfairness in load balancing on application performance. We use a 16-core system, and the details of the experimental setup are presented in Section 5.1. We measure the effect of imperfect CPU resource accounting of CFS using a synthetic benchmark and real applications. The synthetic benchmark is a multi-threaded application, and each thread consumes CPU continuously. It measures the received CPU time for every second. We implemented the measurement part of the benchmark as simple as possible to avoid disk I/O, page fault, interactions with kernels or other processes, and any other side effects that may influence performance. As the real applications, we use two representative types of applications: the throughput-oriented batch tasks that continuously use CPUs, and the latency-sensitive server programs with fluctuating CPU loads.

First, we use a synthetic benchmark to measure the CPU share distribution by CFS, when perfect load balancing is difficult to achieve. Figure 1 shows the results with 20 threads on 16 cores. The left graph shows the received

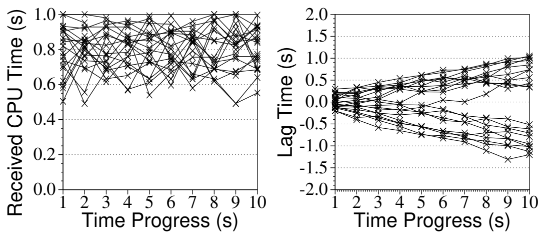

Fig. 1: Synthetic benchmark: 20 threads on 16 cores

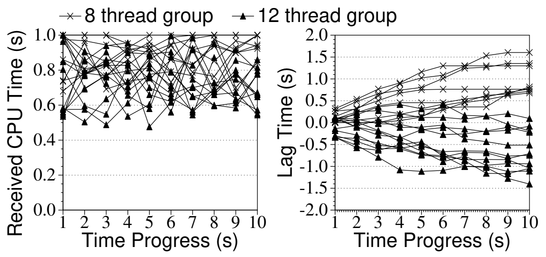

Fig. 2: Synthetic benchmark: 8-thread and 12-thread groups

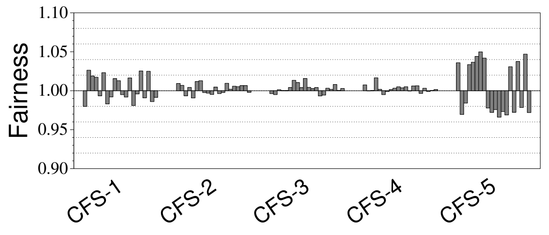

Fig. 3: 20 copies of *namd* on 16 cores

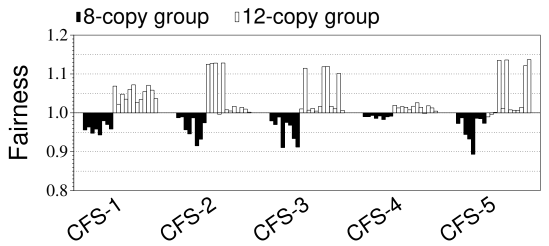

Fig. 4: 8-copy and 12-copy groups of *namd* on 16 cores

CPU time measured in each second. Since we set all the threads to have an equal weight, with the ideal scheduling, threads should receive 0.8 seconds of CPU time per second. However, the received CPU time fluctuates significantly between 0.5∼1.0 seconds. In addition, the cumulative lag times shown in the right figure increase significantly as time progresses. After 10 seconds, some threads receive 1.0 second less CPU time compared to that of ideal scheduling, while some threads receive 1.2 seconds more CPU time than that with an ideal scheduling.

Figure 2 shows the results with thread grouping. There are 20 threads, but 8 threads and 12 threads are grouped separately. We set the weights of two groups equally, and the weights of all the threads within a group are also equal. With the ideal inter-group proportional fairness, each group should receive 8 cores. Therefore, the threads in the 8-thread group should receive 1.0 second for each second, and the threads in the 12-thread group should receive 0.66 seconds for the same time period. However, the received CPU times fluctuate regardless of thread groups, and differences in lag times are larger than the non-grouped case.

The main reason of such unfair CPU allocation is due to the failure in load balancing. Since there is no way to equally distribute 20 threads on 16 cores, load balancing does not provide inter-runqueue fairness. With thread grouping, the loads become more complicated to be equalized by the coarse-grained load balancing approach of CFS.

As the first type of real applications, we use a CPU intensive workload, *namd* from SPECCPU2006 benchmark suite [9], to show the impacts of unfair CPU allocation on the batch jobs. To evaluate the scheduling, we define *fairness* as the normalized performance to the performance with ideal scheduling as follows.

$$fairness = \frac{\text{actual perf}}{\text{ideal perf}} = \frac{\text{actual perf}}{\frac{\text{ideal share}}{\text{share of solo run}} \times \text{solo run perf}}$$

Since we use CPU bound workloads, the ideal performance can be calculated from the solo run performance and CPU share with the ideal scheduling. For *namd*, the performance is defined as the reciprocal of execution time.

The experiment scenarios are similar to the previous cases for the synthetic benchmark and the experiments are repeated 5 times. Figure 3 shows the results with 20 copies of *namd*, running 20 threads on the 16-core system, and Figure 4 shows the results with thread grouping. 8 copies and 12 copies are grouped separately. The results were similar to the ones with the synthetic benchmark. The actual performance of *namd* varies from the ideal performance, up to 5% in the non-grouped scenario and up to 13% in the grouped scenario.

Note that the performance variance of *namd* copies is not as large as that with the synthetic benchmark. The main reason is the randomized scheduling effect. Since there are many kernel threads or service threads which wake up occasionally, the load distribution in the system changes and the load balancing tries to re-balance loads again. Then, *namd* threads that have received less CPU shares can have chances to receive more CPU shares later. Even with the long-term randomization of loads, the grouped scenario shows worse fairness than the non-grouped one since the group weights are not accurately accounted with the current CFS implementation. It can lead to potential violation of service-level agreement for CPU resources, where the weights of virtual machines or containers are contracted for each user.

Finally, to show the impacts of unfair scheduling on server programs, we use four server workloads from TailBench [10]. Since server programs are latency-critical workloads, we use two tail latencies, 95%-tile and 99%-tile latencies, as performance metrics.

Figure 5 shows 95%-tile and 99%-tile latencies of the server workloads when batch jobs share the same system. We use 16 copies of *namd* for the batch job, which can fill all 16 CPUs. With 16 batch jobs, the shares of batch jobs are varied to measure the response time changes of the latency-sensitive workloads. In the figure, the x-axis represents the batch job share varied from 0 to 90%. We group the threads of server workloads and the threads of batch jobs separately, and set the group's weight to adjust the ideal CPU share ratio. The threads in each group have an equal weight. The

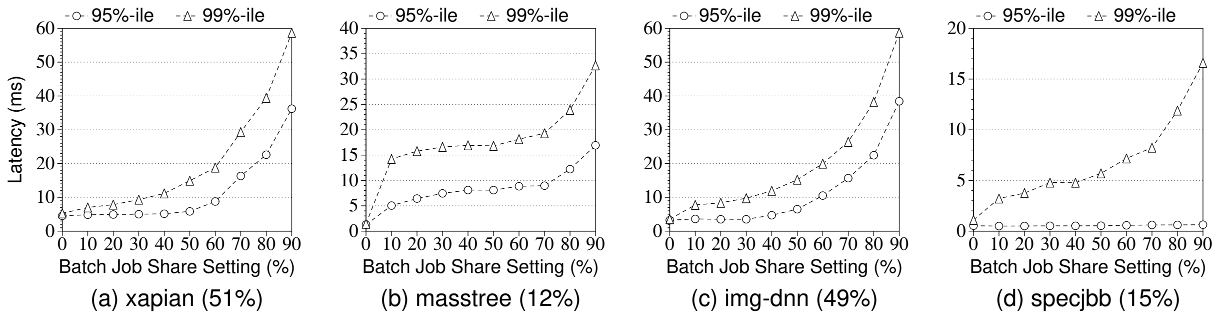

Fig. 5: Tail-latency of TailBench with batch jobs. The percentage in the parentheses represents CPU utilization in a solo run.

y-axis shows the tail latencies in ms.

