Lag-based Load Balancing for Linux-based Multiprocessor Systems

Dongwon Ok[1], Byeonghun Song[1], Hyunmin Yoon[1], Peng Wu[1], Jaesoo Lee[2], Jungkeun Park[3], and Minsoo Ryu[4]

[1]Department of Electronics and Computer Engineering, Hanyang University, Seoul, Korea

[2]The-AiO, Seongnam, Gyoenggi, Korea

[3]Department of Aerospace Information Engineering, Konkuk University, Seoul, Korea

[4]Department of Computer Science and Engineering, Hanyang University, Seoul, Korea

Abstract - *In this paper, we present a lag-based load balancing approach to achieve global fairness with the Linux CFS (Completely Fair Scheduler). Lag of each task is defined as the ideal CPU time it should have received minus the actual CPU time it has received. The proposed approach monitors the lag of each task at runtime and moves tasks to under-loaded processors so that each task can bound its lag. We implemented the proposed approach in the Linux kernel and experimentally evaluated it. The results demonstrate that our algorithm shows significant fairness improvements.*

Keywords: Linux, Multi-core scheduling, Completely Fair Scheduler, Fairness, Load balancing

1 Introduction

The goal of fair scheduling is to share CPU resources among tasks so that each task receives CPU time proportional to its weight. Perfect fairness is generally impossible since it requires infinitesimal CPU quanta for scheduling. Most fair schedulers can provide approximate fairness attempting to minimize the gap between the ideal GPS (generalized processor sharing) [2] scheme and their actual one.

Since the Linux 2.6.23 kernel release, Linux introduced a fair scheduler, CFS (completely fair scheduler), replacing the O(1) scheduler. CFS is the first fair scheduler implemented in general purpose operating systems. Previously, most operating systems such as Windows and earlier Linux versions provide round-robin style sharing of CPU resources rather than weight-based proportional sharing. In contrast, CFS associates each task with a specific weight value determined by the task's nice value and attempts to assign CPU time proportionally. CFS uses the notion of virtual runtime to track the ratio of the actual CPU time each task has received and the ideal CPU time each task should have received. Scheduling decisions are made by finding one that has the minimum virtual runtime and thus CFS can guarantee proportional sharing of CPU time among tasks.

Unfortunately, CFS does not ensure global fairness for multiprocessor systems as Linux uses partitioned scheduling. Linux maintains a separate run queue for each processor and each run queue is scheduled by a separate CFS. Therefore, local fairness can be achieved by the CFS scheme on each processor while global fairness across multiple processors cannot be guaranteed by the CFS scheme. The Linux kernel attempts to mitigate this problem by balancing workload among processors, but this approach often results in unacceptable global fairness.

In this paper, we present a lag-based load balancing approach to achieve global fairness with CFS. We define lag as the ideal CPU time each task should have received minus the actual CPU time each task has received. Our proposed approach monitors the lag of each task at runtime and moves tasks across processors whenever their lag values seem to exceed a specified upper bound. By moving such tasks to under-loaded processors that are able to bound their lag values, our approach can provide global fairness on multiprocessor Linux systems.

The remainder of this paper is organized as follows. Section 2 describes the Linux CFS (completely fair scheduler), its load balancing mechanism and its limitation. Section 3 describes the lag-based load balancing algorithm. Section 4 concludes this paper.

2 Completely Fair Scheduler

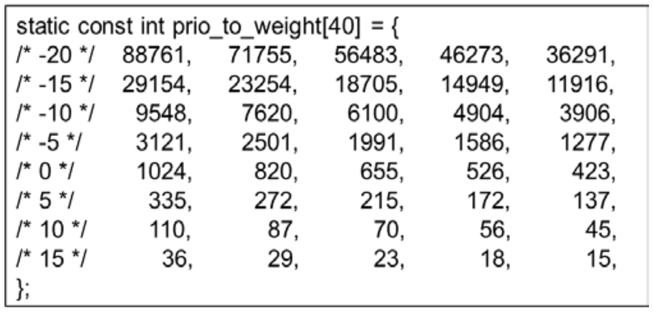
Completely Fair Scheduler (CFS) has been employed as the Linux scheduler since Linux 2.6.23 to provide weighted fairness for task scheduling. The weight of each task is the function of its nice value, integer value from -20 to 19, where a small nice value corresponds to a large weight value. Linux creates a separate run queue for each CPU and keeps track of virtual runtime for each task to represent the ratio of the actual CPU time the task has received and the ideal CPU time the task should have received. A smaller virtual runtime value indicates that the task has received less CPU time.

