International Journal of New Computer Architectures and their Applications (IJNCAA) 4(4): 184-192
The Society of Digital Information and Wireless Communications, 2014 (ISSN: 2220-9085)

Effect of Thread Weight Readjustment Scheduler on Fairness in Multitasking OS

Samih M. Mostafa[1, 2] and Shigeru KUSAKABE[1]
[1]Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka Fukuoka, 819-0395, Japan.
[2]Math. Dept., Faculty of Science, South Valley University, Qena, Egypt.
E-mail: samih@nanotsu.ait.kyushu-u.ac.jp
kusakabe@ait.kyushu-u.ac.jp

ABSTRACT

In this paper, we investigate the effectiveness of Thread Weight Readjustment Scheduler (TWRS) for multitasking operating systems from the view point of fairness. Fairness is one of the most important criteria in designing any operating system scheduler. TWRS is a proportional share CPU scheduler designed explicitly for scheduling multithreaded processes depending on weight readjustment. We show that weight readjustment enables existing scheduler to significantly reduce, but not eliminate, the unfair in its allocations. TWRS preallocates certain amount of CPU time to each thread of the running multi-threaded processes. The scheduler was implemented and evaluated under specific hardware and software environment. We implement our scheduler in the Linux kernel and demonstrate its efficacy through an experimental evaluation. According to our evaluation results, our scheduler is promising to optimize some scheduling criteria, fairness in this context. We conclude from our results that TWRS is practical and desirable for general-purpose operating systems.

KEYWORDS

Fairness, Simultaneous Multithreading, CFS Linux scheduler, thread-level parallelism (TLP).

I. INTRODUCTION

A. Overview

Symmetric multiprocessing (SMP) systems have allowed several processes to run concurrently by providing multiple physical processors. However, a recent practice in computer hardware has been to place multiple processor cores on the same physical chip, resulting in a multicore processor. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor. Current operating systems are designed to be multitasking OSs to get the most benefit from hardware architecture.

Multitasking (or time sharing) is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A multitasking operating system allows many processes to share the computer simultaneously. As the system switches rapidly from one process to the next, each process is given the impression that the entire computer system is dedicated to its use, even though it is being shared among many processes [16].

A multitasking operating system is one that can simultaneously interleave execution of more than one process. On single processor machines, this gives the illusion of multiple processes running concurrently. Multicore machines enable processes to actually run concurrently, in parallel, on different cores. Multitasking operating systems come in two flavors: *cooperative multitasking* and *preemptive multitasking*. Linux, like all Unix variants and most modern operating systems, 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.

Allowing multiple threads to run concurrently in a processor and share resources, CPU time, is a recent architectural technique that improves resource utilization. Simultaneous multithreading (SMT) is one of the architectural techniques has become widespread concern in recent years that improves resource utilization [1]. Simultaneous Multithreading (SMT) architectures execute instructions from multiple streams of execution (threads) each cycle to increase instruction level parallelism [25, 26, 27]. Parallelism is one of the most important issues in multicore systems because it offers great performance, and future computing is progressing towards use of multicore machines. One of the efficient and scalable way to benefit from multicore architecture is using of multi-threading. Multi-threading is the process of executing multiple threads concurrently on cores and it is a widespread programming and execution model that allows multiple threads to exist within the context of a single process and these threads share the process's resources, but are able to execute independently.

An operating system is a software application that acts as an interface between user and the computer hardware. Its major responsibilities are to manage and ensure proper operations of the hardware resources [16]. At any time, there can be more than one runnable threads/processes demanding service from the Central Processing Unit (CPU). In order to handle the oversubscription of threads/processes, an operating system scheduler is required to ensure each thread/process obtains its share of CPU time as fair as the scheduling algorithm can guarantee [3].

B. Motivation

The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one thread/process at a time) and multiplexing (transmit multiple flows simultaneously). Scheduling is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. Nowadays, Single-threaded processor performance is becoming power limited, so processor architects are increasingly turning to multi-core designs to improve processor performance [28]. Due to the growing availability of chip multiprocessors (CMP), software applications are encouraged to be designed using multiple threads so that the benefit of thread-level parallelism (TLP) can be exploited [22]. The operating system scheduler is designed to allocate system resources, CPU time, proportionally to all processes.

