Fixed Share Scheduling via Dynamic Weight Adjustment in Proportional Share Scheduling Systems

Hyungwoo Kim¹, Hyunmin Yoon², Peng Wu², and Minsoo Ryu¹
¹Department of Computer Science and Engineering, Hanyang University, Seoul, Korea
²Department of Electronics Computer Engineering, Hanyang University, Seoul, Korea

Abstract - There exist many compute-intensive soft real-time applications, such as video encoding/decoding and data encryption/decryption, that require a fixed percentage of CPU cycles to maintain an acceptable QoS (Quality of Service). Unfortunately, major operating systems do not support well such compute-intensive soft real-time applications, since they commonly rely on a proportional share policy. In this paper, we present a novel scheduling policy, called FSS (fixed share scheduling), to enable a proportional share scheduler to support compute-intensive soft real-time applications as well as non-real-time applications. The goal of FSS is to guarantee an absolute, constant share of CPU cycles for soft real-time tasks regardless of workload conditions, whereas traditional proportional share schedulers focus on relative proportional guarantees. To do so, working on top of a proportional share scheduler, FSS dynamically changes the weight value of each soft real-time task to match the demanded amount of CPU share under varying workload conditions. The weighted fairness mechanism of the underlying proportional share scheduler will then provide the demanded amount of CPU share for each soft real-time task. To demonstrate the efficacy of FSS, we have implemented a fixed shared scheduling prototype in the Linux CFS (completely fair scheduler) and conducted experiments to show the correctness and efficiency of the FSS scheme.

Keywords: Fixed Share Scheduling, Proportional Share Scheduler, QoS, soft-real-time.

1 Introduction

There exist many compute-intensive soft real-time applications, such as video encoding/decoding and data encryption/decryption, that require a fixed percentage of CPU cycles to maintain an acceptable QoS (Quality of Service). For example, a HVEC (High Efficiency Video Coding) decoder demands around 60% of CPU utilization on ARM Cortex A9 processor at 1.5 GHz. When the CPU cycle demand cannot be met, the provided QoS will be significantly degraded.

Unfortunately, such compute-intensive soft real-time applications are not well supported by major operating systems. Most operating systems use a proportional share scheduling policy, often combined with a priority-based scheduling policy. The common goal of these schedulers is to achieve fairness and responsiveness as a general purpose operating system scheduler, rather than providing any resource guarantees for real-time applications.

In this paper, we present a novel scheduling policy, called FSS (fixed share scheduling), to enable a proportional share scheduler to support compute-intensive soft real-time applications as well as non-real-time applications. The goal of FSS is to guarantee an absolute, constant share of CPU cycles for soft real-time tasks regardless of workload conditions, whereas traditional proportional share schedulers focus on relative proportional guarantees. To do so, working on top of a proportional share scheduler, FSS dynamically changes the weight value of each soft real-time task to match the demanded amount of CPU share under varying workload conditions. The weighted fairness mechanism of the underlying proportional share scheduler will then provide the demanded amount of CPU share for each soft real-time task.

To demonstrate the efficacy of FSS, we have implemented a fixed shared scheduling prototype in the Linux CFS (completely fair scheduler). We have also conducted experiments to show the correctness and efficiency of the FSS scheme.

2 Fixed Share Guarantees with Proportional Share Schedulers

Proportional share scheduling provides a useful abstraction for multiplexing resources among different tasks [10]. The key idea of proportional share scheduling is to allocate resources to tasks proportional to their weights. The ideal model of proportional share resource allocation is the Generalized Processor Sharing (GPS) scheme. GPS assumes a fluid-style resource model and guarantees perfect fairness based on an infinitesimal fluid resource model. For real systems where resources cannot be provided infinitesimally, approximate scheduling schemes like WFQ (Weighted Fair Queuing) and PGPS (Packet by Packet GPS) have been proposed. Figure 1 shows the ideal proportional share scheduling and approximate proportional scheduling.
Let $Weight_i$ be the weight of task $\tau_i$ and let $\Phi$ be the set of all active tasks at time $t$. The share $S_i(t)$ of a task $\tau_i$ at time $t$ is defined as below.

$$ S_i(t) = \frac{Weight_i}{\sum_{j \in \Phi} Weight_j} $$

(1)

As the number of tasks can change at runtime, the resource share $S_i(t)$ also changes. For example, when a new task arrives, the total weight $\sum_{j \in \Phi} Weight_j$ of tasks will increase, decreasing the share $S_i(t)$ of task $\tau_i$. Therefore, a proportional share scheduler only guarantees a relative share of CPU cycles depending on the workload condition, rather than an absolute, fixed share of CPU time.