The percentage after the workload name presents CPU utilization of the workload when no batch jobs are running. For example, `(b) masstree` uses only 12% of CPUs when running alone. Thus, ideally, the tail latencies of latency-sensitive workloads should not be affected by batch jobs until the share setting exceeds the required CPU utilization of latency-sensitive workloads. However, as shown in the figure, tail latencies increase significantly even when the batch job share is set to be relatively low. As exemplified in `(b) masstree`, even if the latency-sensitive workload requires only 12% of CPUs, when the share of the batch group exceeds 10%, the 99%-tile latency jumps to 14ms. The result shows that even short-term unfairness can lead to a significant quality-of-service degradation.

In conclusion, we show that CPU resource accounting of CFS can be unfair if the number of threads is not a multiple of the number of CPUs. For long-running batch jobs, the impact of unfair accounting can be amortized by the randomized scheduling effect. However, for server workloads, the temporal unfairness in CFS significantly affects the performance consistency across many responses, causing significant increases of the tail latencies.

3 Global Virtual Time Fair Scheduler

This section describes the design of GVTS (Global Virtual Time Fair Scheduler) in three parts. First, Section 3.1 describes *topology-aware global virtual time balancing*, which is a scalable mechanism to provide thread-level proportional fairness with high accuracy. Second, Section 3.2 explains how GVTS supports inter-group proportional fairness. Third, Section 3.3 proposes optimization techniques to reduce thread migration overheads.

3.1 Topology-aware Global Virtual Time Balancing

For GVTS, the main difference from CFS is that virtual time is a global value for all CPUs in a system. GVTS makes *virtual time* of all threads globally progress equally. This enables the progress of all threads to be fair according to their weights.

At each CPU, GVTS is very similar to CFS. It always picks up a thread with the lowest *virtual time*. It runs the thread until the *virtual time* of the thread is not the lowest one anymore. Then, it selects the next thread with the lowest *virtual time*, and the procedure is repeated.

To make virtual time progress fair between threads in a system, GVTS uses *global virtual time balancing*. The mechanism sets *target virtual time* for CPUs, and the target value works as a barrier for *virtual time* progress. When the virtual times of all threads in a CPU exceed the target, the CPU stops picking up a thread in the runqueue. Instead, it scans the other CPUs to find threads whose *virtual time* do not exceed the target. If such threads are found, it pulls and runs the threads. Otherwise, if there are no such threads, the CPU increases the target as much as *target interval*. Then, it runs its own threads until all the threads pass the new target.

Determining *target interval* is an important issue for GVTS. A short interval incurs frequent balancing and it may result in the performance degradation due to frequent thread migration. At the same time, short intervals keep the fairness among threads at fine granularity, as such fine-grained barriers minimize the difference in *virtual time* across all threads. On the other hand, long intervals reduce the thread migration overhead, but increase temporal unfairness among threads.

To balance the trade-off between the migration overhead and fairness, GVTS employs *topology-aware global virtual time balancing*. Based on the CPU topology, GVTS builds multi-level scheduling domains and sets a different *target interval* for each level of domains. For example, suppose that there is a many core system with multiple NUMA nodes and each node has several cores with SMT (Simultaneous Multi-Threading) support in each core. In the system, the logical SMT CPUs within a physical core constitutes a scheduling domain (SMT domain), which is the lowest level domain. In addition, several SMT domains in the same NUMA node constitute the next level domain. The domain of a NUMA node has the SMT domains as its children. Finally, all the NUMA nodes constitute the highest level scheduling domain.

The thread migration overheads also depend on the CPU topology. Migrating threads between the logical SMT CPUs in a core incurs little overheads, while migrating threads between NUMA nodes incurs large overheads. Therefore, we set a short interval for the scheduling domain with less migration overheads, and a long interval for the scheduling domain with larger migration overheads. This mitigates the overall thread migration overhead, while reducing temporal unfairness. Note that the lower level of scheduling domains have short intervals. The fairness between threads is maintained at fine granularity within the domains.

Figure 6 summarizes the overall procedure of *topology-aware global virtual time balancing*. When all the threads in a

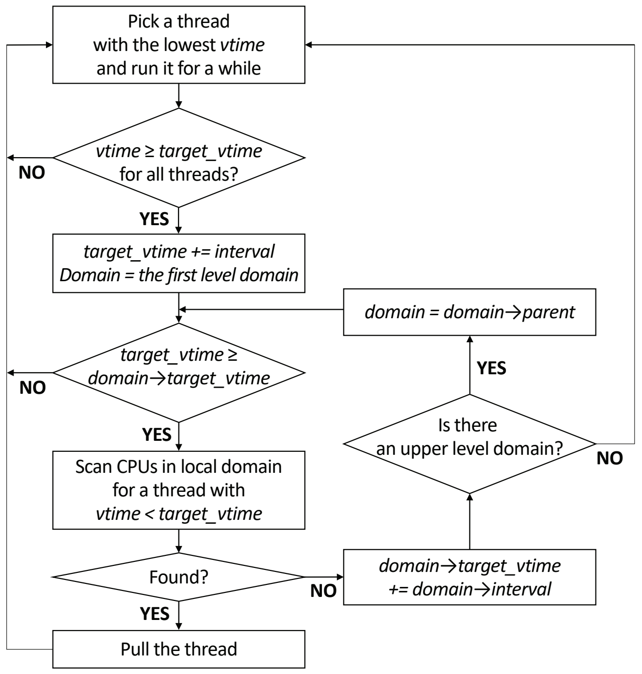
Pick a thread with the lowest vtime and run it for a while
vtime ≥ target_vtime for all threads?
NO
YES
target_vtime += interval
Domain = the first level domain
domain = domain→parent
target_vtime ≥ domain→target_vtime
NO
YES
YES
Is there an upper level domain?
NO
Scan CPUs in local domain for a thread with vtime < target_vtime
Found?
NO
domain→target_vtime += domain→interval
YES
Pull the thread

Fig. 6: Topology-aware global virtual time balancing

CPU pass the target virtual time, the CPU scans the CPUs in the lowest level domain to find other threads to pull. If there is no thread to pull, the target of the lowest level domain increases by the interval of the domain. Then, the scanning is extended to the higher level domain until any remote thread can be pulled or any local thread does not pass the target virtual time of the domain.

3.2 Effective weight for inter-group fairness

Topology-aware global virtual time balancing provides proportional fairness between threads for many cores. To support proportional fairness between groups, we propose an *effective weight* mechanism which also supports nested grouping.

The weight of a thread represents the desirable CPU share ratio of the thread. However, this is not true if threads are grouped since CPU share should first be distributed to groups according to the group's weight, and then CPU share for each group can be distributed to threads according to the thread's weight.

Thus, we propose *effective weight* to represent the desirable CPU share ratio for both grouped and non-grouped threads. For non-grouped threads, the effective weight is the same with its weight. For threads in a group, the effective weight of the thread, w^{eff}_{thread}, is defined as follows.

$$w^{eff}_{thread} = \frac{w_{thread}}{\sum\limits_{entity \in group} w_{entity}} \times W^{eff}_{group}$$

In the definition, *entity* is a thread or group which belongs to the group. Since GVTS supports nested grouping, a group can belong to another group. The effective weight of a group is defined in the same way. If a group does not belong to any other group, its effective weight is the same with its weight. Otherwise, the effective weight of the group is defined as follows.

$$W^{eff}_{group} = \frac{W_{group}}{\sum\limits_{entity \in parent} w_{entity}} \times W^{eff}_{parent}$$

where *parent* is the parent group.

