Proportional Share Scheduling employing Performance-aware Virtual Time in Multiprocessor Systems

Munseok Kim[1], Hyunmin Yoon[2], and Minsoo Ryu[1]
[1]Department of Computer Science and Engineering, Hanyang University, Seoul, Korea
[2]Department of Electronics Computer Engineering, Hanyang University, Seoul, Korea
{mskim, hmyoon}@rtcc.hanyang.ac.kr, msryu@hanyang.ac.kr

Abstract – *In proportional share scheduling, the different performances of CPUs can make running tasks unfair during a period. To make them fair in a system, we present a proportional share scheduling employing performance-aware virtual time (PVT) maintained globally. This PVT is the share of CPU time received by a task and increases at a rate proportional to the performance of CPU where the task is running on and inversely proportional to the weight of the task. The schedulers of CPUs, when they assign CPU time to a task, utilize PVT to make a decision which task and how long it should preempt CPU to minimize the difference of PVTs among tasks. We evaluated our approach experimentally on general purpose operating system in the homogeneous and heterogeneous multiprocessor (HMP) systems. On both systems, the results show the significant improvement that is near-perfect (around 99% better) fairness in the homogeneous multiprocessor system and much better (more than 60% better) in the HMP system.*

Keywords: Proportional share scheduler, virtual time, performance-aware, heterogeneous multiprocessor, fairness.

1 Introduction

Proportional share scheduling which provides abstractions for multiplexing resources among tasks allocates resources to a task proportional to its weight to guarantee the weighted fairness in a system. Unfortunately, generally this fairness cannot be completely achieved in practice because infinitesimal CPU quanta are required in theory. To minimize the difference of CPU time between a task ideally needed in theory and actually received, previous works have introduced various approaches such as [3] and [12]. These approaches, however, do not consider that the unfairness among tasks can arise also by the different performance of CPUs. In practice, the unfairness arises in the homogeneous multiprocessor system which has CPUs of different frequencies like x86-based, and it is more obvious in case of heterogeneous multiprocessor (HMP) systems.

The HMP system such as ARM big.LITTLE processor was introduced to make an energy-aware scheduling possible by processing tasks on the core of less energy consumed. In such a system, since each CPU has different capacity and frequency, unless schedulers consider the performance of CPUs, the unfairness could be amplified and more frequent than the homogeneous. Nevertheless, in HMP related works, most focus is on the performance optimization, energy-saving and showing the benefit of them [6]-[8], while not much effort is being given to guarantee the fairness among tasks.

The fairness among tasks is significantly important factor to guarantee quality-of-service (QoS) of multi-program workloads [9], [10]. For example, applications such as immersive virtual environments and interactive multi-media can lead to unpredictable and undesirable result because they require real-time computation and communication services from the operating system on the assumption that all tasks make equal progress on CPUs. Yet, this expectation cannot be guaranteed in the case of that a task running on a big core (or the higher frequency of CPU) works more than the other on a small core (or the lower frequency of CPU). In this case, finally, the difference of work done among tasks should be minimized with the consideration about the performance of CPUs to guarantee QoS based on the fairness.

To achieve this goal with considering about the frequency and capacity of CPUs as major factors of the performance, we present a proportional share scheduling employing performance-aware virtual time (PVT). PVT, a virtual time of a task maintained in a system widely, increases at a rate proportional to the performance of CPU where the task is running on and inversely proportional to the weight of the task based on CPU time received by the task. This PVT makes the unfairness among tasks traceable relatively, and schedulers utilize it when they assign CPU time to a task. By leveraging existing proportional share scheduler employing PVT, this work provides near-perfect fairness (more than 99%) in the homogeneous multiprocessor system and much better fairness (more than 60%) in the HMP system than the previous one.

The remainder of this paper is composed of several sections as follows. Section 2 describes an existing proportional share scheduling and its limitation, and Section 3 introduces PVT and how to utilize it in a system. In section 4, we show substantially improved results based on the completely fair scheduler (CFS) employing PVT in Linux as a representative fair scheduler in practice. Section 5 concludes this paper with the consideration of future works.

2 Proportional Share Scheduler and Limitation

Proportional share scheduling which is able to provide abstractions for multiplexing resources among tasks is allocating resources to a task proportional to its weight to guarantee weighted fairness in a system [1]. Proportional share resource allocation is ideally generalized processor sharing (GPS) scheme [2]. A fluid-style resource and prefect fairness based on an infinitesimal fluid resource model are assumed, but actual system cannot provide resource infinitesimally in practice. Therefore, approximate scheduling scheme is being proposed like packet by packet GPS (PGPS) [4], and weighted fair queuing (WFQ) [2]. Figure 1 shows the ideal scheme and quantum-based scheduling which is able to be implemented to achieve proportional share scheduling in practice [11].