The scheduling problem can be stated shortly as: *which* thread should be moved to *where, when* and *for how long* [5, 9]. 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 [10, 11]. 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 many definitions of good scheduling performance often lead to a give-and-take situation, 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.

The scheduler is the basis of a multitasking operating system such as Linux. By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously.

Managing the timeslice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the processor. On many modern operating systems, the timeslice is dynamically calculated as a function of process behavior and configurable system policy.

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

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, waiting time, and others [16, 29]. Fairness is one of the most important criteria of any operating system scheduler. Fairness is the ability to distribute CPU resources to each running process according to their weights.

Kernel hackers try to make improvements in Linux scheduling algorithm, this is because Linux scheduling algorithm is self-contained and relatively easy to follow [4, 5]. In this work, we make change kernel performance significantly by modifying just a few key parameters. This work investigates the effect of changing weights of sibling threads (threads created in the same process) and evaluate this modification in terms of fairness.

C. Linux kernel scheduler

1) O(1) Scheduler

A version of Linux scheduler was developed with the release of kernel 2.6. This new scheduler is called the O(1) scheduler[4,5]. The name was chosen because the scheduler's algorithm required constant time to make a scheduling decision, regardless of the number of processes. The algorithm used by the O(1) scheduler relies on active and expired arrays of processes to achieve constant scheduling time. Each process is given a fixed time slice, after which it is preempted and moved to the expired array.

Once all the processes from the active array have exhausted their time slice and have been moved to the expired array, an array switch takes place. This switch makes the active array the new empty expired array, while the expired array becomes the active array. Each array in the run-queue represents 140 different priority levels: from 0 (highest) to 99 (lowest) use real-time policy. Priority levels from 100 (highest) to 139 (lowest) represent time-shared processes under the SCHED_NORMAL scheduling [18,19].

2) Completely Fair Scheduler

Completely Fair Scheduler (CFS) is introduced by Ingo Molnár to replace the O(1) scheduler [6,7,14,15]. This scheduler was designed to provide good interactive performance while maximizing overall CPU utilization [8]. It also strives to provide fairness in every process without sacrificing the interactivity performance [20]. 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. 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 [4].

Linux's CFS scheduler does not directly assign timeslices to processes. Instead, in a novel approach, CFS assigns processes a proportion of the processor. On Linux, therefore, the amount of processor time that a process receives is a function of the load of the system.

CFS will run each process for some amount of time, round-robin, selecting next the process that has run the least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. Instead of using the nice value to calculate a timeslice, CFS uses the nice value to *weight* the proportion of processor a process is to receive. Each process then runs for a "timeslice" proportional to its weight divided by the total weight of all runnable threads [5].

D. Problem statement

The Linux Completely Fair Scheduler (CFS) uses thread fair scheduling algorithm, this algorithm allocates CPU resources based on the number of

threads in the system not within the running processes. In the current scheduler (CFS), when a new process is created, it appears as a thread where both PID and TGID 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) but it inherits its TGID from its parent as shown in Fig. 1. Therefore scheduler does not distinguish between threads and processes and that way; the kernel can happily schedule threads independent of what process they belong to. Each forked thread is assigned a weight which determines the share of CPU bandwidth that thread will receive.

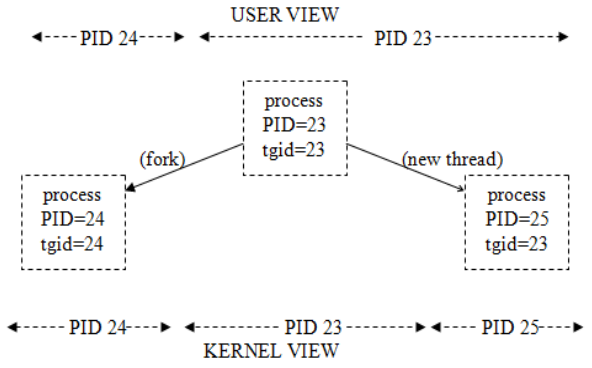

Fig. 1. Identifications of process and thread created from parent process.