Figure 7 shows an example of effective weight values. Rectangles represent groups, and ellipses represent threads. Suppose that the weight of all groups and threads are 1024. The effective weight of each entity is shown inside the rectangle or ellipse, and the CPU share ratio for a thread is shown under of the ellipse.

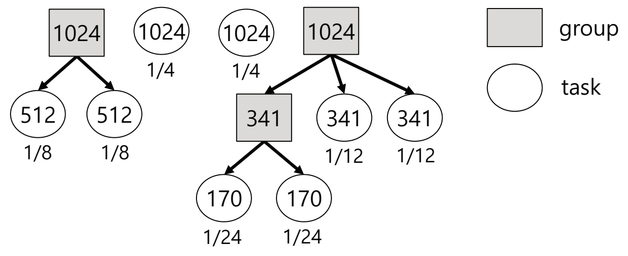

Fig. 7: An example of effective weight values

3.3 Optimizations to Reduce Thread Migrations

One negative effect of GVTS is that the number of thread migrations is likely to increase. When the effective weight of threads cannot be distributed equally among CPUs, threads jump from a CPU to another to receive fair amounts of CPU time. When a thread is migrated, it loses the data loaded on caches and other processor states such as TLB, branch predictor, etc, and its performance can be degraded.

Although *topology-aware global virtual time balancing* adjusts the thread migration frequency considering the thread migration overhead, we add two more optimizations to reduce throughput degradation by the thread migrations. First, when global virtual time balancing occurs, the scheduler attempts to make CPUs reach the next target virtual time as similarly as possible. This reduces the number of required thread migrations for the next virtual time balancing. Second, GVTS skips virtual time balancing if progress among CPUs are similar. This further reduces the thread migrations when the unfairness is negligible.

In the rest of this section, we first explain a technique, *remaining time estimation*, which is used for the optimizations, and then describe the optimization techniques in detail.

Remaining Time Estimation: For the optimization techniques, it is necessary to estimate the required CPU time of a thread to reach the next target. We call it *remaining time* or *remain_time* which represents the remaining time to reach the next target virtual time.

To estimate *remaining time*, we firstly define *remain_weight* of a thread, which represents the required CPU time to increase 1 virtual time, as follows.

$$w^{remain}_{thread} = w^{eff}_{thread} \times CPUutil_{thread}$$

Note that when a thread takes $1ms$ CPU time, its virtual time increases by $1ms/w^{eff}_{thread}$. In addition, the CPU utilization factor is included to account the virtual time increase only by receiving actual CPU time. However, when a thread wakes up from sleeping state, the virtual time of a thread is increased to a new value approximated to the current minimum virtual time in the system, to make the thread receive a fair amount of CPU share when it finally becomes active after the sleep state. The detailed mechanism will be explained in Section 4.

Using *remain weight*, the remaining time of a thread to reach the target can be estimated as follows.

$$remain\_time_{thread}(target) = (target - vtime_{thread}) \times w_{thread}^{remain}$$

The first part of the definition represents the remaining virtual time to the target. The target should be provided as an argument. The second part represents the estimated time to increase 1 virtual time. Thus, the multiplication of two terms represents the required CPU time to reach the target. Then, *remaining time* of a CPU can be defined as the sum of *remaining time* of threads in the CPU's runqueue. A CPU's *remaining time* represents the required CPU time to reach the target provided.

$$remain\_time_{CPU}(target) = \sum_{thread \in CPU} remain\_time_{thread}(target)$$

Optimizing virtual time balancing: When a CPU pulls a thread from another CPU for topology-aware virtual time balancing, this optimization attempts to balance the remaining times of CPUs to reach the next target. With the balancing, two CPUs are likely to reach the next target at a similar time, and less thread migrations are required later. Similar to *load* in CFS, *remaining time* can be changed only at thread granularity. In addition, since *remaining time* increases or decreases depending on the target or received CPU time, the remaining times of two CPUs are not exactly equal for most cases. However, unlike the load balancing of CFS, *remaining time* does not affect the fairness, as it can change only migration frequency. This optimization, as well as thread migration, does not change the virtual times of threads, and the fairness is maintained by equal progress in virtual time.

Skipping virtual time balancing: In the second optimization, the scheduler skips the balancing step if the progresses of CPUs are similar. If the remaining times of CPUs are similar, they will reach the next target at a similar time frame. Therefore, virtual time balancing is not necessary in this case. To determine whether the progresses of two CPUs are similar enough to skip the balancing step, we add a parameter, *tolerance*, which represents the allowed difference of *remaining time*. If the difference in remaining times is less than the *tolerance* value, the scheduler skips the balancing procedure, avoiding thread migration. While this optimization may slightly increase the unfairness of a system, the unfairness cannot increase boundlessly, since the scheduler skips the balancing only when CPUs progress similarly. Furthermore, this optimization also does not affect *virtual time* of threads, and the unfairness will be fixed soon by *topology-aware global virtual time balancing*.

4 IMPLEMENTATION

GVTS is implemented on Linux kernel 4.6.4 by modifying the CFS scheduler. Most of the codes related to load balancing are removed and replaced by *topology-aware global virtual time balancing*. Since our implementation is done within the Linux kernel interfaces, the KVM hypervisor or Dockers can be used with our implementation without modification. In addition, our implementation has the same level of portability as the Linux kernel and is able to recognize various system topologies as the Linux kernel can.

For the *target interval* parameters, we use the following values according to the resource sharing level of scheduling domains. First, for the SMT domain, *target interval* is set to 30ms. Then, for domains on the same chip, whose CPUs share a last level cache, *target interval* is set to 90ms. Finally, for NUMA nodes, *target interval* is set differently, depending on the distance, to *the number of hops* $* \ 300ms$.

For *tolerance* parameters, which is used for the optimization techniques, the value for each domain is set to 30% of *target interval* value of the domain. Setting the tolerance to the relative value to the target interval of each domain, allows temporal unfairness across higher level scheduling domains with large target interval values. At lower level scheduling domains, virtual time management is more strictly enforced for fine-grained proportional fairness support.

There are several implementation issues to maintain virtual time as global values. The rest of this section explains how to address the implementation issues.

Thread group management: To manage thread groups efficiently, GVTS modifies the hierarchical scheduling of CFS. In CFS, a group has its virtual time and an associated runqueue containing member threads with their own virtual time independent from the group virtual time. In GVTS, as the virtual times of threads are globally maintained, the virtual time of a group is just set to the minimum virtual time of member threads. Since the hierarchical scheduling selects the lowest virtual time entity in the runqueue, such setting makes the lowest virtual time thread to be selected regardless of the group hierarchy.

In addition, GVTS adds a shared variable within a group, the sum of weights of active threads and active child groups. This is used to calculate *effective weight*, as described in Section 3.2. Since the weights of threads rarely change, the weight variable needs to be updated only when a thread forks, exits, sleeps, or wakes up. It does not need to be updated when a thread migrates to another CPU. Due to such infrequent change, the shared variable per group does not incur any performance impact.

Virtual time of waking up task: A thread waking up from the idle state must be assigned with a new virtual time. In CFS, when a thread wakes up, the minimum value of virtual time in the local runqueue is used for the virtual time of the thread. The minimum virtual time setting gives the highest priority to the activated thread, and improves I/O latency if the thread had been in sleep state to wait for an I/O response.

To support such a mechanism in GVTS, it is necessary to maintain the globally minimum virtual time of all threads in the system, since virtual time is globally enforced in GVTS. Finding an accurate minimum virtual time may require global synchronization with high overheads. Instead, GVTS uses an estimated value for the globally minimum virtual time. It is estimated by the minimum target virtual time of the lowest level scheduling domains subtracted by a half of the target interval of the lowest level scheduling domains. This approximate virtual time assignment for a newly woken-up thread allows it to be selected as the next thread to run, and guarantees a certain amount of share, as its virtual times is smaller than the local minimum minus a half interval.

