Information Engineering Express
International Institute of Applied Informatics
2015, Vol. 1, No. 2, 1 – 10

Effect of Thread Weight Readjustment Scheduler on Scheduling Criteria

Samih M. Mostafa[†*], Shigeru Kusakabe[†]

Abstract

In this paper, we propose a new approach to significantly optimize some scheduling criteria, context switches and turnaround times in this context, when running multithreaded processes concurrently. Current proportional sharing schedulers allocate CPU time based on the weights of running threads in the system and don't take into consideration the greedy behavior of multithreaded processes (processes with more threads receive more aggregate CPU time from the scheduler relative to processes with fewer threads). In order to minimize turnaround times and context switches of running multithreaded processes in simultaneous multithreaded (SMT) architectures, we investigate the effect of adjusting the weights of running sibling threads (threads forked from the same process) using novel proportional sharing scheduler, Thread Weight Readjustment Scheduler (TWRS), a proportional share CPU scheduler designed for multithreaded processes, which aims to reduce undesirable events (e.g. context switches) and turnaround times. TWRS provides a practical solution for multitasking operating systems because it operates in concert with existing kernels. We have implemented TWRS in Linux 2.6.24-1, which represents the most prevalent scheduler design (i.e. Completely Fair Scheduler (CFS)). Our evaluation shows that our scheduler minimizes context switches and turnaround time.

Keywords: Multithreaded processes, multitasking, CFS, fairness, turnaround time.

1 Introduction

A. General Overview

Proportional Share Scheduler (PSS), also sometimes referred to as a fair-share scheduler might try to guarantee that each process obtains a certain percentage of CPU time. PSS is a type of scheduling which preallocates certain amount of CPU time to each process. PSS is based upon maintaining fairness between competing processes. PSSs are often primarily evaluated based on the level of fairness that they can provide [28]. Other evaluation criteria depend on what the scheduler is designed for.

Cooperative multitasking and *preemptive* multitasking are two flavors in which multitasking operating systems come in. Linux, like all Unix variants and most modern operating systems,

[*]Mathematics Department, Faculty of Science, SVU Uni., Qena, Egypt
[†]Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan

implements preemptive multitasking. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to begin running. The act of involuntarily suspending a running process is called preemption. The time a process runs before it is preempted is usually predetermined, and it is called the *timeslice* of the process. The timeslice, in effect, gives each runnable process a slice of the processor's time [4].

B. Scheduling and its Criteria

The scheduling problem can be stated shortly as: *which* thread should be moved to *where*, *when* and *for how long* [4][6]. In computer science, a scheduling algorithm is the method by which threads, processes or data flows are given access to system resources (e.g. processor time). This is usually done to load balance a system effectively or achieve a target quality of service [5][6]. Software known as a scheduler and dispatcher carry out this assignment.

Scheduling algorithms have been found to be NP-complete in general form (i.e., it is believed that there is no optimal polynomial-time algorithm for them [12][13]). The give-and-take situation comes from the many definitions of good scheduling performance, such that improving performance in one sense hurts performance in another. Some improvements to the Linux scheduler help performance all-around, but such improvements are getting more and harder to come by [16].

Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following: CPU utilization, throughput, fairness, execution time, turnaround time, context switches, completion time, waiting time, and others [7][8][9].

Context switches and turnaround time are two of the most important criteria in designing any operating system scheduler. The context switches time is a pure overhead because the system does no useful work while switching. Reducing such overhead leads to minimizing turnaround time.

C. Paper Organization

The rest of this paper is structured as follows: section 2 discusses the contribution of this paper. Current Linux kernel scheduler, CFS, is discussed in section 3. Section 4 presents the problem statement. In section 5, we discuss the related research. Section 6 discusses TWRS. The experimental setup and scheduling modes are given in section 7. Section 8 presents the evaluations.

2 Research Contribution of this Paper

In this paper, we proposed and evaluated TWRS, which is a proportional share CPU scheduling, intending to minimize the context switches and turnaround times of processes. In proportional share algorithm every thread has a weight, and thread receives a share of the available resources proportional to its weight [18].

In this work, a modification is implemented to CFS. This modification is based on changing threads' weights of sibling threads created in the same process and assigning a specific time slice to each of these sibling threads.