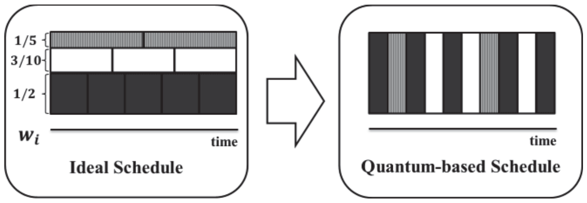

Figure 1. Proportional share scheduling.

Let W_i be the weight of task τ_i and Φ be the set of all active tasks at time t. The share $S_i(t)$ of a task τ_i at time t is defined as follows.

$$S_i(t) = \frac{W_i}{\sum_{j\in\Phi} W_j}.$$

The share $S_i(t)$ is changeable in runtime because the number of tasks in a system can be changed dynamically. For example, if a new task is initialized, the total weight $\sum_{j\in\Phi} W_j$ is increased and the share $S_i(t)$ of task τ_i is decreased on the contrary. Therefore, a proportional share scheduler only guarantees a relative share of CPU time according to the total weight changes.

To measure the difference of CPU time received by a task between the ideally needed and the actually received, virtual-time domain [1] can be utilized. In this domain, the virtual time is the share of CPU time received by a task. The share of CPU time is allocated to a task proportional to the weight of the task. Therefore, the virtual time can be computed as follows.

$$VT(\tau_i, t) = A(\tau_i, t) \times \int_0^t \frac{\sum_{j\in\Phi(t')} W_j}{W_i} dt'.$$

Let $A(\tau_i, t)$ be the CPU time assigned to task τ_i by time t, and let $\Phi(t')$ be the set of all tasks active at time t'. Because the virtual time increases at a rate proportional to the sum of weights of all tasks, if the total sum of weights increases, the virtual time of τ_i increases faster and vice versa.

Additionally, the *lag* is defined as the ideal CPU time which should be assigned to a task by subtracting the actual CPU time received by a task. Suppose that task τ is active and have a fixed weight in the interval $[t, t']$. Let $S_{\tau,A}(t, t')$ denotes the CPU time received by the task τ in $[t, t']$ under a certain scheduling scheme A, and $S_{\tau,GPS}(t, t')$ denotes the CPU time under the Generalized Processor Sharing (GPS) scheme; an idealized scheduling model which achieves perfect fairness. The *lag* of task τ at time t ($t \in [t, t']$), for any interval $[t, t']$, is formally defined as

$$lag_\tau(t) = S_{\tau,GPS}(t, t') - S_{\tau,A}(t, t').$$

However, in the case of proportional share schedulers based on partitioned scheduling, they have each run queue individually and try to guarantee the fairness with the consideration about the weights of tasks only within the run queue where the scheduler involved in. Although this approach has no problem in a system which has the same performance of CPUs, it can incur a problem in a system which has different performance of CPUs. In such a system, as well as the sum of weights of tasks on each CPU, the performance of each CPU can be different by the dynamic frequency changes or the static capacity of processors.

In case of the different frequency of CPUs, for example, consider four tasks τ_1, τ_2, τ_3 and τ_4 which have the same weight value 100 individually on the dual-core processor which has CPUs 1 and 2. The frequency of CPU 1 is 1 GHz while it of CPU 2 is 2 GHz. They can operate as following scenarios.

1. Four tasks τ_1, τ_2, τ_3 and τ_4 start simultaneously at time 0
2. Execute τ_1 and τ_2 in CPU 1, τ_3 and τ_4 in CPU 2 during 2 sec
   a. τ_2 moved to CPU 2 without delay
   b. τ_3 moved to CPU 1 without delay
3. Execute τ_1 and τ_3 in CPU 1, τ_2 and τ_4 in CPU 2 during 2 sec
   a. τ_3 moved to CPU 2 without delay
   b. τ_2 moved to CPU 1 without delay
4. Execute τ_1 and τ_2 in CPU 1, τ_3 and τ_4 in CPU 2 during 2 sec

