Fairness and interactive performance of O(1) and CFS Linux kernel schedulers

Conference Paper · September 2008

DOI: 10.1109/ITSIM.2008.4631872 · Source: IEEE Xplore

5 authors, including:

Chee Siang Wong
Universiti Tunku Abdul Rahman
9 PUBLICATIONS **120** CITATIONS
SEE PROFILE

Ian KT Tan
Monash University (Malaysia)
38 PUBLICATIONS **226** CITATIONS
SEE PROFILE

Rosalind Deena Kumari
Multimedia University
6 PUBLICATIONS **106** CITATIONS
SEE PROFILE

Wey Fun
National University of Singapore
3 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers

C.S. Wong, I.K.T. Tan, R.D. Kumari, J.W. Lam
Multimedia University
{cswong,ian,rosalind,jwlam}@mmu.edu.my

W. Fun
Intel Penang Design Centre
funwey@intel.com

Abstract

The design of an Operating System (OS) scheduler is meant to allocate its resources appropriately to all applications. In this paper, we present the scheduling techniques used by two Linux schedulers: O(1) and Completely Fair Scheduler (CFS). CFS is the Linux kernel scheduler that replaces the O(1) scheduler in the 2.6.23 kernel. The design goals of CFS are to provide fair CPU resource allocation among executing tasks without sacrificing interactive performance. The ability to achieve good fairness in distributing CPU resource among tasks is important to prevent starvation. However, these design goals have never been scientifically evaluated despite the fact that there are many conventional operating system benchmarks that are geared towards measuring systems performance in terms of throughput. We therefore scientifically evaluate the design goals of CFS by empirical evaluation. We measure the fairness and interactivity performance by using fairness and interactivity benchmarks. To provide a meaningful representation of results, comparisons of O(1) and CFS kernel schedulers of the open source Linux OS are used. Our experience indicates the CFS does achieve its design goals.

1. Introduction

An operating system is software that handles computer hardware and acts as an intermediary between the user of a computer and the computer hardware [1]. Central Processor Unit (CPU) scheduler of an OS is responsible to distribute the CPU bandwidth among tasks. The algorithm of the scheduler will dictate the order and performance of executing tasks.

The Linux kernel has been one of the areas that are actively being improved and developed in the open source community. Due to emerging number of Linux users in desktop environments, the Linux kernel scheduler has been improved to boost its interactivity performance. Unlike server environments, most applications in desktop environment are interactive and require low latency to enhance user experience. Past research [2] has shown that the most important performance criterion for interactive applications is responsiveness instead of throughput.

Fairness is another important design goal of CPU schedulers [3]. More fairness in terms of CPU bandwidth allocation among tasks means each tasks will be proportionally advancing forward when executed. The opposite of fairness is starvation, which means certain tasks are left waiting for CPU resource for a period of time. In other words, a scheduler that provides fairness among tasks prevents starvation.

The two most recent Linux kernel schedulers: 2.6 O(1) and 2.6.23 Completely Fair Scheduler (CFS) share some characteristic in terms of fairness and interactivity performance. The design goals of the Linux kernel scheduler version 2.6 O(1) is to achieve fairness and boost interactivity performance [4]. In the kernel version 2.6.23, a totally new scheduler is introduced to replace O(1), called CFS [5]. The design goals of CFS is quite similar to O(1), but with the criterion that the fairness accuracy is improved without significantly decreasing the interactive performance [6]. However, these design goals of CFS have never been scientifically evaluated despite that Linux kernel development is an open source project. In [9], the fairness accuracy of the time sharing policy in CFS was evaluated. Thus, in this paper, we further evaluate the fairness and the interactivity performance of the time-sharing policy in both schedulers.

The following section of this paper gives an overview of O(1) and CFS Linux kernel schedulers by explaining how the scheduling algorithm for time-sharing tasks works, especially how the fairness and interactivity performance are attained. Section 3 describes the tasks fairness measurement micro-

978-1-4244-2328-6/08/$25.00

benchmark. Section 4 describes the interactivity measurement micro-benchmark and results. The result and discussion of both micro-benchmarks are presented respectively. Finally, we conclude the paper in Section 5.

2. Linux Kernel Schedulers

2.1. O(1) scheduler

With the intention to resolve the inadequacy of the O(n) scheduler in the 2.4 kernel, especially in handling large number of threads when running Java virtual machines, Ingo Molnar [7] introduced the O(1) scheduler in the Linux 2.6.0 kernel. This scheduler is used in every release in 2.6.x until it is replaced by CFS in the 2.6.23 kernel.