To efficiently maintain the minimum target virtual time of the lowest level scheduling domains, each scheduling domain maintains the minimum target virtual time of the child domains. When a scheduling domain updates its target virtual time, it also updates the minimum target virtual time of the parent domain if necessary. Then, the scheduler can find the minimum target virtual time with $O(logP)$ time complexity where P is the number of CPUs. In addition, the scheduler can skip the estimation if the thread already has a virtual time larger than the local minimum virtual time, further reducing the overhead.

Infeasible weight: Unlike CFS, infeasible weight incurs a negative side effect in GVTS. In CFS, threads with infeasible weights have an extraordinarily high *load*, and load balancing of CFS gives a whole CPU to each of the threads. Even though the definition of proportional fairness mandates the thread to receive more than a CPU, the CFS decision for the thread is proper, since a thread cannot receive more than a CPU. In addition, since CFS manages a runqueue in each CPU independently, there is no side effect.

Similar to CFS, a thread with an infeasible weight in GVTS receives a whole CPU, as the virtual time of the thread increases very slowly compared to the other threads and its *remaining time* becomes much larger than those of the other threads. However, the slow increase of *virtual time* becomes a significant problem in GVTS, since it prevents the global minimum virtual time from increasing. This leads to a problematic situation where a thread waking up has an abnormally low *virtual time*. The newly woken-up thread receives a large amount of CPU shares until it catches up the virtual times of the other long running threads, leading to unfair scheduling.

To address this problem, we implement an infeasible weight detection mechanism and exclude the CPU with the infeasible weight threads from maintaining the global minimum target virtual time. By this exclusion, threads waking up can have proper virtual time values, even when a thread with an infeasible weight exits in the system. The conditions for the detection is as follows. 1) A CPU is lagging behind the other CPUs for more than 3 intervals. 2) The CPU has only one thread. 3) The CPU's *remain_weight* is larger than those of the other CPUs. If all conditions are satisfied, the scheduler decides that the thread has an infeasible weight, and it excludes the CPU executing the thread from maintaining the minimum target virtual time.

5 Evaluation

5.1 Methodology

For evaluation, we use two systems with 16 cores and 80 cores respectively. The first system has an AMD Opteron 6282 SE processor with 32GB RAM. The processor has 16 cores (*16-core system*). Among 16 cores, each pair of cores shares an FPU, L1 instruction cache, and 2MB unified L2 cache. Each pair constitutes a level-1 scheduling domain. In addition, a group of 8 cores share an 8MB L3 cache, and they constitute a level-2 scheduling domain. Finally, two 8-core groups constitute the final level 3 domain. This system was also used in Section 2.4.

The second system has 4 Intel Xeon E5-4620 processors with 256GB RAM. Each processor has 20 cores and the total number of cores in the system is 80 (*80-core system*). In this system, a pair of logical CPUs is coupled by Hyper-Threading technology. For *topology-aware global virtual time balancing*, each pair of logical CPUs constitute a level-1 scheduling domain, the 20 cores in a chip constitute a level-2 scheduling domain, and four processor chips in the system constitute the highest level scheduling domain.

In this section, our GVTS implementation on Linux kernel 4.6.4 is compared with CFS and DWRR [8]. First, CFS is the virtual time based proportional fair scheduler for Linux kernel, as described in Section 2.2. For CFS evaluation, we use the same 4.6.4 kernel. Second, DWRR is a weighted round-robin based fair scheduler. For each *round*, a CPU schedules each local thread for $w_i \times$ *round_interval*. After finishing a round for the local threads, it scans CPUs at the lower round to pull their threads. This makes the progress of CPUs fair at round granularity, and the lag time of threads is kept within $w_i \times$ *round_interval*. For *round_interval*, we use *30ms* as in the DWRR paper. Note that DWRR has a trade-off between thread migration overhead and fairness granularity. If *round_interval* is short, the maximum lag time is reduced but threads can be frequently migrated across CPUs. We ported DWRR to Linux kernel 4.6.4 based on the original DWRR implementation on Linux kernel 2.6.24 [8]. However, as DWRR does not support thread grouping, we do not evaluate DWRR for the inter-group fairness.

We use several workloads to compare fairness and throughput of the schedulers. First, we use a synthetic benchmark which was described in Section 2.4. This is used to show the CPU share allocation in details. Second, to show the real impact of global fair scheduling on batch jobs, we use two single thread benchmarks and a multi-threaded benchmark. For the single thread workloads, we select *namd* and *milc* from SPECCPU2006 [9]. The former one is a CPU intensive benchmark and CPU allocation has a major impact on performance, while the latter one is a memory intensive benchmark and the migration overhead also has a significant impact on the performance. For the multi-threaded batch workload, we use *swaptions* of PARSEC [11], which exhibits high CPU utilization.

Last, we use two kinds of server workloads. *TailBench* suite [10] is used to test the schedulers with various server programs. It includes 8 real-world server workloads such as web search engine, in-memory and disk-based databases, face recognition, speech recognition, etc. In addition, we use *memcached* 1.4.25 which is a representative key-value store application. For the client driver, we use *treadmill* [12] to mimic a real usage scenario. We run the client driver on a remote server connected to our experimental systems through TCP/IP.

5.2 Results of Synthetic Benchmark

As described in Section 2.4, the synthetic benchmark creates the specified number of threads which continuously run for 10 seconds. Figure 8 shows the results when the benchmark creates 20 threads on 16-core system. The results show

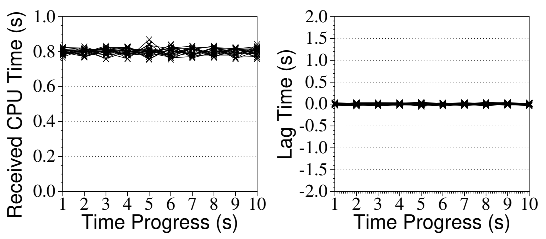

Fig. 8: Synthetic benchmark: 20 threads on 16 cores

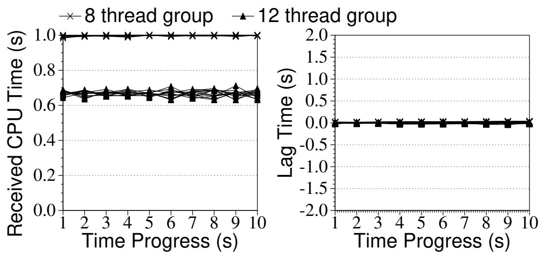

Fig. 9: Synthetic benchmark: 8-thread and 12-thread groups on 16 cores

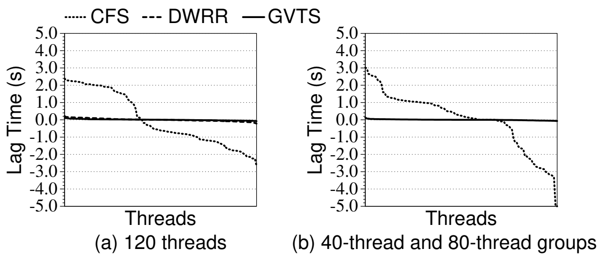

Fig. 10: Synthetic benchmark results on 80 cores

that, unlike the CFS results shown in Figure 1, GVTS can maintain inter-thread proportional fairness. As shown in the left graph, threads receive from 0.76 to 0.87 seconds of CPU share for each second. Since the ideal scheduling should distribute 0.8 seconds of CPU share to all threads for each second, this results is much closer to the ideal scheduling than the results of CFS. Moreover, temporal unfairness is being resolved over time as shown in the right graph. Lag times do not diverge and are maintained between -0.05∼0.04 seconds.

