Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm

Sungju Huh[1)], Jonghun Yoo[2)], Myungsun Kim[2)] and Seongsoo Hong[1), 2)]
[1)] Department of Intelligent Convergence Systems,
Graduate School of Convergence Science and Technology
[2)] School of Electrical and Computer Engineering
Seoul National University, Republic of Korea
{sjhuh, jhyoo, mskim, sshong}@redwood.snu.ac.kr

Abstract— **While Linux is the most favored operating system for an open source-based cloud data center, it falls short of expectations when it comes to fair share multicore scheduling. The primary task scheduler of the mainline Linux kernel, CFS, cannot provide a desired level of fairness in a multicore system. CFS uses a weight-based load balancing mechanism to evenly distribute task weights among all cores. Contrary to expectations, this mechanism cannot guarantee fair share scheduling since balancing loads among cores has nothing to do with bounding differences in the virtual runtimes of tasks. To make matters worse, CFS allows a persistent load imbalance among cores. This paper presents a virtual runtime-based task migration algorithm which directly bounds the maximum virtual runtime difference among tasks. For a given pair of cores, our algorithm periodically partitions runnable tasks into two groups depending on their virtual runtimes and assigns each group to a dedicated core. In doing so, it bounds the load difference between two cores by the largest weight in the task set and makes the core with larger virtual runtimes receive a larger load and thus run more slowly. It bounds the virtual runtime difference of any pair of tasks running on these cores by a constant. We have implemented the algorithm into the Linux kernel 2.6.38.8. Experimental results show that the maximal virtual runtime difference is 50.53 time units while incurring only 0.14% more run-time overhead than CFS.**

Keywords- multi-core scheduling; fair share scheduling; load balancing; cloud server computing

I. INTRODUCTION

Cloud computing has emerged as a new computing paradigm and has been rapidly adopted in diverse business domains. This naturally incurs difficult challenges to cloud service providers in terms of the increased complexity and scale of cloud servers. For instance, the type of services that cloud computing provides becomes diversified beyond ordinary service features such as web-hosting, data processing and utility computing. Also, cloud data centers are requested to service a large number of clients and thus are constructed at a huge scale. Among such challenges, the most imminent is fair service provisioning at the presence of extreme and diverse workloads.

In principle, cloud computing allows tenants to pay for computing resources which are dynamically provisioned by a cloud service provider. The cloud service provider puts its abundant computing resources in a consolidated cloud server and lets tenants rent a slice of those resources. It is thus essential to guarantee the provisioning of required computing resources as specified in service level agreements signed by a tenant and the cloud service provider. Among various types of resources, CPU deserves special attention since the multicore nature of cloud servers complicates the fair share scheduling of tasks of various tenants.

Linux is the most favored operating system for an open source-based cloud infrastructure due to its numerous advantages. Linux comes with a number of tool chains and software packages for cloud computing which are easily customized to fit the special needs of diverse tenants [1]. Examples of a Linux-based cloud infrastructure include Eucalyptus [2] and OpenNEbula [3], to name a few. Cloud service providers can thus effectively construct cloud data centers for their own purposes, exploiting any of those systems.

Despite such advantages, Linux falls short of expectations when it comes to fair share multicore scheduling. This is because the primary task scheduler of the mainline Linux kernel, called the completely fair scheduler (CFS), cannot provide a desired level of fairness in a multicore system. The goal of CFS is to provide fair share scheduling by giving each task CPU time proportional to its weight. In a single core system, it successfully achieves this goal by using the notion of a task's virtual runtime, defined as the task's cumulative runtime inversely scaled by its weight. At every scheduling decision point, it dispatches the task with the smallest virtual runtime so as to balance the virtual runtimes of runnable tasks.

On the other hand, CFS uses a weight-based load balancing mechanism to achieve fair share scheduling in a multicore system. Specifically, it maintains the sum of weights for each core and tries to balance the sums among all cores in the system. Unfortunately, this mechanism cannot

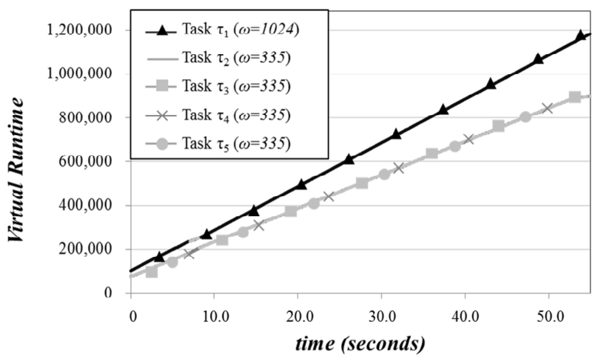

Figure 1. Divergence of virtual runtime difference in CFS.

guarantee fair share scheduling in a multicore system. This is because balancing loads among cores has nothing to do with bounding differences in the virtual runtimes of tasks. Note that virtual runtime distribution varies on cores regardless of loads and CFS is a per-core scheduling algorithm which runs independently of each other. To make matters worse, CFS cannot provide perfect load balancing for two reasons: weight quantization error and reluctant load balancing. The quantization error is resulted from the fact that a task's weight is specified as a predefined positive integer in Linux implementations. In addition to this, CFS allows for a persistent load imbalance among cores since it is very reluctant in performing load balancing unless load imbalance exceeds a given threshold.