In the O(1) scheduler, a run-queue is defined for each processor in the system [4]. Each run-queue keeps track of all runnable tasks assigned for its associated CPU using 2 arrays: an active array and an expired array. The scheduler selects a runnable task with the higher dynamic priority from the active array to execute on the CPU for a pre-determined time slice. Tasks within the same priority are scheduled round robin. Upon yielding the CPU, the next time slice for the task is calculated and it is put into the expired run-queue. When all the tasks in the active array have used up their time slices (i.e. the active array is empty), the pointers to the active array and expired array are switched. The active array becomes expired array and vice versa.

The O(1) scheduler can be classified under time-dependent priority scheduler, such as [14] and [15]. In a time-dependent priority scheduler, the time-slice allocation of a task is dependent to the priority of the task.

To enforce prioritization of tasks in the system, each array in the run-queue is 2 dimensional, representing 140 different priority levels: priority levels from 0 (highest real-time priority) to 99 (lowest real-time priority) use real-time policy which is not the focus of this paper. The priority levels from 100 (highest priority) to 139 (lowest priority) represent the conventional, time-shared processes under the SCHED_NORMAL scheduling class.

The priority levels from 100 to 139 are called static priorities. Nice value is the value that influences the priority of a process. Each of these static priorities corresponds to a nice value ranging from -20 (highest static priority, not nice to other processes) to 19 (lowest static priority, nicest to other processes). Each task is allocated a base time slice based on its static priority/nice value using the equation:

$$\textit{Base time slice (in milliseconds)} = \begin{cases} \textit{(140-static priority) x 20} & \textit{if static priority} < 120 \\ \textit{(140-static priority) x 5} & \textit{if static priority} \geq 120 \end{cases}$$

Thus, a process with higher static priority will obtain a longer base CPU time slice. By implementing that, tasks with the same static priority shall receive same CPU time slice, which is attributed as fairness.

In addition to static priority, a dynamic priority is calculated for each task as well. Dynamic priority is applied to dynamically modify the priority of each task by taking into consideration the average sleep time of a task. Average sleep time of tasks is tracked because the scheduler tries to identify interactive tasks with the assumption that interactive tasks sleep frequently, especially when the tasks are waiting for input from users. The average sleep time is estimated by tracking the average number of nanoseconds that a process spent sleeping. Starting at 0ms, for every 100ms increment in the average sleep time, the bonus value grows from -5 to 5. Based on that, a penalty of -5 to -1 or a bonus of 1 to 5 is given to the process. Thus, dynamic priority is obtained by the formula:

$$\textit{dynamic priority} = \textit{max(100,min(static priority-bonus,139))}$$

Furthermore, to ensure user responsiveness, interactive applications are favoured by the scheduler over batch processes. This feature is another effort to boost interactive performance. The scheduler attempts to keep interactive processes in the active runqueue even if their time slices have been exhausted. To identify interactive processes, a heuristic using interactive delta is defined:

$$\textit{interactive delta} = \textit{(static priority / 4) - 28}$$

A process is considered interactive if

$$\textit{bonus} \geq \textit{interactive delta}$$

An interactive process will be placed into the expired array if the eldest expired process has already waited for a long time, or if there is a higher priority process in the expired array. Thus, there will be no starvation of non-interactive processes.

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

2.2. Completely Fair Scheduler

Completely Fair Scheduler (CFS) is introduced by Ingo Molnar to replace the O(1) scheduler [5]. Components that present in O(1) scheduler such as array of run queues, interactive processes identification, and priority-based time slices concepts are removed to improve efficiency. Other improvements such as group and user fair scheduling and modular scheduler framework are also introduced [16]. However, those improvements are not highlighted in this paper because the focus of this paper is to investigate the fairness and interactivity performance of schedulers.

This scheduler was designed to provide good interactive performance while maximizing overall CPU utilization. Even during high load, the interactivity shall be maintained. It also strives to provide fairness in every task without sacrificing the interactivity performance [6]. This is done by giving a fair amount of CPU time to each task proportional to its priority. This method is also called proportional share algorithm, where a share is allocated for each task, which is associated with the task's weight [13].