The goal of FSS is to guarantee a constant amount of CPU utilization for soft real-time tasks regardless of dynamic changes in workload. This idea can be easily implemented in a proportional share scheduling system by simply changing the weight values of soft real-time tasks during runtime. Let $U_i$ be the CPU utilization demanded by a soft real-time task $\tau_i$. The actual CPU utilization allocated to task $\tau_i$ is determined by weight values of other non-real-time tasks as follows.

$$ U_i = \frac{Weight_i}{\sum_{j \in \Phi, j \neq i} Weight_j + Weight_i} $$

(2)

A simple arithmetic manipulation gives the following equation.

$$ Weight_i = \frac{U_i}{1 - U_i} \times \sum_{j \in \Phi, j \neq i} Weight_j $$

(3)

Therefore, we can determine the new weight value of task $\tau_i$ using Eq. (3) for a given value of fixed share demand $U_i$.

3 Incorporating Fixed Share Scheduling into Linux CFS

In this section, we demonstrate that the proposed idea can be incorporated into the Linux completely fair scheduler (CFS).

CFS has been introduced since Linux 2.6.23 to provide weighted fairness. CFS uses virtual runtime for each task to keep track of the actual CPU time the task has received and the ideal CPU time the task should have received. Let $VR(\tau_i, t)$ be the virtual runtime of task $\tau_i$ at time $t$ and $A(\tau_i, t)$ be the CPU time consumed by the task $\tau_i$ by time $t$. Let $Weight_0$ be the weight for nice value 0.

$$ VR(\tau_i, t) = \frac{Weight_0}{Weight_i} \times A(\tau_i, t) $$

(4)

Note that $Weight_i$ is determined by the nice value of task $\tau_i$. The weight value has a range from 88761 (nice level -20) to 15 (nice level 19) where small nice values have large weights. The nice values are stored in the `prior_to_weight[]` array in "sched.h" of the Linux kernel source code as shown in Figure 2.

Note that CFS guarantees weighted fairness by allocating different time slices to tasks of different weights. Specifically, the length of the time slice $TS_i$ of $\tau_i$ is proportional to its weight as below.

$$ TS_i = \frac{Weight_i}{\sum_{j \in \Phi} Weight_j} \times P $$

(5)

where $P$ is the length of round in which the scheduler executes every task only once scanning the run queue. It is defined by the following.

$$ P = \begin{cases} 
\text{systcl_sched_latency, if } n < \text{systcl_nr_latency} \\
\text{systcl_sched_min_granularity} \times n, \text{ otherwise}
\end{cases} $$

(6)

where $n$ is the total number of tasks in the run queue, `systcl_sched_latency` and `systcl_sched_min_granularity` are constants specified as 6ms and 0.75ms, respectively, in our Linux setting, and `systcl_nr_latency` is defined by Eq. (7).

$$ systcl_nr_latency = \begin{cases} 
\text{systcl_sched_latency} \\
\text{systcl_sched_min_granularity}
\end{cases} $$

(7)

which becomes 8 as `systcl_sched_latency` is 6ms and `systcl_sched_min_granularity` is 0.75ms.

A fixed share scheduling scheme can be easily implemented in Linux by slightly modifying the scheduler to perform weight recalculation. Since CFS can also be viewed as an approximate implementation of GPS, the implementation of fixed share scheduling in CFS is not much
different from implementing it in other proportional share schedulers. A major difference is that CFS uses discrete weight values ranging from 15 to 88761, which requires us to find the nearest weight value that is greater than the recalculated weight value. For example, when a new weight value should be 8100, we need to choose 9548 and assign -10 as a new nice value.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>CPU</th>
<th>Samsung Exynos5 Octa big.LITTLE processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>2 GByte LPDDR3 RAM</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>Operating System</td>
<td>Android 4.2.2 Jelly Bean, Kernel Version : 3.4.5</td>
</tr>
<tr>
<td></td>
<td>GNU gcc</td>
<td>arm-eabi-gcc (GCC) 4.6.x-google 20120106</td>
</tr>
</tbody>
</table>

Figure 3. Implementation environment.

4 Experimental Evaluation

We have implemented and incorporated FSS into Linux CFS. Figure 3 summarizes the development environment. We then conducted a set of simple experiment to validate the correct behavior of our fixed share scheduling scheme.

Figure 4. CPU utilization and nr_running in the original Linux CFS scheduler

In the first set of experiments, we observed the utilization of a target task with nice value of 0 under the original CFS scheme. Figure 4 shows that the utilization of target task decreases as the number of tasks, nr_running, increases.

In the second set of experiments, we observed the utilization of a target task under the proposed fixed share scheduling scheme. The fixed share demand of target task was set to be 50% of CPU utilization. Figure 5 shows that the actual utilization of target task is maintained around 50% even though the number of tasks varies dynamically.

Figure 5. CPU utilization, nr_running, and nice value in the proposed FSS scheme

5 Conclusion

In this paper, we have presented a FSS (fixed share scheduling) that can guarantee an absolute, constant share of CPU cycles for soft real-time tasks. FSS works on top of a traditional proportional share scheduler and dynamically changes the weight value of each soft real-time task to guarantee the demanded amount of CPU share under varying workload conditions. To demonstrate the efficacy of FSS, we have also implemented a fixed shared scheduling prototype in the Linux CFS (completely fair scheduler). Our evaluation shows that FSS provides accurate fixed share guarantees under varying workload conditions.

6 Acknowledgment

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