Fig. 1 shows a motivating example where CFS fails to achieve fairness in a multicore system. In this example, five CPU-intensive tasks are running on two cores. Their weights are specified on the top left hand side of the figure. In principle, CFS allocates a task on the core with the smallest sum of weights when the task is being forked. In our example, τ_1 is created first and assigned on either one of the cores and the remaining tasks are on the other. CFS initiates load balancing when the amount of load imbalance gets larger than a threshold derived depending on a given workload. Since load imbalance in our example is not large enough, all the tasks remain running on the same cores with their weight sums being 1024 and 1340, respectively. As shown in Fig. 1, the virtual runtime difference between the two cores diverges as time goes on.

In order to overcome the shortcomings of the current CFS, we propose a virtual runtime-based task migration algorithm. This algorithm focuses on providing fairness within a pair of cores. For a given pair of cores, our algorithm sorts runnable tasks in the decreasing order of their virtual runtimes. It then partitions them into two groups such that the load difference between the two is bounded by the largest weight in the task set. Each group is then assigned to a dedicated core. Since a core with larger virtual runtimes receives a larger load, its virtual runtime advances at a slower pace. Since retarded tasks on the lightly-loaded core catch up with tasks on the heavily-loaded core, the algorithm needs to be run periodically.

Perfect fair share scheduling such as GPS (generalized processor sharing) is achieved when the virtual runtime difference among all runnable tasks is made zero at all times [4]. Since it is impossible to implement GPS in reality, we attempt to bound the virtual runtime difference of any pair of tasks by a small number. The difference converges to the balancing period multiplied by the upper bound on the pace difference of the tasks. Note that the pace difference of any pair of tasks is bounded by some constant because the load difference is also bounded by the largest task weight in our algorithm. By doing so, our algorithm provides pairwise fairness between two cores. For a general multicore system with more than two cores, an additional virtual runtime balancing policy is needed such that it repeatedly applies the proposed algorithm in a hierarchical fashion to achieve global fairness. Such a policy is beyond the scope of this paper and we leave it as our future work.

A tradeoff exists between fairness and run-time overhead in our algorithm. The balancing period is a tunable parameter that can control such a tradeoff. A smaller period leads to a smaller bound in virtual runtime difference while incurring a higher overhead due to frequent task migration and vice versa. The balancing period is selected by a system administrator who has to strike a balance between the desired level of fairness and the system performance.

We have implemented the proposed algorithm into the Linux kernel 2.6.38.8 and measured the maximal virtual runtime difference of tasks as well as the run-time overhead of the algorithm. Experimental results prove the effectiveness of our algorithm. The maximal virtual runtime difference with 50.53 time units was observed when the balancing period was set to 100 milliseconds. It is equal to 50.53 milliseconds when converted into the physical runtime of a task with nice value 0. The extra performance overhead of our algorithm was only 0.14% of the legacy CFS. This implies that our algorithm incurs only a negligible run-time overhead.

The rest of this paper is organized as follows. Section 2 discusses the existing fair share scheduling algorithms for multicore systems. Section 3 formally analyzes the Linux CFS and its load balancing mechanism. In Section 4, we model the target system and formulate the problem. Section 5 describes our virtual runtime-based task migration algorithm in detail. Section 6 reports on the experimental evaluation and Section 7 provides our conclusion.

II. Related Work

There have been a number of fair share scheduling algorithms for multicore systems in the literature. They can be divided into two categories according to their run-queue structure: one based on a centralized run-queue [5, 6, 7, 8, 9, 10] and the other based on distributed run-queues [11, 12, 13, 14]. The former generally outperforms the latter in a system with a small number of cores because they make scheduling decisions globally by exploiting information on available resources. However, they suffer from the lack of scalability as the number of cores increases since lock contentions on the centralized run-queue become a performance bottleneck.

Algorithms based on distributed run-queues mostly avoid such a scalability problem since only a limited amount of data is shared among multiple cores. Thus, they can easily scale up with the number of cores. Instead, they require some kind of progress-balancing algorithm to provide global fairness among distributed run-queues. Most algorithms found in the literature make use of a weight-based load balancing mechanism which aims just to balance the sum of weights on each core [11, 12, 13]. Caprita et al. propose Grouped Distributed Queues (GDQ) [11] which is a fair share scheduling algorithm for multiprocessor systems. It schedules tasks within a core using a round-robin algorithm. The algorithm periodically groups tasks based on their weights and reallocates the groups to cores so that the aggregated group weight of each core becomes balanced. The Linux CFS [12] also attempts to provide fair CPU allocation in multicore systems. It schedules tasks by using the notion of virtual runtime and redistributes tasks to cores to balance the weight sum of each core. As mentioned above, the weight-based algorithm fails to achieve fairness in practice even if the exactly same amount of load is given to each core.

We identify a class of load balancing algorithms and call them progress-based algorithms in order to address the limitation of the weight-based ones. Particularly, we propose a virtual runtime-based task migration algorithm which directly aims to balance tasks' virtual runtimes. To the best of our knowledge, there has existed only one algorithm in the literature which can be classified as a progress-based one. The Distributed Weighted Round-Robin (DWRR) algorithm is proposed by Li et al. as a scalable multiprocessor fair share scheduling algorithm [14]. It schedules tasks via weighted round-robin on each core. Across cores, it performs load balancing to ensure that all tasks go through the same number of rounds. This number represents the progress of each core. Unfortunately, DWRR may suffer from poor interactivity due to the existence of two run-queues originated from the Linux *O(1)* scheduler [13]. DWRR maintains an extra run-queue for balancing the round number in addition to a conventional run-queue on each core. Note that in the *O(1)* scheduler, tasks on the expired run-queue could suffer from severe starvation by I/O bound tasks [15]. DWRR inherits a similar property.