A red-black tree is used to find a task that has the smallest virtual runtime. The red-black tree places the task of the smallest virtual runtime at its leftmost leaf. Whenever CFS makes a scheduling decision, it selects the leftmost task from the red-black tree.

Let w^0 be the weight of nice value 0, w_i be the weight of task τ_i and $PR(\tau_i, t)$ be the CPU time consumed by task τ_i by time t. The virtual time of task τ_i by time t is defined as below.

$$\mathrm{VR}(\tau_{i,}t) = \frac{PR(\tau_i, t)}{w_i} \times w^0 \qquad (1)$$

Note that w^0 and w_i are determined from nice values, as shown in Figure 1 taken from ``sched.c'' in the Linux kernel source code.



```
static const int prio_to_weight[40] = {
/* -20 */   88761,   71755,   56483,   46273,   36291,
/* -15 */   29154,   23254,   18705,   14949,   11916,
/* -10 */    9548,    7620,    6100,    4904,    3906,
/* -5 */     3121,    2501,    1991,    1586,    1277,
/* 0 */      1024,     820,     655,     526,     423,
/* 5 */       335,     272,     215,     172,     137,
/* 10 */      110,      87,      70,      56,      45,
/* 15 */       36,      29,      23,      18,      15,
};
```

Figure 1. Mapping between nice values and weight values.

The main idea behind CFS is to achieve fairness by using virtual runtime values. However, in the current Linux kernel, virtual runtime values are not examined across CPUs, thus leading to unfairness from a global point of view. CFS performs weight-based load balancing to mitigate this problem, but this approach often results in unacceptable global fairness.

For load balancing, Linux defines the load of run queue as the sum of all task weights in a run queue and keeps its value as `load` in `struct rq`. It also specifies when to perform load balancing, usually every *k* scheduling ticks for a certain positive integer *k* in each scheduling domain. Scheduling domain is a set of CPUs that are managed by a single scheduling policy. Each scheduling domain may contain one or more CPU groups and each group may contain one or more CPUs. Linux tries to balance the load across CPU groups within a domain. At every scheduling tick, CFS checks if it needs to perform load balancing. If so, it starts load balancing by calling `load_balance()`. For each scheduling domain with `SD_LOAD_BALANCE` flag, it finds the busiest group by calling `find_busiest_group()`. Before finding the busiest one among the CPU groups, it checks again whether to proceed load balancing or not. The kernel performs load balancing only when the load of current group is sufficiently low. To do so, `find_busiest_group()` examines two cases. The first case is when the load of the current group is no less than the average load of scheduling domain. The second case is when the difference of the load of the current group and the maximum load in the scheduling domain does not exceed a certain imbalance value defined by `imbalance_pct` in `struct sched_domain`. When the load of current group is sufficiently low, `find_busiest_group()` function calculates the amount of load to move using the following imbalance metric.

$$L_{imbal} = \min(L_{max} - L_{avg}, L_{avg} - L_k) \qquad (2)$$

where L_{max} is the maximum load of the busiest group in the scheduling domain, L_{avg} is an average load in the system and L_k is the load of the current group. Linux checks again if the imbalance L_{imbal} is greater than twice of the smallest weight in the busiest run queue. If so, Linux moves tasks from the busiest group to the current under-loaded group.

3 Lag-based Load Balancing

In this section, we propose a lag-based load balancing approach to achieve global fairness for CFS on multiprocessor hardware. The proposed approach relies on the notion of lag. The lag is defined as the ideal CPU time each task should have received minus the actual CPU time each task has received [1]. Suppose that task τ is runnable and have a fixed weight in the interval $[t_1, t_2]$. Let $S_{\tau,A}(t_1, t_2)$ denotes the CPU time that task τ receives in $[t_1, t_2]$ under a certain scheduling scheme *A*, and $S_{\tau,GPS}(t_1, t_2)$ denotes the CPU time under the Generalized Processor Sharing (GPS) [2] scheme; an idealized scheduling model which achieves perfect fairness. For any interval $[\mathrm{t}_1, \mathrm{t}_2]$, the lag of task τ at time $\mathrm{t} \in [\mathrm{t}_1, \mathrm{t}_2]$ is formally defined as