We have implemented our scheduler in the Linux kernel and experimentally demonstrated the improvement of our scheduler over the current scheduler, CFS, using multithreaded programs in Sysbench benchmark [27]. Our experimental results show that TWRS scheduler minimizes context switches and turnaround times.

3 Linux Kernel Schedulers

A. Completely Fair Scheduler

Completely Fair Scheduler (CFS) is introduced by Ingo Molnár to replace the O(1) scheduler [17][18][19][20]. This scheduler was designed to provide good interactive performance while maximizing overall CPU utilization [21]. It also strives to provide fairness in every process without sacrificing the interactivity performance [22]. This is done by giving a fair amount of CPU time to each process proportional to its priority. This method is also called proportional share algorithm, where a share is allocated for each process, which is associated with the process's weight.

CFS is successor of the O(1) scheduler and one of the most distinct changes from previous scheduler is the policy of setting the priority for each thread. In CFS, the scheduler counts the execution time of each thread and calculates the priority as vruntime (virtual runtime). CFS sets the higher priority for the threads with less vruntime. The run queue of CFS is composed of Red-Black tree, where each node represents the thread and the value of each node represents the vruntime of each thread [14].

4 Problem Statement

This section discusses the main problem the current scheduler faces.

A. Overview

Each process and thread is a task in the eyes of the Linux scheduler. CFS uses thread fair scheduling algorithm, which allocates CPU resources between running threads in the system not between the running processes. In the current scheduler, CFS, when a new process is created, it appears as a thread, where both PID (Process Identification) and TGID (Thread Group Identification) are the same (new) number, although when a thread starts another thread, that new thread gets its own PID, so the scheduler can schedule it independently, and inherits its TGID from its parent as shown in Figure 1.

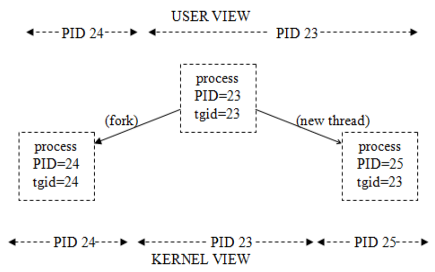

Figure 1. Identifications of process and thread created from parent process.

Therefore, CFS scheduler does not distinguish between threads and processes, and that way, the kernel can happily schedule threads independent of which process they belong to. Each forked thread is assigned a weight which determines the share of CPU bandwidth that thread will receive. Greedy users could take advantage by spawning more additional threads in order to obtain larger CPU resources.

In other words, we can summarize this statement as: the default Linux scheduler is process-agnostic and allows for greedy behavior, where processes with more threads may receive more aggregate CPU time from the scheduler relative to processes with fewer threads. That is, the scheduler schedules threads only and does not take process membership into account, or inter-process fairness.

From the point of view of users, one of the important criteria is how long scheduler takes to execute their process. Turnaround time, total time between submission of a process and its completion, is one of the scheduling criteria that the scheduler is concerned mainly with [16]. Turnaround time depends on the size of the time slice allotted to running processes, and minimizing turnaround time is one of the objectives that the scheduler strives to achieve. Previous work focused on the fairness criterion, however, in our work we focus on turnaround time and context switches.

5 Related Research

Generalized Processor Sharing (GPS) [1][26] is an idealized scheduling algorithm that achieves perfect fairness, and all schedulers use it as a reference to measure fairness. A scheduler is perfectly fair if it allocates CPU time to threads in exact proportion to their weights.

Chandra [2] presented Surplus Fair Scheduling (SFS), a proportional-share CPU scheduler designed for symmetric multiprocessors. Chandra first showed that the infeasibility of certain weight assignments in multiprocessor environments results in unfairness or starvation in many existing proportional-share schedulers. They presented a novel weight readjustment algorithm to translate infeasible weight assignments to a set of feasible weights. They showed that weight readjustment enables existing proportional-share schedulers to significantly reduce, but not eliminate, the unfairness in their allocations.

Chee [10] proposed an algorithm based on weight readjustment of the threads created in the same process. This algorithm, Process Fair Scheduler (PFS), is proposed to reduce the unfair allocation of CPU resources in multithreaded environment. Chee assumed that the optimal number of threads, best number to create in a process in order to have the best performance in a muti-processing environment, equals to the number of available cores. PFS changes the weight of thread according to the equation:

$$weight(thread) = \frac{weight(process)}{\alpha}$$

where α equals the number of threads created in the process.