In this paper, we propose an instance of the progress-based multicore fair share scheduling algorithm in that virtual runtime represents the progress of each task. Our algorithm makes use of a single run-queue on each core and thus does not suffer from the interactivity problem. As we have extended CFS to incorporate the virtual runtime-based task migration mechanism, we formally analyze the behavior of CFS and its original load balancing mechanism in the following section.

III. Understanding Linux CFS

This section gives an overview of CFS, including its data structure, the per-core fair share scheduling algorithm and the weight-based load balancing mechanism for a multicore system.

A. Data structure

CFS is a symmetric multiprocessor scheduling algorithm with a distributed run-queue structure. The Linux kernel maintains a dedicated run-queue for each core and lets a CFS instance of a core make scheduling decisions independently of each other. A run-queue for a core keeps a list of its runnable tasks. These tasks are sorted according to the non-decreasing order of their virtual runtimes. In order to reduce the run-time complexity of the run-queue manipulation, the Linux kernel implements a run-queue with a self-balanced binary tree called a red-black tree [16]. CFS can select the task with the smallest virtual runtime with *O(1)* complexity since it is found in the leftmost leaf of the tree.

B. Per-core fair share scheduling algorithm

An important task attribute in CFS is the weight of a task since CFS attempts to give each task CPU time proportional to its weight. In order to allow a system administrator to specify weights for tasks in a way consistent with conventional Linux kernels, CFS makes use of nice values. In conventional Linux, nice values are used to denote task priorities. In CFS, a nice value is mapped to a specific weight value. Nice values range over [-20, 19] where a smaller value corresponds to a larger weight.

In a single core system, CFS tries to schedule runnable tasks such that all tasks on a given core have virtual runtimes only with small differences. A task's virtual runtime is defined as the task's cumulative runtime inversely scaled by its weight. If virtual runtimes are the same among all the tasks at a given point in time, then the tasks are given the exactly fair amount of CPU time at that time.

Let ω_0 be the weight of nice value *0* and $W(\tau_i)$ be the weight of task τ_i. Suppose $PR(\tau_i,t)$ denotes the amount of the cumulative physical runtime of task τ_i at time *t*. In CFS, the virtual runtime of task τ_i at time *t* is defined as bellow.

$$VR(\tau_i,t) = \frac{\omega_0}{W(\tau_i)} \times PR(\tau_i,t) \tag{1}$$

The smaller a task's virtual runtime is, the more the task needs to be scheduled.

In order to enforce fair share scheduling at a reasonable run-time cost, CFS makes use of the notion of a time slice. A

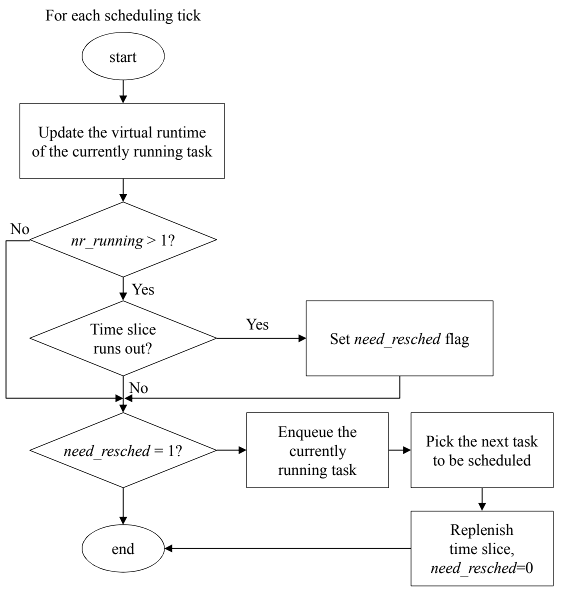
For each scheduling tick
start
Update the virtual runtime of the currently running task
nr_running > 1?
No
Yes
Time slice runs out?
Yes
Set need_resched flag
No
need_resched = 1?
Enqueue the currently running task
Pick the next task to be scheduled
Replenish time slice, need_resched=0
end

Figure 2. Flowchart of per-core fair share scheduling algorithm in CFS.

time slice is associated with a task and is defined as a time interval for which the task is allowed to run without being preempted. In CFS, the length of a task's time slice is proportional to its weight. The time slice of task τ_i is computed by

$$TS_{\tau_i} = \frac{W(\tau_i)}{\sum_{\tau_j \in S} W(\tau_j)} \times P \tag{2}$$

where S is the set of runnable tasks, $W(\tau_i)$ is the weight of τ_i and *P* is the constant for given workload. *P* is defined as

$$P = \begin{cases} sysctl\_sched\_latency & \text{if } n < nr\_latency, \\ min\_granularity \times n & \text{otherwise} \end{cases} \tag{3}$$

where *n* is the number of tasks. *sysctl_sched_latency*, *nr_latency* and *min_granularity* are system-wide constants and their values are 6, 8 and 0.75, respectively, in the current Linux implementations.