Figure 9 uses the same microbenchmark, but 8 threads and 12 threads are grouped separately. Comparing the CFS results in Figure 2, the results also show that GVTS provides much more accurate fairness than CFS. Lag times are near zero for all threads with GVTS.

Finally, Figure 10 shows the results on the 80-core system. Since there are many threads, we show the lag time distribution of threads after 10 seconds in the graph. In the figure, the x-axis represents the threads, and the y-axis represents the lag time in seconds. The three lines show the lag times of threads with CFS, DWRR, and GVTS respectively. When 120 threads run, the lag times with CFS increase up to 2.2 seconds. When the threads are grouped, the results of CFS become worse as shown in the right graph. Meanwhile, the curves for GVTS in the left and right graphs are close to zero, as its global virtual time fair scheduling distributes CPU share proportional to threads' weight. DWRR also shows the near-zero lag times thanks

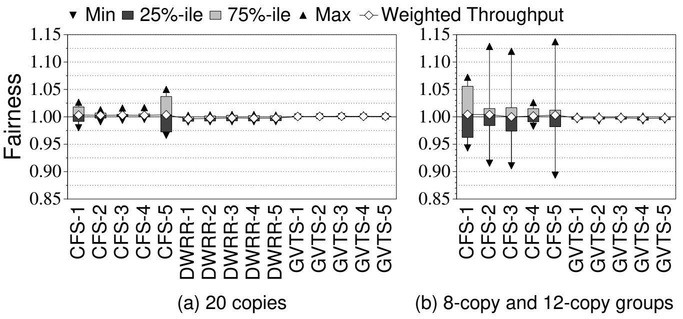

Fig. 11: Results of *namd* on 16-core system

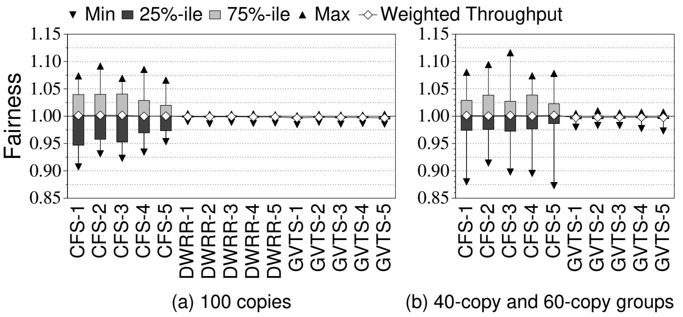

Fig. 12: Results of *namd* on 80-core system

to the round based fair scheduling. However, it does not support thread grouping, and the result is omitted from the right graph.

5.3 Results of Batch Jobs

In this section, we evaluate long time scheduler behaviors and their effect on performance. We use *namd* and *milc* from SPECCPU2006 [9] and *swaptions* from PARSEC [11]. For benchmarks from SPECCPU2006, *reference input* is used for all scenarios. Note that *namd* is a CPU intensive benchmark while *milc* is a memory intensive benchmark. For *swaptions*, *native input*, 128 point dimensions with 1,000,000 input points, is used on 16-core system. We increase input size to 160 points dimensions with 10,000,000 input points for 80-core system to make sure that the benchmark sufficiently utilizes 80 cores.

We use two metrics to evaluate fairness and throughput of CPU scheduling. First, the *fairness* metric represents the normalized performance compared to the performance of ideal scheduling, as used in Section 2.4. The second metric is *weighted throughput* which represents the overall system throughput. The metric is the weighted average of *fairness* of threads, where the weights of threads are used as the weight for averaging.

$$weighted\_throughput = \frac{\sum fairness \times w_{thread}}{\sum w_{thread}}$$

Single thread benchmark: Figure 11 shows the results of *namd* on the 16-core system. The same setup was used in Section 2.4. In the figures, the x-axis represents each experimental run repeated 5 times for both schedulers, and the y-axis represents the *fairness* metric. For each column, the triangles indicate the minimum and the maximum value of *fairness* and the lower and upper boxes show the 25%-ile

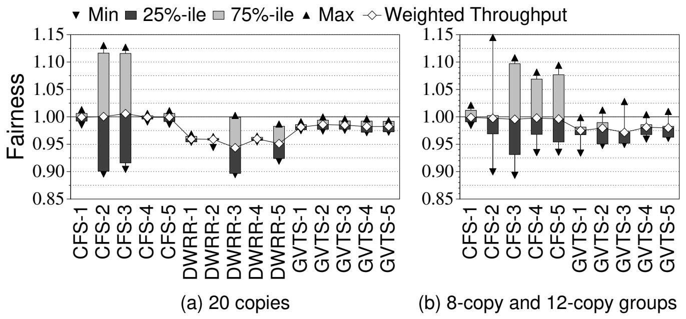

Fig. 13: Results of *milc* on 16-core system

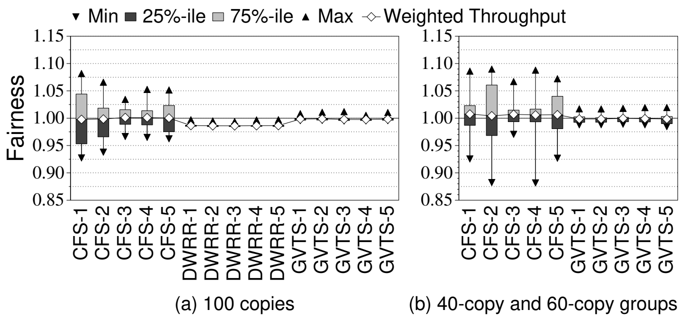

Fig. 14: Results of *milc* on 80-core system

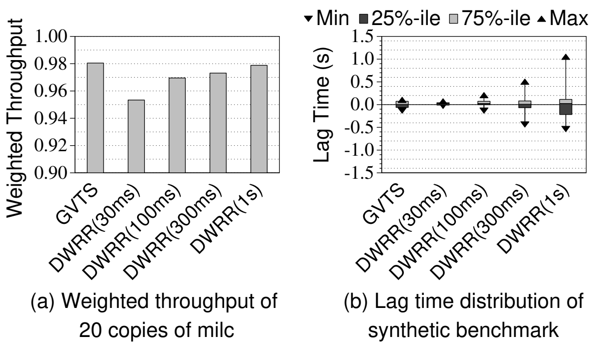

Fig. 15: DWRR results with varying round slice

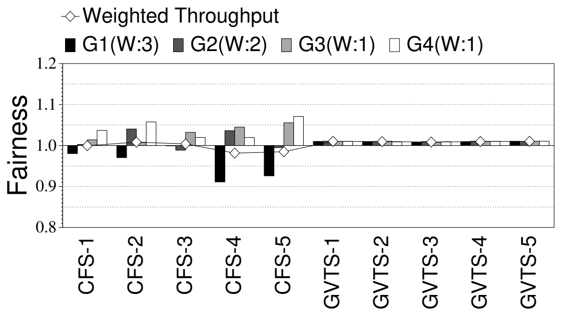

Fig. 16: 4 copies of *swaptions* with 40 threads. Weight=3:2:1:1

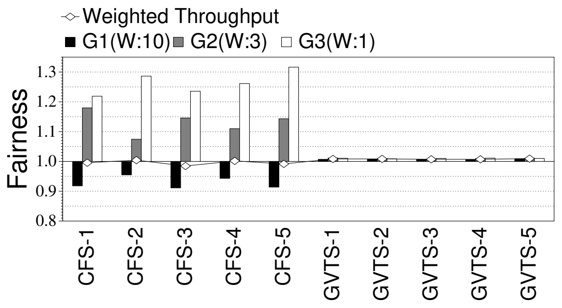

Fig. 17: 3 copies of *swaptions* with 80 threads. Weight=10:3:1

and 75%-ile value of *fairness*. The diamonds represent the weighted throughput for each experiment.