The forked threads from the same process inherit their weights from their parent and consequently the same time slice will be assigned to each thread. Greedy users could take advantage by spawning more additional threads in order to obtain more CPU resources when there are other processes with lower number of running threads, this has a negative effect on scheduling criteria because these criteria are affected by the time assigned to the additional threads.

E. Research Contribution of This Paper

In this paper, we investigate the effectiveness of *thread weight readjustment scheduler (TWRS)* [30], a CPU scheduling algorithm for multitasking environment. The main contribution of this work is that we evaluate TWRS which is a proportional share CPU scheduling intended to reduce the greedy behaviour of multithreaded processes in order to achieve better fairness. In proportional share algorithm, every thread has a weight and thread receives a share of the available resources proportional to its weight [17].

In this work a modification is implemented to CFS. The modification is based on changing sibling threads' weights created from the same process and assigning a specific time slice to each of these sibling threads. Our criterion in this work is achieving fairness between running multithreaded processes. Fairness depends on the size of the time slice assigned to processes.

We have implemented our scheduler in the Linux kernel and experimentally demonstrated the improvement of our scheduler over the current scheduler, CFS, using benchmarks. Our experimental results show that TWRS scheduler is closer to the ideal case than CFS.

The rest of this paper is structured as follows: in section II, we discuss the related research. Section III discusses TWRS. The experimental setup and scheduling modes are given in section IV. Section V presents the experimental results.

II. RELATED RESEARCH

Chandra [3] presented a novel weight readjustment algorithm, surplus fair scheduler (SFS), to translate infeasible weight assignments to a feasible set of weights. Although the implementation of surplus fair scheduling was done in an old version 2.2.14 of the Linux kernel which is a predecessor of current scheduler CFS, this algorithm was the first one that has been explicitly designed for proportionate allocation of processor bandwidth in multiprocessor environments.

Chee [22] proposed an algorithm based on weight readjustment of the threads created in the same process. This algorithm, PFS, is proposed to reduce the unfair allocation of CPU resources in multi-threaded environment. Chee assumed that the optimal number of threads, best number to create in a program in order to have the best performance in a muti-processing environment, equals to the number of available cores. A limitation of this algorithm is that all processes

executing at the same nice value will receive the same amount of time slice.

A modification of PFS algorithm has been proposed to overcome this limitation by implementing TFPS [24]. TFPS shall give the greedy threaded program (e.g. program tries to dominate most CPU time) the same amount of CPU bandwidth as optimally threaded program, and both of their time slices are larger than the single-threaded program.

According to PFS, all processes executing at the same nice value will receive the same time slice regardless the number of threads, this is considered as a defect in this algorithm. And because each thread will receive the same amount of time slice, the system may suffer from overhead due to the number of context switches resulting from executing greedy program. According to PFS and TFPS, multi-threaded programs are not rewarded; this is because in PFS; the time slice of greedy program is the same as the amount of time slice assigned to the process which has the lowest number of threads, and in TFPS; the time slice of greedy program is the same as amount of time slice allotted to the optimally-threaded process.

The basic idea of TWRS is that the working environment and the method of readjusting the weights different from previous ones. As we will see in section III, our scheduler readjusts the thread's weight taking into consideration the number of all running threads in the current CPU and does not restrict the amount of time slice assigned to a process with others.

III. THREAD WEIGHT READJUSTMENT SCHEDULER

A. Overview

TWRS is a kernel-level thread scheduler to enhance performances of multi-threaded programs by focusing on weight readjustment 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. The share allocated to a thread is a ratio of its weight to the sum of weights of all active threads in the run queue. This is given by the equation:

$$share = \frac{se->load.weight}{cfs->load.weight}$$

Where se->load.weight 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) [21], and each nice value has its respective weight. cfs->load.weight is the total weight of all threads under CFS run queue. The time-slice that a thread should receive in a period of time is given by:

$$slice = \frac{se->load.weight}{cfs->load.weight} \times period$$

where period is the time quantum the scheduler tries to execute all threads, by default this is set to 20ms [2]. If number of runnable processes does not exceed (sched_latency_ns / sched_min_grnularity_ns), then period = sched_latency_ns, otherwise, period = number_of_running_tasks x sched_min_ granularity_ns . By default, sched_latency_ns = 20ms and sched_min _granularity_ns = 4ms [18].