$$lag_\tau(t) = S_{\tau,GPS}(t_1, t_2) - S_{\tau,A}(t_1, t_2). \qquad (2)$$

A positive lag at time t implies that the task has received less CPU time than under GPS, and a negative lag indicates that the task has received more CPU time than required. If all tasks in a run queue have positive lags, then this implies that the load of the run queue is relatively higher. Similarly, negative lags imply lower load. Let T_i be the time slice that task τ_i can consume without preemption. When task τ_i is not scheduled for T_i, the lag of task τ_i will increase by the amount of Δlag_i.

$$\Delta lag_i = T_i \times \frac{Weight_i}{\sum_{j\in\Phi} Weight_j} \times N \qquad (3)$$

where $Weight_i$ is the weight of task τ_i, Φ is the set of all the runnable tasks in the entire system, and N is the number of CPUs. As the average load of the entire system is

$$Average\ Load = \frac{\sum_{j\in\Phi} Weight_j}{N}, \qquad (4)$$

Δlag_i can also be defined as below.

$$\Delta lag_i = T_i \times \frac{Weight_i}{Average\ Load} \quad (5)$$

Since the lag increases consistently unless the task is scheduled, the time that the task should be scheduled can be calculated back from lag_i and Δlag_i. Let $laxity_i$ denote the remaining time until task τ_i exceeds a certain specified lag bound without being scheduled. The $laxity_i$ for any task τ_i is defined by

$$laxity_i = \left\lfloor \frac{lag\ bound - lag_i}{\Delta lag_i} \right\rfloor. \quad (6)$$

Note that tasks tend to have large lag values in a high load run queue. As time progresses, the lag values of one or more tasks will exceed a specified bound. To avoid this, we need to constantly monitor the lag values and check in advance if they will exceed the bound or not. Specifically, whenever the Linux kernel makes a scheduling decision for each run queue, the proposed approach checks if there exist more than one tasks that will have zero laxity at some identical time point. If found, only one of those tasks remains in the original run queue and other tasks are moved to less-loaded run queues. Figure 2 shows this algorithm in flowchart where count_laxity_zero(τ) checks if there exists more than one tasks that will have zero laxity at some identical time point, min_vruntime(CPU) is the value of minimum virtual runtime for the given CPU, and vruntime(τ) is the virtual runtime of the given task.

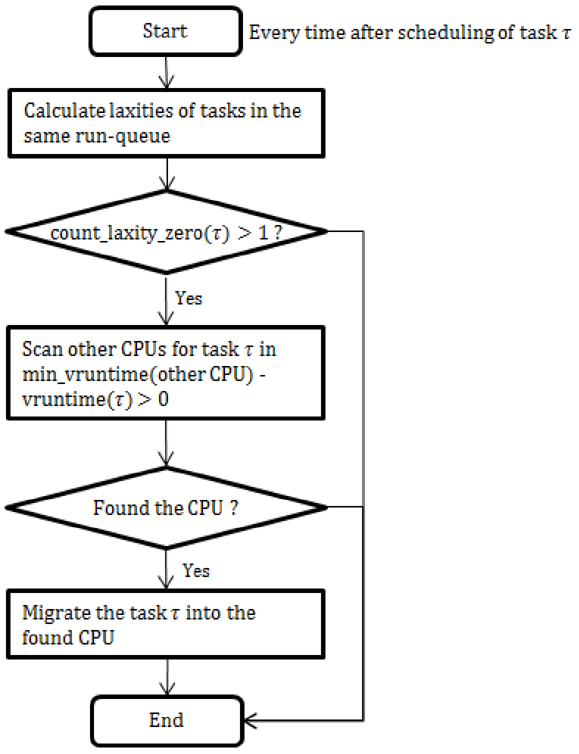

Start
Every time after scheduling of task τ
Calculate laxities of tasks in the same run-queue
count_laxity_zero(τ) > 1 ?
Yes
Scan other CPUs for task τ in min_vruntime(other CPU) - vruntime(τ) > 0
Found the CPU ?
Yes
Migrate the task τ into the found CPU
End

Figure 2. Flowchart of lag-based load balancing algorithm.