The results show that GVTS successfully maintains proportional fairness even when CFS fails to balance loads and fails to provide fairness, as expected in the synthetic benchmark results. Moreover, the difference in *weighted throughput* between two schedulers is less than 0.5%. The results show that GVTS can almost eliminate fairness violation without any severe throughput degradation. The results with DWRR are also similar to GVTS as it distributes CPU resource evenly to threads at round interval granularity. Note that *namd* is a CPU intensive workload and CPU allocation alone has a major impact on performance.

Figure 12 shows the results on the 80-core system. The scenarios are similar to the ones on the 16-core system, but the number of *namd* copies is increased 5 times. The trends are similar to the results on 16-core. This shows that our scheduler has enough scalability to work on large systems.

Figure 13 and 14 show the results with *milc*. Since *milc* is a memory intensive benchmark, the frequent migrations may lead to notable performance degradation. As similar to *namd* results, CFS shows a high performance variance between threads. In addition, the intensity of performance variance also differs between runs.

DWRR reduces the performance variance significantly, but it also reduces the overall throughput due to the thread migration overhead. The throughput degradation is 4∼6% on 16-core system and 1.4% on 80-core system. There is a notable difference in throughput degradation across machines, since CPUs of two machines are from different CPU vendors and the thread migration overhead is significantly affected by the CPU architectures.

Finally, GVTS shows the minimum performance variance with acceptable throughput degradation, due to optimization techniques to reduce thread migrations. The throughput degradation is around 2∼3% for 16-core system and less than 0.3% for 80-core system. However, although GVTS shows a less performance variance compared to CFS, there is a still minor performance variance, especially for Figure 13 (b) case. The main reason is that the number of thread migration events for fair scheduling can be biased between threads. This problem results in imbalanced migration overheads of threads and causes uneven performance of threads. Balancing the number of migrations between threads will be our future work.

To further analyze the trade-off of DWRR between fairness accuracy and migration overhead, we run the synthetic benchmark and *milc* with varying *round_interval* for DWRR which ranges from 30ms (default) to 1s. Figure 15 (b) shows the weighted throughput of 20 *milc* copies on 16-core system. We repeat the experiment 5 times for each parameter, and the average is shown in the figure. Note that the weighted throughput with CFS scheduler is close to 1 as shown in Figure 13 (a). The default *round_interval* (30ms) shows 5% additional throughput degradation compared to CFS. The overhead decreases as *round_interval* increases, and 1 second *round_interval* shows weighted throughput comparable to GVTS. However, 1 second *round_interval* degrades fairness accuracy of the scheduler. Figure 15 (a) shows the

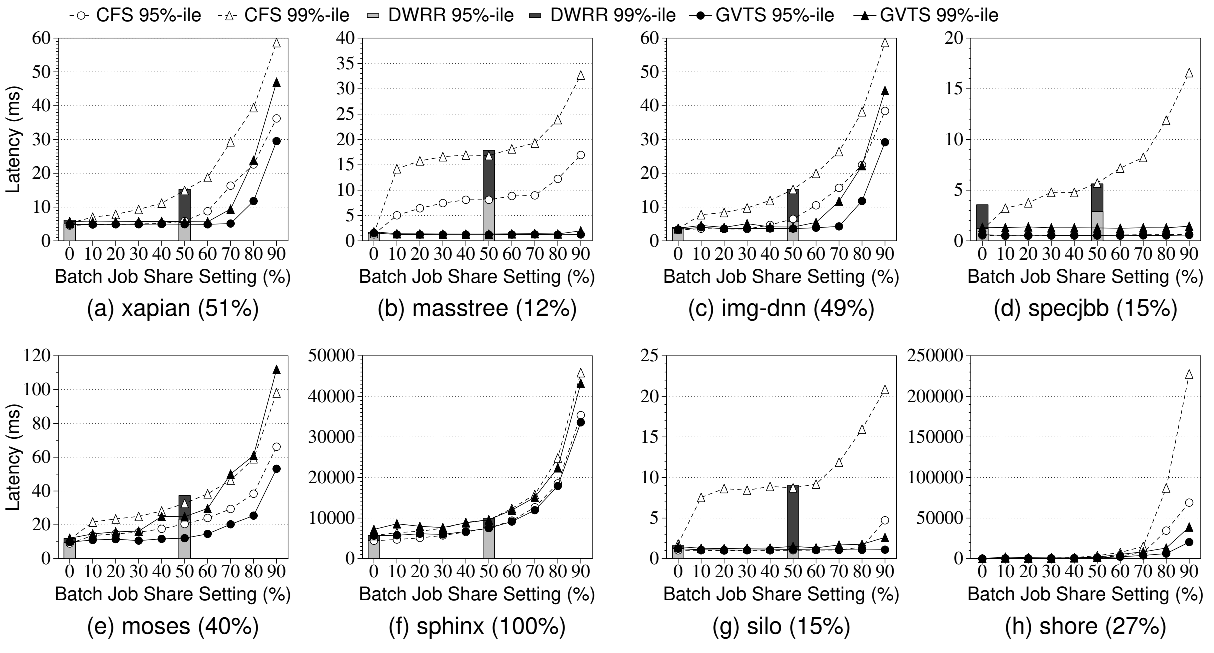

Fig. 18: Tail-latency of TailBench on 16 cores

lag time distributions of our synthetic benchmark, when it runs 20 threads on the 16-core system. DWRR with 1 second *round_interval* incurs lag time longer than 1 second, while GVTS shows a much shorter lag time. Consequently, GVTS achieves both accurate fairness and low migration overhead due to the topology-aware virtual time balancing and other optimizations.

Multi-threaded benchmark: To evaluate inter-group proportional fairness with a multi-threaded benchmark, we use 4 scenarios with *swaptions*. For all scenarios, each instance of *swaptions* belongs to different thread groups. Then, we adjust the weights of groups respectively for each scenario. The fairness between thread groups is measured in terms of the performance of each instance. Although we experiment 4 scenarios on both experimental platforms, we show 2 scenarios on 80-core system due to the space limit. The rest of the results show similar trends.

Figure 16 shows the results of the first scenario. We run 4 instances of *swaptions* with 40 threads per instance. Each instance belongs to different groups and the weight ratio between the groups is 3:2:1:1. In the graph, each column presents a run repeated 5 times for both of the schedulers, and the bars in each column show the *fairness* value of the instances of *swaptions*. Since *swaptions* is a CPU intensive workload and highly scalable, the performance of the copies must be proportional to its CPU share. As shown in the figure, CFS shows large performance variances up to 8%, while GMFS successfully provides inter-group proportional fairness, close to the ideally fair scheduler.

Figure 17 shows the results of the second scenario. The scenario runs 3 instances of *swaptions* with 80 threads per instance. As before, threads of each instance constitute a thread group, and the weight ratio between the three groups is 10:3:1. With CFS, the heavy weighted group tends to receive less CPU share and exhibit low *fairness*, and the light weighted groups tend to receive more CPU share than the ideal scheduling. On the other hand, GVTS does not show such problems and provides near-perfect fairness.

Note that all *fairness* with GVTS in Figure 16 and 17 slightly exceed 1. This is the effect of the fine-grained intra-group fair scheduling. Since *swaptions* has a barrier at the end of the procedure, it can finish when all of its threads finish. Fine-grained intra-group fair scheduling can fully utilize as much core as possible, and help all threads finish simultaneously. Without such scheduling, some threads lag behind other threads, and the program should wait until all lagged threads to finish.

5.4 Results of Server Workloads

As shown in Section 2.4, tail latency critical workloads, such as web servers, may suffer from the temporal fairness violations, degrading their quality-of-service due to tail latency increases. Since server workloads may be consolidated with batch jobs in clouds to improve resource utilization, fine-grained fair scheduling is important.