A modification of PFS algorithm has been proposed to overcome the limitation of PFS by implementing Thread Fair Preferential Scheduler (TFPS) [23]. TFPS shall give the greedy threaded process (i.e. process tries to dominate most CPU time) the same amount of CPU bandwidth as optimally threaded process, and both of their time slices are larger than the single-threaded process.

Inter-Core Time Aggregation Scheduler (IAS) [1] executes two scheduling policies at the same time. The first scheduling policy is the Time Aggregation Scheduler (TAS) [3][11], which executes sibling threads in a row on a single core. The second scheduling policy is the inter-core aggregation, which executes sibling threads on different cores at the same time by dividing each core into master and slave cores. IAS lets every core cooperatively aggregate sibling threads by making slave cores follow the aggregation on master core. The basic idea of TAS and IAS schedulers is to dynamically give a priority bonus to the sibling thread of the currently executed threads. The priority of a thread is higher when the vruntime of the thread is smaller. Therefore, the priority bonus for the scheduler works to reduce the vruntime of the sibling thread.

6 Thread Weight Readjustment Scheduler

This section gives an overview of TWRS, and discusses its mechanism.

A. Overview

TWRS is a kernel-level thread scheduler to enhance system's performance running multithreaded processes by readjusting the weights of threads forked from the same process to significantly achieve better some scheduling criteria. TWRS works on top of an existing scheduler that uses run queues for per-CPU, such as Linux 2.6. As its name suggests, TWRS depends on proportionally distributed CPU time between threads by changing their weights. We will explain the policy of allocation CPU time to running threads in the next section.

B. TWRS and Thread Allocation of CPU Time

Each thread is assigned a weight which determines the share of CPU bandwidth that thread will receive [10]. The weight of thread *i* is given by,

$$w_i = \{-20 \le nice \le 19 : NICE\_0\_LOAD.(1.25^{-(\mathrm{nice})})\}$$

nice denotes the thread priority assigned by the user, and the value of 'nice' lies within [-20,19]. The default value of 'nice' is '0' in which case w_i=NICE_0_LOAD, and its value is assumed to be 1024.

The share in a time interval *[t_1, t_2]* of a runnable thread *i* is a ratio of its weight to the sum of weights of all active threads in the run queue. This is computed as:

$$Share_i(\mathrm{t}_t, \mathrm{t}_2) = \frac{w_i}{\sum_{\forall j \in R} w_j}(\mathrm{t}_2 - \mathrm{t}_1). \qquad (1)$$

R is the set of all runnable threads that are currently in run queue of a core. w_i is the weight of the thread, it is mapped from its nice value in prio_to_weight[], (defined as a variable in kernel/sched.c file) [24]*, and each nice value has its respective weight.*

The time-slice that a thread *i* should receive in a period of time is given by:

$$slice_i = share_i \times period \qquad (2)$$

where period is the time quantum the scheduler tries to execute all threads, by default this is set to 20ms [15][25].

If the number of runnable processes does not exceed

$$\frac{sched\_latency\_ns}{sched\_min\_grnularity\_ns}$$

then

$$period = sched\_latency\_ns$$

otherwise,

$$period = number\_of\_running\_tasks \times sched\_min\_granularity\_ns$$

By default;

$$sched\_latency\_ns = 20ms$$

and

$$sched\_min\_granularity\_ns = 4ms$$

sched_min_granularity_ns is a scheduler tuneable, this tuneable decides the minimum time a thread will be allowed to run on CPU before being preempted out. sched_latency_ns is a scheduler tuneable. sched_latency_ns and sched_min_granularity_ns decide the scheduler period.

In our consideration, the scheduler counts the number of total threads in the CPU and the number of sibling threads. The weights of the threads will be changed according to the next equation;

$$thrd->se.load.weight = new\_weight \qquad (3)$$

where *new_weight* is the new weight of the current thread and calculated from the equation:

$$new\_weight = p->se.load.weight \times (prcsr->totl\_thrds - curr\_proc->nr\_thrds) \qquad (4)$$

where p->se.load.weight is the weight of the current thread, prcsr->totl_thrds is the total number of threads in the current processor and curr_proc->nr_thrds is the number of threads in the current process.