In this case, all of tasks can receive the same amount of CPU time; however, at the point of the amount of work done, it can be totally different result. If we consider the performance of CPUs, according to the proportion of performance of each CPU, we can describe the relative ratio of the amount of work done by each task based on CPU time like follows. The τ_1 processed $(2 + 2 + 2)$ of work in CPU 1, τ_2 processed $(2 + 2 \times 2 + 2)$ and τ_3 processed $(2 \times 2 + 2 + 2 \times 2)$ in both CPUs while the τ_4 processed $(2 \times 2 + 2 \times 2 + 2 \times 2)$ in CPU 2. Finally, the amount of work done by each task is in the order of $\tau_1 < \tau_2 < \tau_3 < \tau_4$. If the length of execution time at the scenario *2, 3,* and *4* could be manipulated properly, τ_2 and τ_3 can be fair while τ_1 is of the least work done and τ_4 is of the most. Even though the total sum of weights in each CPU during execution is absolutely balanced, the result of work done among tasks can be totally different by the performance of CPUs and it eventually leads tasks to the unfairness. Finally to guarantee the fairness among tasks in a system, as well as the sum of weight of tasks, the performance of CPUs should be considered together.

3 Performance-aware Virtual Time

In this section, we propose performance-aware virtual time (PVT) where the performance of CPU is defined using two major factors: frequency and capacity. PVT, as a virtual time of a task maintained globally, increases at a rate proportional to the performance of CPU where the task is running on and inversely proportional to the weight of the task based on CPU time received by the task. Additionally, we utilize PVT to monitor the fairness measure in a system. Basically our approach considers both systems homogeneous and heterogeneous multiprocessor (HMP), and each CPU individually has a dynamic voltage and frequency scaling (DVFS) which is efficient technology for dynamic power management (DPM). Therefore, PVT of task τ_i of weight W_i in CPU P is calculated as follows.

$$PVT_P(\tau_i\,,t) = \frac{A(\tau_i,t)\times W_{max}}{W_i}\times\frac{C_P\int_0^t F_P\,(x)dx}{F_{max}\times t}\,.$$

Let W_{max} and F_{max} be the maximum weight of a task and CPU frequency acceptable in a system, and $\int_0^t F_K\,(x)dx$ be the total amount of frequencies by time t, and C_P $(0 \le C_P \le 1)$ is a constant value of CPU P which indicates the relative ratio of the performance among CPUs from the fastest ($C_P = 1$). This constant value can be determined depending upon the types of tasks running on a system based on the results of various benchmarks or some specific metrics reported by chip vendors also.

Additionally, if we scale W_{max} and F_{max} as a same value to reduce a fraction, we finally can simplify equation as follows.

$$PVT_P\big(\tau_j\,,t\big) = \frac{A\big(\tau_j,t\big)}{W_j}\times C_P\int_0^t F_P\,(x)dx.$$

All tasks in a system have their own PVT values. By utilizing these values, we are able to monitor the change of fairness in a system by the changes of results of periodic repeating follows.

$$MaxDiff(t) = \max_{\tau\in\Phi} PVT_P(\tau,t)\; - \min_{\tau\in\Phi} PVT_P(\tau,t)\,.$$

The maximum result of $MaxDiff(t)$ should be maintained as a similar level. This means eventually that the results are always bounded into the specific value, and the level of fairness measure is being maintained in a system.

The difference (lag) of CPU time received by the task τ_i between the ideally needed and the actually received by time *t* based on the performance of CPU *P* can be derived as follows.

$$lag_{\tau_i}(t) = \left(\frac{\sum_{j\in n}\left(\int_0^t F_j\,(x)dx\,\times\,C_P\right)}{\sum_{l\in\Phi} W_l\times t}\times W_i - \frac{\int_0^t F_P\,(x)dx}{W_i}\right)\times A(\tau_i,t)\,.$$

Let n be the number of CPUs in a system. When $lag_{\tau_i}(t)$ is greater than zero, τ_i received less time than the time ideally needed, and in case of zero the time was ideally received and the more time received in the other case; however, operating system cannot reclaim CPU time from a task that has already received. Therefore, we utilize PVT to find a task with the lower PVT and give more CPU time prior to the tasks with higher PVT in a system to minimize the difference of PVTs.