To evaluate server workloads, we use two benchmarks, *TailBench* [10] and *memcached*, and use *namd* for consolidated batch jobs. First, *TailBench* consists of 8 real-world server applications from various domains, including search engine, key-value store, translation, speech/image recognition, java middleware and OLTP. We use *integrated configuration* which a single process combines clients and application threads. It does not include network latency and network stack overheads, but focuses on the CPU usage of the server applications. Second, *memcached* represents the realistic usage scenario including the network overhead. We use *treadmill* [12] for the client driver on a separate machine. The client machine has two Intel Xeon E5-2630 CPUs and totally 40 cores. Third, for batch jobs, *namd*, we set the number of batch jobs to be equal to the number of CPUs in the system, so the batch jobs can fully utilize the whole system, if they have enough share. All scenarios are repeated 5 times and we use the average value of tail latencies to compensate for the performance variances across runs [12].

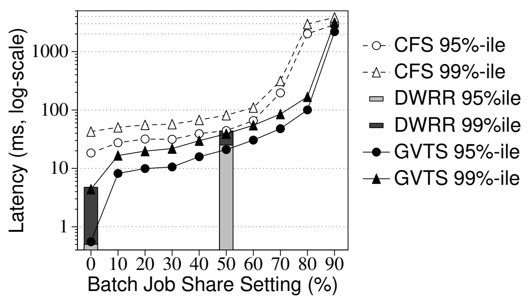

Fig. 19: Memcached Results on 16 cores

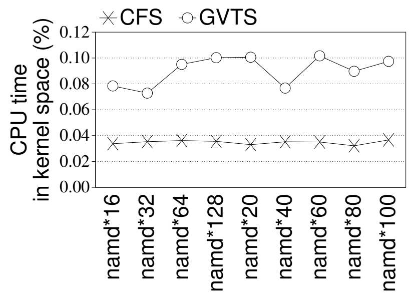

Fig. 20: Scheduling overhead: CPU time in kernel space

Figure 18 shows the results of *TailBench* [10] with batch jobs. The x-axis represents the desired CPU share for batch jobs. We adjust the relative weight of batch jobs and server applications to set the desired CPU share. When the desired CPU share for batch jobs is 10%, the server applications can use up to 90% of CPU share with the ideal scheduling. The y-axis represents the tail latency of server applications. The circles show the 95%-ile latencies and the triangles show the 99%-ile latencies. In addition, the empty marks with dotted lines represent the results of CFS and the filled marks with solid lines represent those with GVTS. For DWRR, which does not support thread grouping, we only show the results of two cases: one is when there are no batch jobs, and the other is when the weight of the batch jobs are equal to the weight of server workloads. The percentage after the application name shows the CPU utilization of the application without batch jobs. This represents the maximum CPU utilization for each application.

Similar to Figure 5 in Section 2.4, CFS has a harmful effect on tail latency in most of the applications. For example, *silo*'s maximum CPU utilization is just 15%, but its 99%-ile latency is drastically increased, when it can use 90% of CPU share. On the other hand, the fine-grained fair scheduling with GVTS removes such effect, and shows the stable tail latency until the batch job share setting is less than the maximum CPU utilization of server applications. On average, when the desired batch job's CPU share is 70%, GVTS reduces 95%-ile latency by 2.0X and 99%-ile latency by 3.0X compared to CFS.

Figure 19 shows the results of *memcached* with batch jobs. The notations are identical with the previous figure, but the y-axis is plotted on a logarithmic scale. Without batch jobs, the average CPU utilization of *memcached* is about 78%. When the desired batch job's CPU share is 10%$\sim$40%, which shows a reasonable tail latency with batch jobs, GVTS reduces 95%-ile latency by 2.1X$\sim$4.1X and 99%-ile latency by 2.0X$\sim$3.7X compared to CFS.

For both of the server workloads, DWRR shows the longer tail latency compared to GVTS. In addition, as DWRR does not support the thread grouping, the fine-grained adjustment of weights of groups is not supported.

5.5 Scheduling Overhead

In this section, we analyze the scheduling overhead of GVTS. There are two types of overheads from schedulers. First, the scheduling decision may negatively impact application performance by thread migrations. For example, if a scheduler frequently incurs thread migrations, the threads cannot efficiently exploit caches and thus their performance can be degraded. The previous experimental results show that this type of the overheads are negligible, as the weighted throughput of GVTS is similar to CFS, Even when all threads in a system are running memory intensive workloads, such as *milc* in our experiments, the overhead is less than 0.3%$\sim$3% depending on the CPU architecture.

Second, the scheduler itself consumes CPU share to run its algorithms. This directly degrades the application performance, since the applications cannot run while the scheduler is running. However, it is very difficult to measure the CPU time consumed by scheduler, since schedulers run for a very short time. Instead, we compare the statistics of CPU time in the kernel space between CFS and GVTS. Since we use the same version of kernel for CFS and GVTS, the comparison shows the additional CPU consumption of GVTS algorithm compared to CFS. Figure 20 shows the results. The numbers in the x-axis show the number of copies for the column, and the y-axis shows the percentage of CPU utilization in kernel space. As shown in the figure, GVTS consumes negligible 0.04$\sim$0.07% more CPU share. In addition, even though the number of threads increases, the CPU share in kernel space does not increases. This shows the scalability of GVTS in terms of the number of threads.

6 Related Work

6.1 Fair CPU Scheduling

There are several ways to support proportional fairness in CPU scheduling. First, as explained in Section 2.1, virtual time based schedulers define the virtual time as the weighted time given to each thread, and always select the thread with the lowest virtual time to run. Surplus Fair Scheduling [5], Borrowed-Virtual-Time [3], and Start-time Fair Queueing [4] are based on virtual time fair scheduling, and Completely Fair Scheduler (CFS) [1] in Linux kernel adopted this method to support proportional fairness since version 2.6.23. As this method restricts the difference in virtual time within a small constant, the maximum lag time is also bounded by a constant. However, as it needs to sort threads in virtual time order, the algorithmic complexity with N threads is $O(logN) \sim O(NlogN)$ depending on the implementation.

Second, Weighted Round-Robin (WRR) [14] based schedulers select a thread from a runqueue in a round-robin manner, and runs the selected thread for $w_i \times interval$. After a round, all threads receive CPU time exactly proportional to their weight. As the thread selection is done in round-robin, its algorithmic complexity is $O(1)$. However, the

	CFS [1]	DWRR [8]	Xen Credit [13]	GVTS(proposed)
Accurate fairness on multi-core	NO	Trade-off with migration overhead	YES	YES
Scalability	YES	YES	limited	YES
Thread group support	Multi-level	NO	1-level only	Multi-level

TABLE 1: Comparison of GVTS with prior approaches

maximum lag time is $w_i \times$ *interval* for each round, which is usually larger than virtual time fair scheduling. Virtual-Time Round-Robin scheduler (VTRR) [15] reduces the maximum lag time to *interval* by reducing the execution time for each selection to *interval* but increasing the number of selections for threads with high weights. Group Ratio Round-Robin scheduler [16] extends VTRR for multiprocessor systems, and Grouped Distributed Queues scheduler [17] and Distributed Weighted Round-Robin scheduler [8] proposed scalable algorithms for large scale system which are based on weighted round-robin.

Last, Lottery scheduling [18], [19] supports proportional fairness probabilistically. Selecting a thread is randomized but with a probability of selection being proportional to its *Lottery tickets*, which are encapsulated resource rights. The algorithmic time complexity with N threads is $O(logN)$, and the lag time is not bounded in a constant time.