Fig. 2 summarizes the run-time algorithm of CFS in a flowchart. At each scheduling tick, CFS updates the virtual runtime of currently running task τ_i using (1). If there is more than one task in its run-queue, CFS checks whether τ_i runs out of its time slice. If so, *need_resched* flag is set. In turn, if the flag is set, it schedules the task with the smallest virtual runtime from the run-queue; it then replenishes the time slice of τ_i and put it back to the run-queue.

C. *Weight-based load balancing mechanism*

In a multicore system, CFS additionally performs weight-based load balancing to evenly distribute the system load among run-queues in the system. The load of run-queue Q_k is defined by

$$L_k = \sum_{\tau_i \in S_k} W(\tau_i) \tag{4}$$

where S_k is the set of tasks in Q_k. In order to keep track of L_k, CFS maintains an integer value, *load* in its algorithm.

CFS performs load balancing at a specific interval measured in jiffies. This interval, denoted by T, is specified offline in the system configuration file. CFS maintains last load balancing time $t_{last,k}$ for run-queue Q_k. For every scheduling tick, it checks if Q_k should be rebalanced by comparing current time $t_{current}$ with $t_{last,k}$. If $t_{current}$ is greater than $T + t_{last,k}$, it triggers load balancing code to move tasks from busiest run-queue $Q_{busiest}$ to Q_k. $Q_{busiest}$ is defined as the run-queue with the largest load. CFS may move multiple tasks and the amount of load to be moved is defined as bellow.

$$L_{imbal} = min(min(L_{busiest}, L_{avg}), L_{avg} - L_k) \tag{5}$$

where $L_{busiest}$ is the load of $Q_{busiest}$ and L_{avg} is an average load in the system. To be conservative, CFS does not move any tasks if the following condition holds.

$$L_{imbal} < \min_{\tau_i \in s_{busiest}} (W(\tau_i)) / 2 \tag{6}$$

where $s_{busiest}$ is the set of tasks in $Q_{busiest}$.

CFS triggers extra load balancing whenever it finds a run-queue empty. In that case, it steals some tasks from the busiest run-queue and the amount of tasks to be moved is defined in (5).

IV. System Modeling and Problem Formulation

In this section, we model the target system and define our metric for evaluating the degree of fairness provided by the proposed algorithm.

A. *System modeling*

A cloud service provider is capable of accepting a large number of tenants to a data center. A tenant is an individual or an organization that buys computing resources from the service provider. A service contract is signed between a tenant and the service provider according to a service level agreement (SLA) [17, 18]. In particular, an SLA can formally specify expected and acceptable computing power in terms of measurable units such as a million instructions per second (MIPS).

Fig. 3 depicts the target system architecture. A data center comprises of hundreds or thousands of servers. Each server is equipped with an SMP processor and runs the Linux operating system. The cloud service provider assigns a number of tasks to each tenant according to the SLA. Each task is a regular Linux task with a specific weight and nice value. A server in a data center may run a wide mix of tasks owned by different tenants. CFS schedules these tasks according to their weights.

B. *Problem definition*

We first define a metric for evaluating the fairness of a given multicore scheduling algorithm, and then formulate our problem at hand. We then explain why CFS fails to achieve the desired level of fairness in a multicore system.

Let $S=\{\tau_1,\tau_2,...,\tau_n\}$ be the set of tasks running on a pair of cores $P=\{P_1,P_2\}$. Their weights are defined by $W: S \rightarrow \mathbb{N}$ where $\mathbb{N}$ is a set of natural numbers. Let $D_{i,j}(t)$ be difference in virtual runtimes of two tasks τ_i and τ_j in S at time *t*.

As a metric for evaluating the fairness, we use the largest $D_{i,j}(t)$ for any τ_i and τ_j in *S* and denote it by $D_{max}(t)$. It represents the largest gap between the progress of two tasks scaled by their weights. Clearly, $D_{max}(t)=0$ always holds in an ideal scheduling algorithm such as GPS. However, it is impractical to implement such an algorithm since an infinitesimally small time quantum must be given to each task. In practice, a scheduling algorithm is said to achieve strong fairness if $D_{max}(t)$ is bounded by a constant $C=\alpha T$ where α is a constant and *T* is a balancing period. The smaller C is, the fairer an algorithm is. On the other hand, we say that an algorithm fails to achieve fairness if $D_{max}(t)$ diverges. We aim to come up with a virtual runtime-based task migration algorithm for CFS such that $D_{max}(t)$ is bounded by αT.

CFS cannot bound $D_{max}(t)$ since it uses a weight-based load balancing mechanism as explained in Section 3. To make matters worse, $D_{max}(t)$ easily diverges in CFS because it allows for a persistent load imbalance and does not move tasks as long as Inequality (6) is met. In real world applications, load imbalance is not avoidable unless the exactly same amount of load is given to every core, which is a very exceptional case. As a result, CFS fails to allocate CPU times to tasks as specified with their weights.

Table I shows such an example task set with which CFS leads to diverging $D_{max}(t)$. It consists of five CPU-intensive tasks running on two cores P_1 and P_2. Initially, CFS assigns τ_1 to P_1 and the others to P_2 since such a configuration leads to the minimum load imbalance of 316. Unfortunately, CFS does not move any tasks since Inequality (6) simply holds in this initial configuration. As a result, the virtual runtime of τ_1 goes faster than those of other tasks and the gap between them diverges indefinitely.