In our consideration we will count the number of total threads in the CPU and the number of threads forked from the same process. Weights of the threads will be changed according to the next equation;

$$thrd->se.load.weight = new\_weight$$

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

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

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.

C. Illustrative Example

We show in Fig. 2 an example to clarify previous explanation. Fig. 2 shows the run queue of CFS, the circle in this figure represents the thread, the color of red or black in red-black tree is ignored. The threads with same pattern refer to sibling threads (e.g. threads created in the same process). We assume in this example two processes A and B, process A has five threads and process B has two threads. In this example all threads have the same nice value 0 and therefore the same weight. The numbers in the threads show the vruntime, and threads in the run queue are in an ascending order according to their vruntime.
Ideally, each thread should get 2.857ms = 20 * 1024 / (7 * 1024). But this is less than sched_min_granularity_ns, which decides the minimum time a thread will run. At the time of writing this paper, sched_min_granularity_ns is set to 4000000ns in Linux 2.6.24. So in this case of seven threads, the period becomes 28ms and each thread will run for 4ms before it is preempted. Users could take advantage by spawning many additional threads in order to obtain more CPU resources. In this paper, we evaluate TWRS which is proposed to distribute CPU time proportionally amongst threads according to their weights.

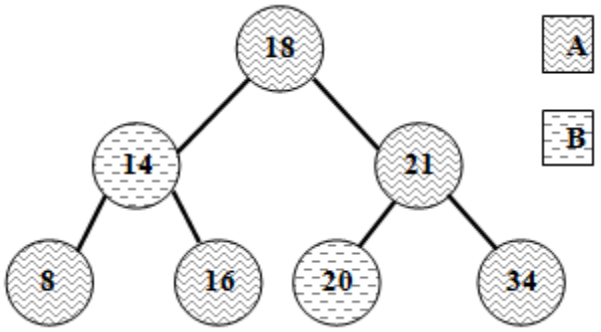

Fig. 2. Two multithreaded processes with different numbers of threads

IV. EXPERIMENTAL SETUP

A. Hardware and Software

TWRS can be easily integrated with an existing scheduler based on per-CPU run queues. To demonstrate its efficacy, we have 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)2 Duo CPU T7250 @ 2.00GHz
cpu MHz	800.000
cpu cores	2
Memory	2565424 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 versatility and performance of the modified kernel, we evaluated using 2 multi-threaded benchmarks on a multitasking 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 [23]. 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- prime 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.

TABLE II. S0 AND S1 SCENARIOS

S0		No. of threads in fixed program "threads"	No. of threads in varied program "threads"	S1		No. of threads in fixed program "threads"	No. of threads in varied program "cpu"
	S0.1	2	4 8 16 32 64		S1.1	2	4 8 16 32 64
	S0.2	4	8 16 32 64		S1.2	4	8 16 32 64
	S0.3	8	16 32 64		S1.3	8	16 32 64

- 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 inst`ance. 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 repeated 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 repeated this simulation with a different number in the fixed program (threads).

V. 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. From our experiments, we assume that the ideal case is when all running processes have the same number of threads. In such ideal case, the average CPU usage for all running processes is approximately equal to 100% divided by the number of running processes, for example, in each of our scenarios, S0 and S1, there are two running processes and therefore the average of CPU usage in the ideal case is 50%. We repeated this simulation many times for each sub-scenario. Our results show a significant improvement in terms of fairness. For each sub-scenario, the two programs were run concurrently 20 times and the average values were taken. Figures 3 and 4 show the results for S0 and S1 respectively, figures 5 and 6 show the percentage error between (CFS, ideal case) and (TWRS, ideal case) in S0 and S1.