7 Experimental Setup

A. Underlying Platform

TWRS can be easily integrated with an existing scheduler based on per-CPU run queues. We implemented TWRS in Linux version 2.6.24-1 which based on CFS. The specification of our experimental platform is shown in Table I.

TABLE I. SPECIFICATION OF OUR EXPERIMENTAL PLATFORM

H/W	
Processor	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
CPU cores	8
Memory	8186756 kb
S/W	
Kernel name	Linux
Kernel version number	2.6.24
Machine hardware name	x86_64 (64 bit)
version of Linux	CentOS release 5.10 (Final)

B. Scheduling Modes

To assess the performance of the modified kernel, we evaluated using 2 multithreaded benchmarks on a SMP system. The benchmarks run under two distinct scheduling modes: (1) The default scheduling in the Linux kernel and (2) The modified kernel.

- In the default scheduling mode, the benchmarks run on the original operating system where the scheduler is allowed to make scheduling decisions. No extra parameter is given to the scheduler to change its native scheduling algorithm.
- The second mode is accomplished in the new modified kernel, where the scheduler operates on the new scheduling policy to give new time slices to running threads.

Because we need to benchmark the scheduler performance, we choose only two test modes, `threads` and `cpu`, amongst all test modes available in Sysbench. In `threads` test a scheduler has a large number of threads competing for some set of mutexes.

In `cpu` test, each request consists in calculation of prime numbers up to a value specified by the --cpu-max-primes option.

We evaluated TWRS in two major scenarios, S0 and S1 in each mode as explained in Table II. The following test was conducted with no other computation intensive applications running.

- In S0, we intended to show that two concurrently executing instances of the same program, threads program in Sysbench, one instance with a fixed number of threads and the other variably from N to 64 threads, where N is greater than the number of threads in the fixed instance. Both programs were initiated at the same time and were initiated through a shell script where these two programs were executed with the same nice value 0. We repeat this simulation with a different number of threads in the fixed one.
- In S1, we intended to show that two concurrently executing instances of different programs, threads and cpu programs in Sysbench, threads program run with a fixed number of threads and cpu program variably from N to 64 threads, where N is greater than the number of threads in the fixed instance. Both programs were initiated at the same time and were initiated through a shell script where these two programs were executed with the same nice value 0. We repeat this simulation with a different number in the fixed program (threads).

TABLE II. S0 AND S1 SCENARIOS

S0		No. of threads in Fixed program "threads"	No. of threads in Varied program "threads"
	S0.1	2	4
			8
			16
			32
			64
	S0.2	4	8
			16
			32
			64
	S0.3	8	16
			32
			64

S1		No. of threads in Fixed program "threads"	No. of threads inVaried program "cpu"
	S1.1	2	4
			8
			16
			32
			64
	S1.2	4	8
			16
			32
			64
	S1.3	8	16
			32
			64

8 Experimental Results

To demonstrate the effectiveness of TWRS, we present some experimental data quantitatively comparing TWRS performance against the popular scheduler CFS considered on different combinations of number of threads for each scenario. We repeated this simulation many times for each sub-scenario and show the result in terms of number of context switches and average turnaround time. Our results show a significant improvement in terms of minimizing turnaround time and context switches. We divided our results into six sub-scenarios according to the previous consideration in Table II. For each sub-scenario, the two programs were run concurrently 20 times and the average values were taken. Figures 2 and 3 show the results of turnaround comparison in S0 and S1 respectively, and figures 4 and 5 show the results of context switches comparison in S0 and S1 respectively.

Concluding Remarks

In this paper a novel scheduler is proposed to distribute CPU time proportionally between threads according to their weights. We proposed TWRS which preallocates certain amount of CPU time to each thread of the multithreaded processes. We focused in this work on context switches and turnaround time. The scheduler was implemented and evaluated under specific hardware and software environment. We used Sysbench benchmark in our test and run under two distinct scheduling modes; the default scheduling in the Linux kernel and the modified kernel. Our results showed that the proposed scheduler achieves better results in minimizing turnaround time and context switches.

Acknowledgments

This work was partly supported by KAKENHI, Grant-in-Aid for Scientific Research(C) 25330084.