In case of a I/O intensive task or a task of starting, its PVT is revised exceptionally because PVT of a task out of run queue can be maintained as the lowest without system progression applied when they are coming back to run queue again. To make a decision of revision needed or not, we classifies two groups of tasks according to the state transition of task. In case of task transition between ready and running, tasks should be revised, and the other cases except for the first are not revised. Therefore, PVTs of I/O tasks of the latter case need to be revised by subtracting as much time as system progressed during being out of run queue. Finally PVTs of the tasks is revised to keep the previous position in a system widely as

$$PVT_M(\tau_i\,,t') = \left(\min_{\tau\in\Phi} PVT_P(\tau,t') - \min_{\tau\in\Phi} PVT_P(\tau,t)\right) + PVT_P(\tau_i\,,t)\,.$$

The I/O intensive task τ_i is out from the run queue of CPU P at time t and inserted into the run queue of CPU M at time t'. In this case, the minimum PVT of each time t and t' is subtracted and added to keep the previous level of fairness position in a system.

4 Experimental Evaluation

Completely fair scheduler (CFS) is the most popular and the first fair scheduler applied to the general purpose operating system while the other operating systems like Windows and Linux of earlier version (before the version of 2.6.23) are providing round-robin scheduling. By these reasons, to achieve our goal, we utilized existing CFS in Linux (after the version of 2.6.23) to employ performance-aware virtual time (PVT). PVT of each task was also utilized to monitor the fairness measure in a system. To evaluate the effectiveness of our proposed approach, we considered two different environments both homogeneous multiprocessor system and heterogeneous multiprocessor (HMP) system. The characteristics of these are shown in Tables 1 and 2.

Table 1. homogeneous multiprocessor environment

Hardware	CPU	Intel i7-4770 3.4 GHz Dual processor
	RAM	4 GB DDR3 SDRAM
Software	Operating System	Ubuntu 14.04 (on VMWare), Linux Kernel Version : 3.18.2

Table 2. heterogeneous multiprocessor environment

Hardware	CPU	Samsung Exynos5 Octa big.LITTLE processor
	RAM	2 GB LPDDR3 RAM
Software	Operating System	Android 4.4.4 Kit Kat, Linux Kernel Version : 3.10.9

To evaluate the fairness among tasks between operating systems the original version and the version of PVT employed in the system Table 1, we created simple application whose performance is mostly proportional to the frequency of CPU to make C_P simply be 1 in all CPUs. The execution time of each instance was compared to measure the fairness among tasks. This application just repeats infinite loop simply and print out the cumulative average time consumed at every 2.5 billionth loop. This cumulative average execution time was approximately 1.3 sec in practice. Three instances of the application were executed concurrently in the system shown in Table 1, and we got the results 10 times per 1 min from the original version and our version of PVT applied in Linux kernel 3.18.2.

Even though there is not a significant difference in the execution of three instances evaluated on the homogeneous dual-core processor, as depicted in Figure 2, the different execution time of tasks arises and being kept consistently. Interestingly, this experiment was evaluated in the one of most popular general purpose operating system and processors even though the type of application was not very common.

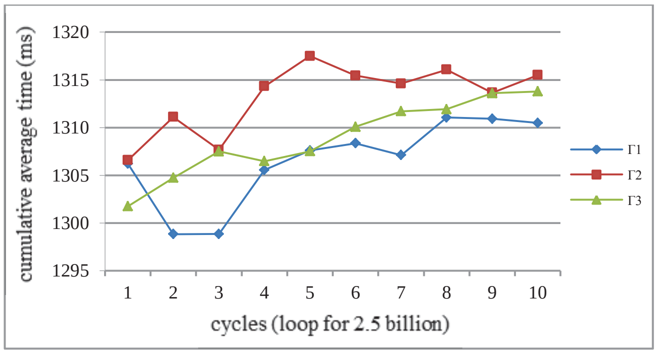

Figure 2. cumulative average time of cycles on Ubuntu 14.04 LTS of kernel 3.18.2

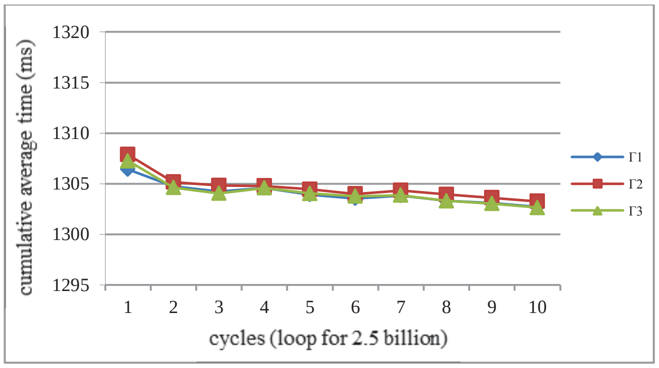

Figure 3. cumulative average time of cycles after PVT applied

CFS employing PVT in the version of inux kernel 3.18.2 achieves near-perfect fairness among three tasks on the homogeneous dual-core processor as depicted in Figure 3 while the unfairness arises obviously in Figure 2. When the time t is 100, Figure 4 shows that $MaxDiff(t)$ of three tasks is bounded in 3 sec (of PVT) approximately.