Another aspect of fair CPU scheduling is to support fairness when cores have different computing capabilities [20], [21], [22], [23]. In such asymmetric multicore processors (AMP), an equal CPU share does not provide an equal performance. Kwon et al. defined the fairness on AMP as the state that *all threads receive the same CPU share for each type of CPUs*, and proposed a scheduler for virtual machines on AMP [20]. Craeynest et al. studied the fairness support for AMPs with HW supports [21]. Kim et al. proposed a SW scheduler supporting the proportional fairness for AMP [22], [23].

6.2 Scheduler Implementations

In this section, we describe the design choices of three open source scheduler implementations and compare them to the proposed scheduler.

CFS: Completely Fair Scheduler (CFS) [1] is the most widely used open source scheduler. As explained in Section 2, CFS use the virtual time fair scheduling for per-CPU scheduling, but use load balancing mechanism for global scheduling. CFS provides nested thread group support and its load balancing mechanism is scalable to the number of CPUs. However, there are many corner cases that load balancing fails to provide accurate fairness on multi-core systems. Lozi et. al. fixed some corner case problems of load balancing [24] and Huh et. al proposed a mechanism which periodically places threads based on the current virtual time [25]. Compared to them, our work fundamentally solves the problems of load balancing by extending virtual time fair scheduling on multi-core systems.

DWRR: Distributed Weighted Round-Robin (DWRR) [8] scheduler implements a round-robin based fair scheduling for multi-core systems. Although DWRR uses the Linux kernel implementation, O(1) scheduler or CFS, for per-CPU scheduling, it uses a weighted round-robin for global scheduling. DWRR maintains *round* for each thread, and *round* is incremented when the thread receives CPU time as $w_i \times$ *round_interval*. Although its round balancing mechanism is somewhat similar to the balancing in GVTS, it works as a whole system is flat, and there is a single *round_interval* value. Depending on the length of *interval*, the thread migration overhead or the lag time bound becomes large. Note that large lag time indicates low fairness accuracy. Moreover, DWRR does not support thread grouping, which is required for virtualization or containerization.

Xen Credit: Xen Credit Scheduler is the current mainline scheduler of Xen hypervisor [13]. It is designed to schedule virtual CPUs of virtual machines, so it supports only flat thread groups for inter-group fairness. In Xen Credit Scheduler, each virtual CPU has a *credit* value, which indicates the right of CPU time usage, and the scheduler periodically distributes credits for all the virtual CPUs according to their weights. With per-CPU runqueues in the scheduler. a physical CPU selects any virtual CPU which has a remaining credit and runs it until all of its *credit* is consumed. If all virtual CPUs in the runqueue are running out of credits, the physical CPU scans other physical CPUs first to find virtual CPUs with remaining credits. In this way, the scheduler provides proportional fairness between virtual CPUs. However, as the credit distribution requires a whole scheduler lock, the scalability is limited. The recently released Xen Credit Scheduler 2 removes the scalability bottleneck. However, its global scheduler mechanism is similar to the load balancing mechanism of CFS.

Comparison: Table 1 compares the aforementioned scheduler implementations with the proposed one. First, CFS has a scalable algorithm but cannot guarantee exact proportional fairness on multi-core systems. Second, DWRR provides accurate fairness on multi-core systems, but the accuracy is traded off with the thread migration overhead. In addition, it does not support any thread grouping. Third, Xen Credit Scheduler also provides accurate fairness on multi-core systems, but it has a scalability bottleneck on distributing *credit* to all virtual CPUs in the system. Finally, the proposed GVTS scheduler provides accurate fairness on multi-core systems with little thread migration overhead. It also supports nested thread grouping.

7 Conclusion

This paper investigated the impact of the state-of-the-art proportional fair scheduling on multi-core systems. The local virtual time fair scheduling in Linux can negatively affect the performance consistency on batch jobs and significantly increase tail latency of server applications. To provide strict proportional fairness on multi-core systems, this paper proposed Global Virtual Time Fair Scheduling (GVTS) which extends the virtual time fair scheduling on multi-core systems with negligible thread migration overhead. The proposed scheduler is implemented on the Linux kernel and its source code is available online. The experimental results show that it removes performance variance of batch workloads and significantly reduces tail latencies of server workloads. Our source code is publicly available at `https://github.com/cdkimcode/gvts`.

ACKNOWLEDGMENTS

This work is supported by the National Research Foundation of Korea (NRF-2016R1A2B4013352) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea.

REFERENCES

[1] I. Molnar, "CFS scheduler," http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt, last accessed: 2018-05-19.
[2] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, Jun. 1993.
[3] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proc. Symposium on Operating Systems Principles (SOSP)*, 1999, pp. 261–276.
[4] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical cpu scheduler for multimedia operating systems," in *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1996, pp. 107–122.
[5] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors," in *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2000, pp. 4:1–4:14.
[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proc. Symposium on Theory of Computing (STOC)*, 1993, pp. 345–354.
[7] A. Chandra and P. Shenoy, "Hierarchical scheduling for symmetric multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 418–431, March 2008.
[8] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," in *Proc. symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009, pp. 65–74.
[9] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, September 2006.
[10] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.
[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.
[12] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2016, pp. 456–468.
[13] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, Sep. 2007.
[14] J. Nagle, "On packet switches with infinite storage," *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 435–438, April 1987.
[15] J. Nieh, C. Vaill, and H. Zhong, "Virtual-time round-robin: An o(1) proportional share scheduler," in *Proc. USENIX Annual Technical Conference (ATC)*, 2001, pp. 245–259.
[16] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, "Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems," in *Proc. USENIX Annual Technical Conference (ATC)*, 2005, pp. 337–352.
[17] B. Caprita, J. Nieh, and C. Stein, "Grouped distributed queues: Distributed queue, proportional share multiprocessor scheduling," in *Proc. Symposium on Principles of Distributed Computing (PODC)*, 2006, pp. 72–81.
[18] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1994, pp. 1:1–1:11.
[19] D. Petrou, J. W. Milford, and G. A. Gibson, "Implementing lottery scheduling: Matching the specializations in traditional schedulers," in *Proc. USENIX Annual Technical Conference (ATC)*, 1999, pp. 1:1–1:14.
[20] Y. Kwon, C. Kim, S. Maeng, and J. Huh, "Virtualizing performance asymmetric multi-core systems," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2011, pp. 45–56.
[21] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proc. the international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 177–187.
[22] C. Kim and J. Huh, "Fairness-oriented os scheduling support for multicore systems," in *Proc. the International Conference on Supercomputing (ICS)*, 2016, pp. 29:1–29:12.
[23] ——, "Exploring the design space of fair scheduling supports for asymmetric multicore systems," *IEEE Transactions on Computers (in press)*, pp. 1–16, DOI: 10.1109/TC.2018.2796077.
[24] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: A decade of wasted cores," in *Proc. the European Conference on Computer Systems (EuroSys)*, 2016, pp. 1:1–1:16.
[25] S. Huh, J. Yoo, M. Kim, and S. Hong, "Providing fair share scheduling on multicore cloud servers via virtual runtime-based task migration algorithm," in *Proc. IEEE International Conference on Distributed Computing Systems (DCS)*, 2012, pp. 606–614.

Changdae Kim is a research fellow in Computer Science at Korea Advanced Institute of Science and Technology (KAIST). His research interests are in computer architecture, operating systems, and cloud computing. He received his BS, MS, and PhD in computer science from KAIST.

Seungbeom Choi has received the BS degree in computer engineering from Sungkyunkwan University (SKKU) and MS degree in computer science from KAIST. His research focuses on cloud computing, parallel computing, deep learning, and GPU acceleration.

Jaehyuk Huh is an associate professor of Computer Science at KAIST. His research interests are in computer architecture, parallel computing, virtualization and system security. He received a BS in computer science from Seoul National University, and an MS and a PhD in computer science from the University of Texas at Austin.