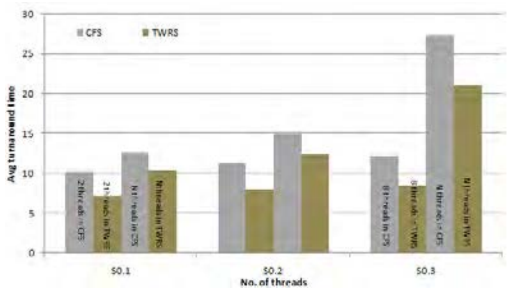

Figure 2. Average turnaround comparison of fixed number of threads of “threads” program vs. N threads of “threads” program in each sub-scenario in S0.

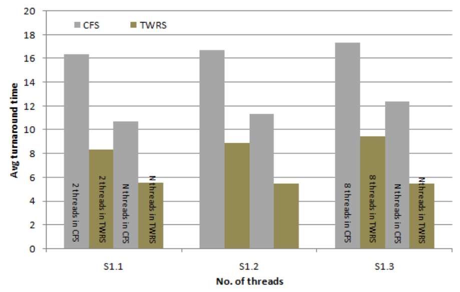

Figure 3. Average turnaround comparison of fixed number of threads of “threads” program vs. N threads of “cpu” program in each sub-scenario in S1.

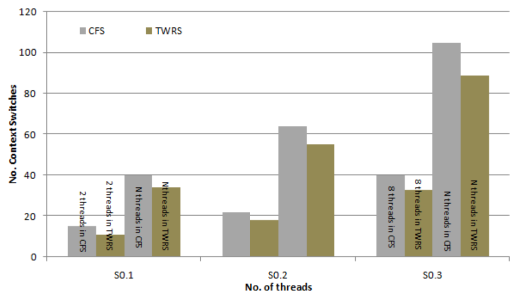

Figure 4. Context switches comparison of fixed number of threads of “threads” program vs. N threads of “threads” program in each sub-scenario in S0.

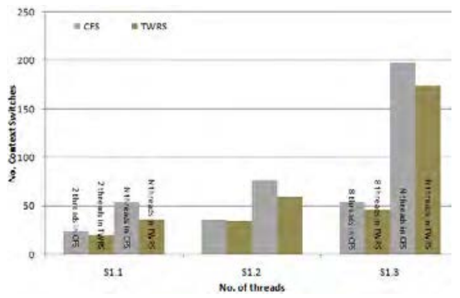

Figure 5. Context switches comparison of fixed number of threads of “threads” program vs. N threads of “cpu” program in each sub-scenario in S1.

9 References

[1] Satoshi Yamada, Shigeru Kusakabe, “Development of A Thread Scheduler for Global Aggregation of Sibling Threads, ” In Research Reports on Information Science and Electrical Engineering of Kyushu University, Vol.13, No.2, pp.69-74, 2008.

[2] Abhishek Chandra , Micah Adler , Pawan Goyal and Prashant Shenoy, "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, p.4-4, October 22-25, San Diego, California, 2000.

[3] Hiroaki Takasaki, Samih M. Mostafa, Shigeru Kusakabe, "Applying Eco-Threading Framework to Memory-Intensive Hadoop Applications," 5th International Conference on Information Science and Applications (ICISA 2014), seoul, Korea, May, 2014.

[4] Love R., "Linux Kernel Development," 3nd edition, Noval Press, ISBN 0-672-32720-1, 2010.

[5] Samih M. Mostafa, S.Z. Rida, S.H. Hamad, "Finding Time Quantum of Round Robin CPU Scheduling Algorithm in General Computing Systems Using Integer Programming," International Journal of Research and Reviews in Applied Sciences (IJRRAS) 5 (1), pp. 64–71, 2010.

[6] Hussain Hameed ,Malik Saif Ur Rehman,Hameed Abdul,Khan Samee Ullah,Bickler Gage,Min-Allah Nasro,Qureshi Muhammad Bilal,Zhang Limin,Yongji Wang,Ghani Nasir,Kolodziej Joanna,Zomaya Albert Y.,Xu Cheng-Zhong,Balaji Pavan,Vishnu Abhinav,Pinel Fredric,Pecero Johnatan,Kliazovich Dzmitry,Bouvry, Pascal,Li Hongxiang,Wang Lizhe,Chen Dan,Rayes Ammar, "A Survey on Resource Allocation in High Performance Distributed Computing Systems. Parallel Computing," vol. 39, no. 11, pp. 709-736, 2014.