CONCLUDING REMARKS

In this paper, we investigate the effectiveness of Thread Weight Readjustment Scheduler (TWRS) for multitasking operating systems from the view point of fairness. TWRS distributes CPU time proportionally for threads according to their weights and preallocates certain amount of CPU time to each thread of the multi-threaded processes. We focused in this work on fairness as it is an important scheduling criterion the scheduler should meet. 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 TWRS is promising to achieve better fairness over current scheduler, and the percentage error of our scheduler is less than the percentage error of current scheduler.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 25330084.

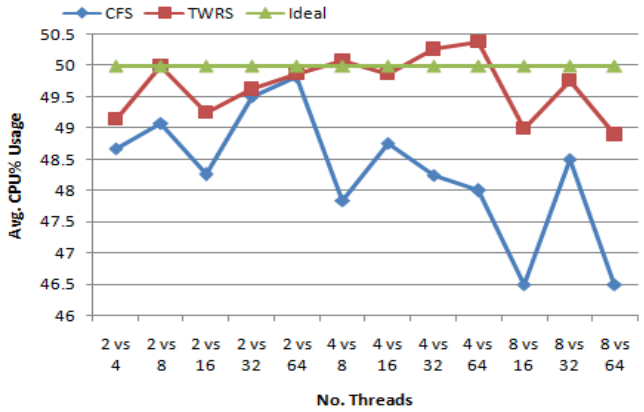

Fig. 3. Fairness comparison; fixed number of threads of "threads" program vs. N threads of "threads" program in each sub-scenario in S0.

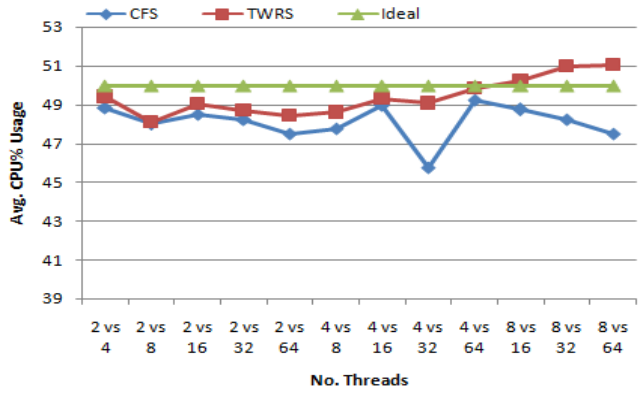

Fig. 4. Fairness comparison; fixed number of threads of "threads" program vs. N threads of "cpu" program in each sub-scenario in S1.

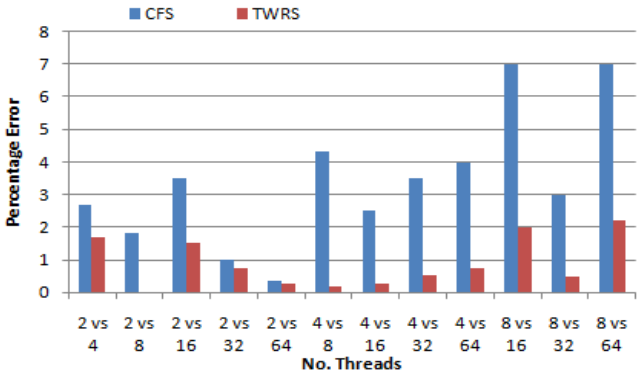

Fig. 5. Percentage error between (CFS, ideal case) and (TWRS, ideal case) in each sub-scenario in S0.

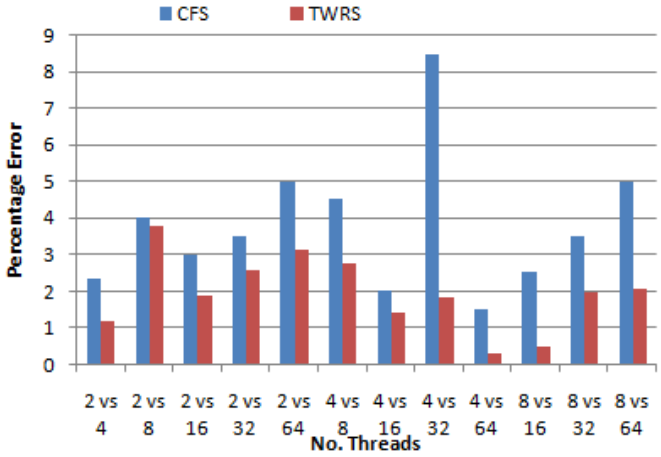

Fig. 6. Percentage error between (CFS, ideal case) and (TWRS, ideal case) in each sub-scenario in S1.