TABLE I. EXAMPLE TASK SET SPECIFICATION THAT LEADS TO VIRTUAL RUNTIME DIVERGENCE IN CFS

	Nice value	*Weight*	*Initial distribution*
τ_1	0	1024	P_1
τ_2	5	335	P_2
τ_3	5	335	P_2
τ_4	5	335	P_2
τ_5	5	335	P_2

V. VIRTUAL RUNTIME-BASED TASK MIGRATION

In this section, we present the virtual runtime-based task migration algorithm and show how it achieves the desired properties.

Fig. 4 shows the overview of our algorithm. It is triggered at time $t=kT$ where k is a positive integer and T is a balancing period. The algorithm takes two task sets S_1 and S_2 as inputs which were executed on P_1 and P_2 in the previous period $[(k-1)T,kT]$, respectively. It then outputs S_1' and S_2'. They will be newly allocated to P_1 and P_2, respectively. The inputs to the algorithm are specified as below.

- (S_1,S_2) : initial partitions of the previous period
- $W: S_1 \bigcup S_2 \rightarrow \mathbb{N}$: weights of tasks

The pseudo code for the algorithm is shown in Fig. 5. Our algorithm is executed every T milliseconds. It sorts the tasks in the decreasing order of their virtual runtimes and stores them in list X in line 1. In lines 2 and 3, it consecutively performs the PARTITION and MIGRATE algorithms.

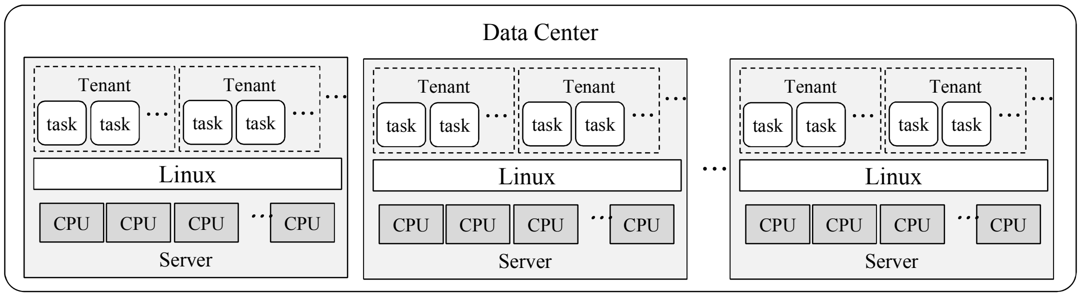

Figure 3. Target cloud computing system architecture.

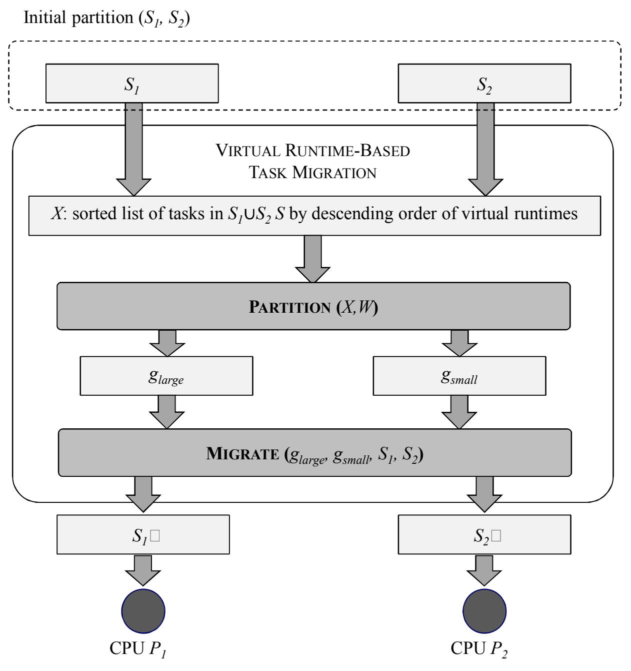
Initial partition (S_1, S_2)
S_1
S_2
VIRTUAL RUNTIME-BASED TASK MIGRATION
X: sorted list of tasks in $S_1 \cup S_2$ S by descending order of virtual runtimes
PARTITION (X, W)
g_{large}
g_{small}
MIGRATE (g_{large}, g_{small}, S_1, S_2)
S_1□
S_2□
CPU P_1
CPU P_2

Figure 4. Overview of the virtual runtime-based task migration algorithm.

We first discuss the PARTITION algorithm. It partitions tasks in X into two groups g_{large} and g_{small} so that the following three properties are met.

(1) A task in g_{large} has an equal or larger virtual runtime than any task in g_{small}.
(2) L_{large} is larger than L_{small} where L_{large} and L_{small} are the loads of g_{large} and g_{small}, respectively.
(3) $L_{large} - L_{small}$ is bounded by the largest weight in the system.

In each iteration of the **while** loop in lines 4~7, a task is removed from the head of X and inserted to g_{large}. The loop terminates as soon as L_{large} becomes larger than the half of the total load in the system. All the remaining tasks in X are then inserted to g_{small} in line 8.