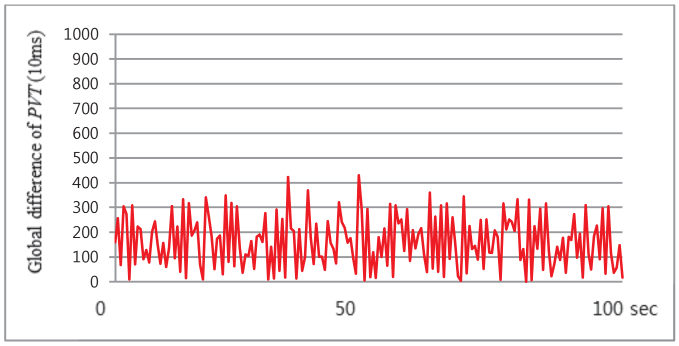

Figure 4. Periodic repeats of *MaxDiff(t)* of 3 tasks in the homogeneous during 100 sec

In case of the system in Table 2, we created a similar application, but the number of loop operation was changed to 1 million, and a sleep code for 5ms was inserted after printing out (per about 850ms in practice) to simulate the effects of I/O operation together. Twelve instances of this application were created and executed concurrently, and we gathered the results 10 times per 1 min.

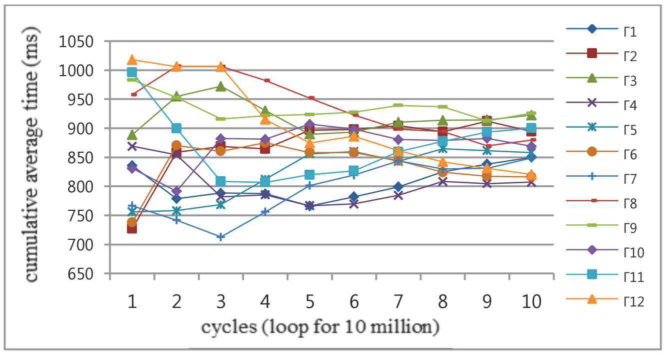

Figure 5. Cumulative average time of cycles on Android KitKat of kernel 3.10.9

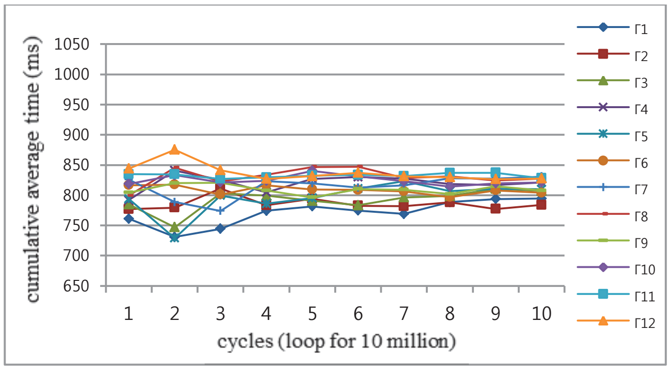

Figure 6. Cumulative average time of cycles after PVT applied

Figure 5 shows the results when we measure the cumulative average time of each task using the existing kernel. There is a lot of variation among the execution times of tasks even though all tasks are identical. In Figure 6, when compared to Figure 5, the cumulative average time values of the twelve tasks are more narrowed down. Based on the results, the maximum deviation of the cumulative average time can be shown as Figure 7.

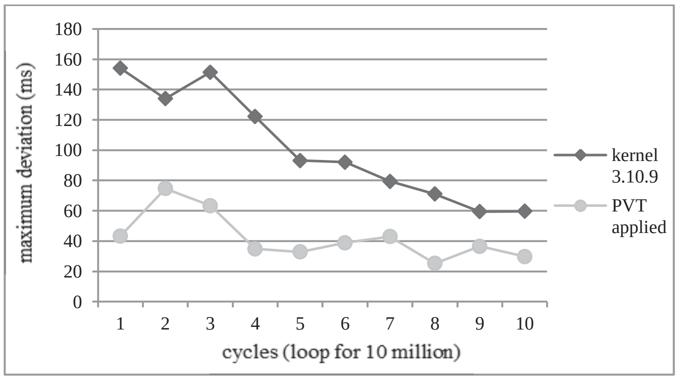

Figure 7. maximum deviation of cumulative average time

Figure 7 shows that PVT applied kernel guarantees the fairness among tasks by giving at least 60% better than the original kernel in HMP system. During 600 sec, $MaxDiff(t)$ of the twelve tasks are bounded into 5 sec as shown in Figure 8.