REFERENCES

1. Chulho Shin, Seong-Won Lee and Jean-Luc Gaudiot: Adaptive dynamic thread scheduling for simultaneous multithreaded architectures with a detector thread. Journal of Parallel and Distributed Computing, Volume 66, Issue 10, pp. 1304–1321, (October 2006).
2. Wong C.S., Tan I.K.T., Kumari R.D., Lam J.W., Fun W.: Fairness and interactive performance of O(1) and CFS Linux kernelschedulers. In: Information Technology, ITSim (2008).
3. 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).
4. Daniel P. Bovet and Marco Cesati: Understanding the Linux Kernel. 3rd edition, O'Reilly Press, ISBN 0-596-00565-2, (2005).
5. Love R.: Linux Kernel Development. 3nd edition, Noval Press, ISBN 0-672-32720-1, (2010).
6. Molnar, I.: Modular Scheduler Core and Completely Fair Scheduler [CFS], http://lwn.net/Articles/230501/
7. Eric Schulte Taylor Groves, Jeff Knockel: Bfs vs. cfs scheduler comparison, http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf, (December 2009).
8. 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).

9. Samih M. Mostafa and Shigeru Kusakabe: Achieving Better Fairness for Multithreaded Programs in Linux using Group Threads Scheduler. International Workshop on ICT at Beppu, Kamenoi Hotel, Beppu, Oita, Japan, 12th to 14th (December 2013).
10. 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).
11. 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).
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. 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).
15. Prajakta Pawar, S.S.Dhotre, 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).
16. Silberschatz A, Galvin PB, Gagne G.: Operating Systems Concepts. John Wiley and Sons. 9Ed, (2013).
17. Kevin Jeffay, F. Donelson Smith, Arun Moorthy, James Anderson: Proportional Share Scheduling of Operating System Services for Real-Time Applications. In: Proceedings of the 19th IEEE Real-Time Systems Symposium, pp. 480-491. Madrid, Spain (December 1998).
18. Jacek Kobus and Rafal Szklarski: "Completely Fair Scheduler and its tuning." http://www.fizyka.umk.pl/jkob/prace-mag/cfs-tuning.pdf, (2009).
19. Josh Aas.: Understanding the Linux 2.6.8.1 CPU scheduler. Silicon Graphics, Inc., (2005).
20. Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah,Johannes E. Gehrke, C. Greg Plaxton: A Proportional Share Resource Allocation Algorithm for Real-time and Time-shared Systems. In: Proc. 17th IEEE Real-Time Systems Symp., (December 1996).
21. http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/kernel/sched.c?v=2.6.25#L1183
22. 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).
23. http://sysbench.sourceforge.net/docs/
24. 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).
25. Dean M Tullsen, Susan J Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: ISCA96, pp. 191–202, (May 1996).
26. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy: Simultaneous multithreading: Maximizing on-chip parallelism. In: 22nd Annual International Symposium on Computer Architecture, pp. 392–403, (June 1995).
27. Jack L. Lo,Joel S. Emer,Henry M. Levy,Rebecca L. Stamm and Dean M. Tullsen,S. J. Eggers: Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. In: ACM Transactions on Computer Systems, (August 1997).
28. Mohan Rajagopalan Brian T. Lewis and Todd A. Anderson: Thread scheduling for multi-core platforms. In: HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems, (May 2007).
29. 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).
30. Samih M. Mostafa, and Shigeru Kusakabe: Towards Maximizing Throughput for Multithreaded Processes in Linux. International Journal of New Computer Architectures and their Applications (IJNCAA) 4(2): pp. 70-78, (2014).