The MIGRATE algorithm allocates g_{large} and g_{small} to P_1 and P_2 in such a way that it can minimize the number of task migrations. Line 1 computes the number of task migrations needed when g_{large} and g_{small} are allocated to P_1 and P_2, respectively. Similarly, line 2 computes the number of task migrations needed in the other way around. The algorithm then chooses a mapping which leads to a smaller number of task migrations in lines 3~9.

In order to prove that our algorithm bounds $D_{max}(t)$ by a constant for a given pair of cores, we first define the following notations. For given time interval $[kT,(k+1)T]$, the virtual runtime increment of task τ_i is denoted by $\Delta VR_i(T)$. The difference in $\Delta VR_i(T)$ between a pair of tasks (τ_i, τ_j) is defined by $\Delta VR_{i,j}(T)$. ω_{max} and ω_{min} denotes the largest and smallest weights in the system, respectively.

```
Algorithm: Virtual runtime-based task migration

VIRTUAL RUNTIME-BASED TASK MIGRATION
inputs: $S_1, S_2, W$
begin
1   List $X \leftarrow$ sort tasks in $S_1 \bigcup S_2$ in the descending order of their
        virtual runtimes
2   $(g_{large}, g_{small}) \leftarrow$ PARTITION$(X, W)$
3   MIGRATE $(g_{large}, g_{small}, S_1, S_2)$
end

PARTITION
inputs: $X$, $W$
begin
1   $g_{large} \leftarrow \emptyset, g_{small} \leftarrow \emptyset$
2   Integer $u \leftarrow 0$
3   Integer $L \leftarrow$ sum of weights of tasks in $X$
4   while $(u < 1/2 \times L)$
5       $t \leftarrow$ remove the head of $X$
6       $u \leftarrow u + W(t)$
7       $g_{large} \leftarrow g_{large} \bigcup \{t\}$
8   $g_{small} \leftarrow$ the set of remaining tasks in $X$
9   return $(g_{large}, g_{small})$
end

MIGRATE
inputs: $g_{large}, g_{small}, S_1, S_2$
begin
1   $\delta_{large-1} \leftarrow g_{large} - S_1, \delta_{small-2} \leftarrow g_{small} - S_2$
2   $\delta_{large-2} \leftarrow g_{large} - S_2, \delta_{small-1} \leftarrow g_{small} - S_1$
3   if $(|\delta_{large-1} + \delta_{small-2}| > |\delta_{large-2} + \delta_{small-1}|)$ then
4       migrate $\delta_{large-2}$ from core $P_1$ to $P_2$
5       migrate $\delta_{small-1}$ from core $P_2$ to $P_1$
6   else
7       migrate $\delta_{small-2}$ from core $P_1$ to $P_2$
8       migrate $\delta_{large-1}$ from core $P_2$ to $P_1$
9   endif
end
```

Figure 5. Pseudo code for the virtual runtime-based task migration algorithm.

Now we show that our algorithm bounds the difference in virtual runtime increment of any pair of tasks for given balancing interval T. Suppose that task set S is executed on core P for interval $[kT,(k+1)T]$. If no task migration occurs during the interval, the physical runtime of each task is equal to the multiple of its time slice defined by (2). Thus,

$\Delta VR_i(T)$ for any task τ_i running on core P is computed as follows.

$$\Delta VR_i(T) = \frac{\omega_0}{\sum_{\tau_j \in S} W(\tau_j)} \times T \tag{7}$$

Assume that there is a pair of tasks (τ_i, τ_j) where the virtual runtime of τ_i is larger than that of τ_j. According to (7), $\Delta VR_{i,j}(T)$ for (τ_i, τ_j) is computed as

$$\Delta VR_{i,j}(T) = \left(\frac{\omega_0}{\sum_{\tau_i \in S_k} W(\tau_i)} - \frac{\omega_0}{\sum_{\tau_j \in S_l} W(\tau_j)} \right) \times T \tag{8}$$

where S_k is the task set which includes τ_i and S_l is the task set which includes τ_j. Note that the proposed algorithm guarantees that the load of S_k is equal to or larger than the load of S_l and the load difference between S_k and S_l is bounded by ω_{max}. Then, the following inequality holds from (8).

$$\left(\frac{\omega_0}{\sum_{\tau_j \in S_l} W(\tau_j) + \omega_{\max}} - \frac{\omega_0}{\sum_{\tau_j \in S_l} W(\tau_j)} \right) \times T \le \Delta VR_{i,j}(T) \le 0 \tag{9}$$

Since the load of any core cannot be smaller than ω_{min}, the following inequality also holds.

$$\left(\frac{\omega_0}{\omega_{min} + \omega_{max}} - \frac{\omega_0}{\omega_{min}} \right) \times T \le \Delta VR_{i,j}(T) \le 0 \tag{10}$$

Similarly, for task pair (τ_i, τ_j) where the virtual runtime of τ_i is smaller than that of τ_j, $\Delta VR_{i,j}(T)$ is also bounded as shown below.

$$0 \le \Delta VR_{i,j}(T) \le \left(\frac{\omega_0}{\omega_{min}} - \frac{\omega_0}{\omega_{min} + \omega_{max}} \right) \times T \tag{11}$$

The proof of the property of the algorithm is done by induction as demonstrated below. We start from the initial condition. When $t = 0$, virtual runtimes of all tasks are 0 and $D_{i,j}(0)$ equals to 0 for any pair of tasks (τ_i, τ_j). Thus, it satisfies the property. Now we show that, for any pair of tasks (τ_i, τ_j), if $D_{i,j}(kT)$ is smaller than a constant C for some k, then $D_{i,j}((k+1)T)$ is also smaller than C.