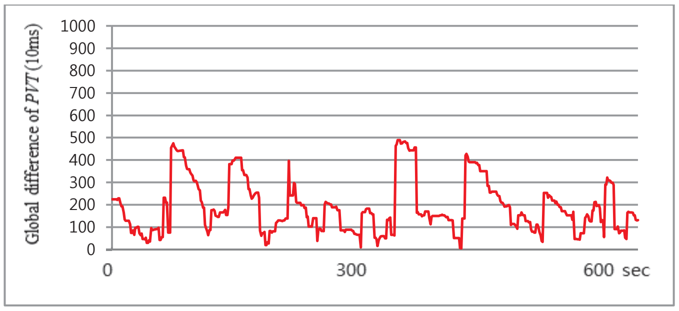

Figure 8. Periodic repeats of *MaxDiff(t)* of 12 tasks in the HMP system during 600 sec

Additionally, according to the each result of $MaxDiff(t)$, the fairness among three tasks in the homogeneous is being guaranteed better than it of twelve tasks in the heterogeneous as described above results.

5 Conclusion

In this paper, we proposed a proportional share scheduling employing performance-aware virtual time (PVT) to guarantee and measure the fairness among tasks in a system. The proposed approach leads the fairness to the significantly improved than previous systems. We also introduced how to monitor the fairness by utilizing PVT to compare the level of fairness measure in a system.

Additionally, if we consider the remaining 40% in the results of HMP system, there could be more factors such as the number of heterogeneous CPUs and migration, the type of tasks, miss-rate of cache, the policy of load balancer and so forth which may affect the fairness of tasks in a system. As a future goal, we plan to focus on the relationship between these factors and the fairness guarantees. As a future work, we also intend to determine the capacity of CPU which can affect the fairness measure.

6 Acknowledgment

This work was supported partly by Seoul Creative Human Development Program (HM120006), partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (NRF-2011-0015997), and partly the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the C-ITRC (Convergence Information Technology Research Center) (IITP-2015-H8601-15-1005) supervised by the IITP(Institute for Information & communications Technology Promotion).

7 References

[1] NIEH, Jason; VAILL, Christopher; ZHONG, Hua. Virtual-Time Round-Robin: An O (1) Proportional Share Scheduler. In: *USENIX Annual Technical Conference, General Track*. 2001. p. 245-259.

[2] MARKATOS, Evangelos P.; LEBLANC, Thomas J. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1994, 5.4: 379-400.

[3] CHANDRA, Abhishek, et al. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000. p. 4-4.

[4] PAREKH, Abhay Kumar; GALLAGER, Robert G. A generalized processor sharing approach to flow control in integrated services networks-the single node case. In:

INFOCOM'92. Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE. IEEE, 1992. p. 915-924.

[5] KUMAR, Rakesh, et al. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In: *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003. p. 81-92.

[6] KOUFATY, David; REDDY, Dheeraj; HAHN, Scott. Bias scheduling in heterogeneous multi-core architectures. In: *Proceedings of the 5th European conference on Computer systems*. ACM, 2010. p. 125-138.

[7] SHELEPOV, Daniel, et al. HASS: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 2009, 43.2: 66-75.

[8] GREENHALGH, Peter. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

[9] OGRAS, Umit Y.; MARCULESCU, Radu. " It's a small world after all": NoC performance optimization via long-range link insertion. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2006, 14.7: 693-706.

[10] BONALD, Thomas; MASSOULIÉ, Laurent. Impact of fairness on Internet performance. In: *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2001. p. 82-91.

[11] KIM, Hyungwoo, et al. Fixed Share Scheduling via Dynamic Weight Adjustment in Proportional Share Scheduling Systems.

[12] D. Ok, B. Song, H. Yoon, P. Wu, J. Lee, J. Park, and M. Ryu, “Lag-Based Load Balancing for Linux-based Multicore Systems,” *The 2013 International Conference on Foundations of Computer Science*, July 2013.