Whenever virtual runtime of the task is calculated, lag-based load balancing performs as follows:

(i) Lag-based load balancing algorithm calculates laxities of tasks in the same run queue.

(ii) If there are two or more tasks with laxity 0, tasks should be migrated.

(iii) The algorithm scans run queues of the other CPUs in min_vruntime(other CPU) – vruntime(τ) > 0. The virtual runtime of task τ should be lower than the minimum virtual runtime of other run queue, so the task τ will get chance to be scheduled right after migration.

If the suitable CPU is found, move the task τ into the run queue of the CPU.

4 Experimental Evaluation

We conducted to evaluate proposed load balancing algorithm in terms of the fairness. The algorithm was implemented in the Linux kernel 2.6.34.13. We used `ideal_time` value, which is calculated by Linux kernel to measure a dynamic time slice to check preemption, as a lag bound. Our experiments were performed on Ubuntu 10.10. In order to evaluate the fairness of the proposed algorithm, we ran four compute-intensive tasks with different weights, 1024, 335, 335 and 335. Let $D_{max}(t)$ be difference in virtual runtime of two tasks, between the task with the largest virtual runtime and the one with the smallest at time t. Since the virtual runtime of task has a concept of weighted CPU time, $D_{max}(t)$ represents unfairness; the lower $D_{max}(t)$ is, the fairer the algorithm is. The experimental result represented by the graph in Figure 3 shows that our approach enhances the fairness.

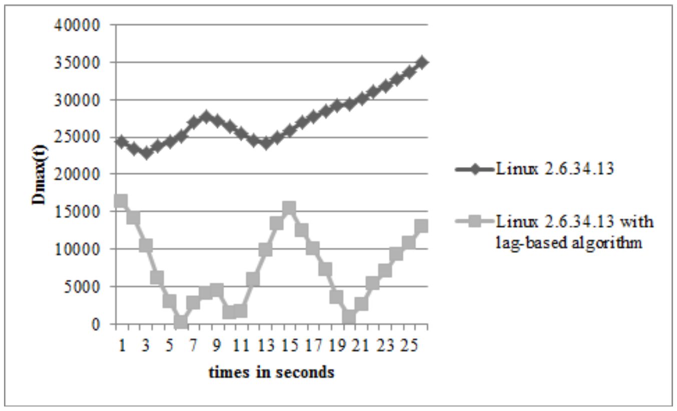


Figure 3. Comparison of $D_{max}(t)$ between legacy Linux 2.6.34.13 and the one with lag-based algorithm.

5 Conclusions

In this paper, we proposed a lag-based load balancing scheme to guarantee global fairness in Linux-based multiprocessor systems. The proposed approach introduces the notion of lag and provides fairness across multiple processors through lag-based load balancing. We also implemented the proposed approach in the Linux kernel and experimentally evaluated it. The results demonstrate that our algorithm shows significant improvement in terms of fairness.

6 Acknowledgement

This work was supported partly by Seoul Creative Human Development Program (HM120006), partly by Mid-career Researcher Program through NRF (National Research Foundation) grant funded by the MEST (Ministry of Education, Science and Technology) (NRF-2011-0015997), partly by the IT R&D Program of MKE/KEIT [10035708, “The Development of CPS (Cyber-Physical Systems) Core Technologies for High Confidential Autonomic Control Software”], and partly by the MSIP(Ministry of Science, ICT&Future Planning), Korea, under the CITRC(Convergence Information Technology Research Center) support program (NIPA-2013-H0401-13-1008) supervised by the NIPA(National IT Industry Promotion Agency).

7 References

[1] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate progress: A notion of fairness in resource allocation,” *Algorithmica,* vol. 15, no. 6, pp. 600-625, 1996.

[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 (TON),* vol. 1, no. 3, pp. 344-357, 1993.

[3] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, "Fairness and interactivity of three CPU schedulers in Linux." pp. 172-177.

[4] T. Li, D. Baumberger, and S. Hahn, “Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin,” *Acm Sigplan Notices,* vol. 44, no. 4, pp. 65-74, Apr, 2009.

[5] S. Huh, J. Yoo, M. Kim, and S. Hong, "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm." pp. 606-614.