Consider the any pair of tasks (τ_i, τ_j) where the virtual runtime of τ_i is larger than that of τ_j. For this task pair, we assume that our algorithm satisfies that $D_{i,j}(kT) \le C$ when $t = kT$. When $t = (k+1)T$, $D_{i,j}((k+1)T)$ is computed as below.

$$D_{i,j}((k+1)T) = VR_i((k+1)T) - VR_j((k+1)T) \tag{12}$$

$$D_{i,j}((k+1)T) = D_{i,j}(kT) + \Delta VR_{i,j}(T) \tag{13}$$

According to (10), $\Delta VR_{i,j}(T)$ is bounded by a constant $C = \alpha T$ where α is the negative constant and it thus satisfies the property that $D_{i,j}((k+1)T) \le C$ for any pair of tasks. Similarly, in the case that the virtual runtime of τ_i is smaller than that of τ_j, the property is also satisfied since $\Delta VR_{i,j}(T) \le C$ holds where $C = \alpha T$ and α is the positive constant.

VI. Experimental Evalution

In this section, we report on the results of our experiments that we conducted to evaluate the fairness provided by the proposed algorithm. We also show a tradeoff between fairness and run-time overhead by varying the load balancing period.

A. Experimental setup

We have implemented the proposed algorithm into the Linux kernel 2.6.38.8. We have run it on the target system whose hardware and software components are shown in Table II.

In order to evaluate the fairness of the proposed algorithm, we ran a CPU-intensive task set consisting of six tasks. They executed the same infinite loop but were assigned different nice values -5, 5, 0, 0, 0 and 0. We then compared $D_{max}(t)$ of these tasks measured under the legacy CFS and the modified CFS, respectively. We also measured the physical runtime of each task and compared it to the ideal runtime that would have been given to the task under GPS.

In order to show the run-time overhead of the proposed algorithm, we ran the *Kernbench* benchmark program [19]. It creates multiple tasks to collaboratively compile the Linux 2.6.38.8 kernel source. We configured the number of tasks to be six and assigned a nice value of 0 to all of them. We then compared the total elapsed times measured under the legacy CFS and the modified CFS, respectively.

B. Experimental results

1) Evaluating the fairness

TABLE II. Hardware and Software Components of the Target System

Hardware	***CPU***	Intel® Core™ 2 Duo E8500 Clock: 3.16GHz
	Main memory	4-GB DDR2
Software	***Operating system***	Ubuntu 11.04 Kernel version: 2.6.38.8
	GNU libc	glibc 2.13

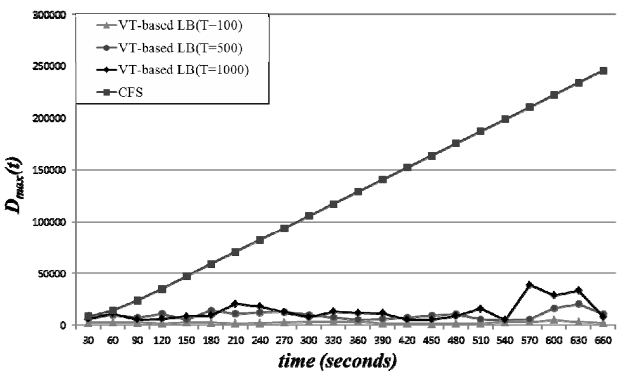

Figure 6. Comparison of $D_{max}(t)$ between unmodified Linux 2.6.38.8 and our approach

$D_{max}(t)$ measured under the legacy CFS and the modified CFS are shown in Fig. 6. The horizontal axis represents the wall-clock time while the vertical axis denotes $D_{max}(t)$. It is obvious from this result that $D_{max}(t)$ constantly increases in the legacy CFS while it remains bounded by a constant in the modified CFS. We also observe that smaller load balancing period T yields smaller $D_{max}(t)$. We set T equal to 100, 500 and 1000 milliseconds and these are marked in triangles, dots and diamonds, respectively. The maximum $D_{max}(t)$ when $T=100$ and 1000 are 50.54 and 387.08 time units, respectively. Note that one unit of time in the virtual runtime is equal to one millisecond in the physical runtime when a task has a nice value of 0.

Physical runtimes of the tasks are shown in Table III. In the legacy CFS, τ_1 is the most starved task in a sense that it is given 15.24% less CPU time compared to the ideal CPU time under GPS. On the other hand, in the modified CFS, every task received at least 97.04% of the ideal CPU time.

2) *Evaluating the run-time overhead*

We also evaluated the run-time overhead of our approach by measuring the total elapsed times for completing kernel compilation under the legacy CFS and the modified CFS, respectively. The result is shown in Table IV. It demonstrates that our approach achieves nearly identical performance compared to the legacy CFS. The extra overhead is only 0.14% of the legacy CFS when $T=100$. It even outperforms the legacy CFS in terms of incurred overhead when $T=500$ and 1000.