[7] Silberschatz A, Galvin PB, Gagne G, "Operating Systems Concepts," John Wiley and Sons. 9Ed. 2013.

[8] Samih M. Mostafa and Shigeru Kusakabe, "Towards Minimizing Processes Response Time in Interactive Systems," International Journal of Computer Science and Information Technology Research (IJCSITR). Vol. 1, Issue 1, pp. 65-73, 2013.

[9] Samih M. Mostafa, Hamad SH and Rida SZ, "Improving Scheduling Criteria of Preemptive Tasks Scheduled under Round Robin Algorithm using Changeable Time Quantum," J Comput Sci Syst Biol. 2011.

[10] Wong C. S., Tan I., Kumari R. D., and Wey F., "Towards achieving fairness in the Linux scheduler," In: SIGOPS Oper. Syst. Rev. 42, 5, pp. 34-43, July, 2008.

[11] Yamada S. and Shigeru Kusakabe, "Effect of Context Aware Scheduler on TLB," In: Workshop on Multi-Threaded Architectures and Applications, Published in CD, 2008.

[12] Jeffrey D. Ullman, "Polynomial Complete Scheduling Problems," In: Proc. of the fourth ACM symposium on Operating system principles, pp. 96 – 101, 1973.

[13] Ramamritham, K., and Stankovic, J. A., "Scheduling Algorithms and Operating Systems Support for Realtime Systems," In: Proceedings of the IEEE, vol. 82, no. 1, 1994.

[14] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel,". 3rd edition, O'Reilly Press, ISBN 0-596-00565-2, 2005.

[15] Jacek Kobus and Rafal Szklarski, "Completely Fair Scheduler and its tuning," http://www.fizyka.umk.pl/jkob/prace-mag/cfs-tuning.pdf, 2009.

[16] Josh Aas., “Understanding the Linux 2.6.8.1 CPU scheduler,” Silicon Graphics, Inc., 2005.

[17] Kevin Jeffay, F. Donelson Smith, Arun Moorthy and James Anderson, “Proportional Share Scheduling of Operating System Services for Real-Time Applications,” In: Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, pp 480-491, Dec., 1998.

[18] Eric Schulte Taylor Groves and Jeff Knockel, “BFS vs. CFS Scheduler Comparison,” http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf, Dec., 2009.

[19] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss: Redline: First class support for interactivity in commodity operating systems. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 73–86, 2008.

[20] Prajakta Pawar, S.S.Dhotre and Suhas Patil, “CFS for Addressing CPU Resources in Multi-Core Processors with AA Tree,” International Journal of Computer Science and Information Technologies, Vol. 5 (1) , pp. 913-917, 2014.

[21] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann, “Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux,” In: Proceedings of the 3rd Workshop on Computer Architecture and Operating System Co-design (CAOS), Paris, France 2012.

[22] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah,Johannes E. Gehrke and C. Greg Plaxton, “A Proportional Share Resource Allocation Algorithm for Real-time and Time-shared Systems,” In: Proc. 17th IEEE Real-Time Systems Symp., Dec., 1996.

[23] Wong C. S., Tan I.K.T., Kumari R.D. and Kalaiyappan K.P., “Iterative Performance Bounding for Greedy-Threaded Process,” In: TENCON IEEE Region 10 Conference, Singapore, 2009.

[24] http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/kernel/sched.c?v=2.6.25#L1183

[25] Wong C.S., Tan I.K.T., Kumari R.D., Lam J.W. and Fun W., “Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers,” In: Information Technology, ITSim, 2008.

[26] Tong Li, Dan Baumberger and Scott Hahn, “Efficient and Scalable Multiprocessor Fair Scheduling using Distributed Weighted Round-Robin,” In Proc. of the 14th ACM Symposium on Principles and Practice of Parallel Programming, Feb., 2009.

[27] http://www.linuxcertif.com/man/1/sysbench/

[28] John Regehr, “Some Guidelines for Proportional Share CPU Scheduling in General-Purpose Operating Systems,” Work in Progress Session of the Proc. IEEE Real-Time Systems Symp. (RTSS ',01), Dec., 2001.