TABLE III. THE PHYSICAL RUNTIME OF THE TASK SET OBTAINED BY LINUX TOP COMMAND

	nice value	**weight**	***Legacy CFS***		***CFS modified by our algorithm***	
			Physical runtime (s)	***Diff. from GPS (%)***	***Physical runtime (s)***	***Diff. from GPS (%)***
τ_1	-5	3121	2336.55	-15.24	2649.41	-0.90
τ_2	5	335	313.30	+5.87	278.49	-2.96
τ_3	0	1024	1004.86	+11.09	884.19	+0.79
τ_4	0	1024	1011.86	+11.86	876.64	-0.06
τ_5	0	1024	1009.77	+11.60	891.21	+1.59
τ_6	0	1024	994.65	+9.96	889.51	+1.40

TABLE IV. OVERALL PERFORMANCE OF THE VIRTUAL RUNTIME-BASED TASK MIGRATIONALGORITHM

	Legacy CFS	***New CFS (T=100)***	***New CFS (T=500)***	***New CFS (T=1000)***
Compilation elapsed time (s)	48.678	48.748	48.442	48.348

VII. CONCLUSION

In this paper, we have presented a virtual runtime-based task migration algorithm that bounds the virtual runtime difference between any pair of tasks running on two cores. We have also formally analyzed the behavior of the Linux CFS to precisely characterize the reason why it fails to achieve the fairness in a multicore system. Our algorithm consists of two sub-algorithms: (1) PARTITION which partitions tasks into two groups depending on their virtual runtimes and (2) MIGRATE which allocates the partitioned groups to dedicated cores while minimizing the number of task migrations. We also proved that our algorithm bounds the maximum virtual runtime difference between any pair of tasks. We have implemented the algorithm into the Linux kernel and experimentally evaluated it. The results demonstrate that our algorithm improves the fairness in a multicore system while its run-time overhead is negligible.

There are future research directions along which our algorithm can be extended. We are looking to extend the algorithm so that it can be applied to a general multicore system with more than two cores. We also plan on extending our algorithm to achieve different objectives including performance, interactivity and power consumption.

ACKNOWLEDGMENT

The work reported in this paper is supported in part by the Technology Innovation Program (No. 10036495) funded by the Ministry of Knowledge Economy (MKE, Korea), by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) (No. 2011-0001572 and No. 2010-0027809), by Digital Media & Communications R&D Center, Samsung Electronics, Co. Ltd (No. 0418-20110011), by Advanced Institutes of Convergence Technology (AICT), Institute for Smart Systems and by Automation and Systems Research Institute (ASRI).

REFERENCES

[1] http://www.ibm.com/developerworks/linux/library/l-cloud-computing/

[2] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. "The eucalyptus open-source cloud-computing system". Proc. the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid,2009, pp. 124-131.

[3] Fontán, J., Vázquez, T., Gonzalez, L., Montero, R., and Llorente, I. "OpenNEbula: The open source virtual machine manager for cluster computing". Proc. Open Source Grid and Cluster Software Conference,2008, pp.

[4] Parekh, A.K., and Gallager, R.G.: "A generalized processor sharing approach to flow control in integrated services networks: the single-node case", IEEE/ACM Transactions on Networking (TON), 1993, 1, (3), pp. 344-357.

[5] Caprita, B., Chan, W.C., Nieh, J., Stein, C., and Zheng, H. "Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems". Proc. the annual conference on USENIX Annual Technical Conference 2005, pp. 337-352.

[6] Srinivasan, A., and Anderson, J.H.: "Fair scheduling of dynamic task systems on multiprocessors", Journal of Systems and Software, 2005, 77, (1), pp. 67-80.

[7] Chandra, A., Adler, M., Goyal, P., and Shenoy, P. "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors". Proc. Proceedings of the 4th conference on Symposium on Operating System Design & Implementation,2000, pp. 45-58.

[8] http://ck.kolivas.org/patches/bfs/bfs-faq.txt

[9] Anderson, J.H., and Srinivasan, A. "Early-release fair scheduling". Proc. 12th Euromicro Conference on Real-Time Systems,2000, pp. 35-43.

[10] Nieh, J., Vaill, C., and Zhong, H. "Virtual-time round-robin: An O (1) proportional share scheduler". Proc. the 2001 USENIX Annual Technical Conference,2001, pp. 245-259.

[11] Caprita, B., Nieh, J., and Stein, C. "Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling". Proc. the twenty-fifth annual ACM symposium on Principles of distributed computing 2006, pp. 72-81.

[12] http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[13] http://people.redhat.com/mingo/O(1)-scheduler/README

[14] Li, T., Baumberger, D., and Hahn, S.: "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin", ACM SIGPLAN Notices, 2009, 44, (4), pp. 65-74.

[15] http://www.hpl.hp.com/research/linux/kernel/o1-starve.php

[16] Pabla, C.S.: "Completely fair scheduler", Linux Journal, 2009, (184)

[17] Dejun, J., Pierre, G., and Chi, C.H. "EC2 performance analysis for resource provisioning of service-oriented applications". Proc. The 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing,2009, pp. 197-207.

[18] Raman, R., Livny, M., and Solomon, M.: "Matchmaking: An extensible framework for distributed resource management", Cluster Computing, 1999, 2, (2), pp. 129-138.

[19] http://ck.kolivas.org/kernbench/