An ideal multitasking CPU is what CFS tries to precisely model, where the CPU divides the CPU time quantum equally to each tasks so that they run at precisely equal rate in parallel. For example, if there are more than one tasks executing on an ideal uni-processor with the same level of nice values, the ideal multitasking CPU should execute them in parallel. Due to simultaneous tasks execution, all tasks are expected to finish executing in the same time. However, in reality, a uni-processor system can only execute a single task at any instance of time. When one task is occupying the CPU, other waiting tasks are lagging behind the currently executing task. This is an undesired scenario because the current occupying CPU task gets unfair amount of CPU time. CFS splits up the CPU time between runnable tasks as close as 'ideal case' as possible in order to minimize the lags. This is done by dividing the time-slice in high resolution using proportional share algorithm. The CFS still maintains the same 140 priority levels that are found in O(1) scheduler.

For time-sharing tasks, each task is assigned a weight which determines the share of CPU bandwidth that tasks will receive. The share allocated to a task is a ratio of its weight to the sum of weight of all active tasks in the runqueue. This is given by the equation:

$$share = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight}$$

where `se->load.weight` is the weight of the schedule-able entity. It is mapped from its nice value in `prio_to_weight[ ]` and each nice value has its respective weight. `cfs_rq->load.weight` is the total weight of all entities under that CFS runqueue. A schedule-able entity can be a container, user, group (for user/group fair scheduling), or tasks. In this paper, only tasks scheduling algorithm is concerned. The algorithm is similar to other proportional share schedulers like Earliest Eligible Virtual Deadline First (EEVDF) scheduler [13]. The time-slice that a task should receive in a `period` of time is given by:

$$slice = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \times period$$

where `period` is the time slice the scheduler tries to execute all tasks. Unlike the O(1) scheduler, the time slice receives by each task is not a constant and is dependent to `period`, which has a minimum value of 20ms. When the number of tasks in the runqueue is increased, the `period` will be stretched so that the minimum preemption granularity is 4ms. This is to prevent excessive scheduling when the number of tasks is much greater than the number of CPU in the system.

CFS uses virtual runtime to track the progress of each entity. Virtual runtime is the weighted time slice (or virtual time in [13]) given to every schedule-able entity. It is expressed using the equation:

$$vruntime += \frac{delta\_exec}{se \rightarrow load.weight} \times NICE\_0\_LOAD$$

where `delta_exec` is the amount of execution time of that task (`delta_exec` equals to `slice` if there is no preemption from higher priority task). `NICE_0_LOAD` is the unity value of the weight.

CFS uses a single time-based red-black tree for each CPU as its time-based data structure to track all the runnable tasks. Red-black tree is a self-balancing binary tree where nodes without children are called leaves [8]. All runnable tasks are sorted in the red-black tree by the value returned by function `entity_key` which calculates the difference of `se->vruntime` and `cfs->min_vruntime`. `cfs->min_vruntime` is the minimum value of all tasks' vruntime in the system, which also denotes the task that has received the minimum amount of time slice. The leaf at the leftmost inside red-black tree has the smallest value of `entity_key` and the task that is most entitled to run at any given time.

For example, there are three tasks: A, B, and C executing in a system at nice level 0. The

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

corresponding weights for nice 0 are 1024 for all three tasks. So, if `period` is 20ms, tasks A, B, and C shall each receives 6.67ms of ideal runtime respectively. Assume this is an ideal case and all three tasks enter the system at the same time with no higher priority tasks preempting them; i.e. the `delta_exec` equals to the `ideal_runtime`. Thus, the `vruntime` of task A, B, and C is 6.67 ms/unit weight. The `vruntime` for all task are expected to be same for each round of `period` of time (minimum 20ms). Practically, tasks within the same runqueue never start executing at the same time instance. So, one task may leads another, and thus be placed on the leftmost node of the red-black tree. That task will occupy the CPU bandwidth at that instance and its `vruntime` value will be increased as it consumes time-slice. This value will be increased until when it is larger than `vruntime` value of any other tasks. That task will be pre-empted and other task with the lowest `vruntime` value will be the left-most node and occupy the CPU bandwidth.

For interactivity optimization, CFS tunes the `vruntime` value that sleep for a period of time. When a task wakes up from sleeping and been inserted into the red-black tree, its `vruntime` value will be re-adjusted. For a task that sleeps within `slice` period of time, the `vruntime` value will be updated as in the equation below:

$$vruntime -= \frac{sysctl\_sched\_latency}{cfs\_rq->load} \times NICE\_0\_LOAD$$

where `sysctl_sched_latency` is equal to `period`. This readjustment of `vruntime` value has an impact to tasks that sleep frequently. For example, Task A sleeps and runs for 2ms in an alternating fashion. When Task A is executed concurrently with a full-load Task B that never sleep, the ratio of the CPU bandwidth received by Task A to Task B is 1:1 instead of 1:2. This is because the `vruntime` readjustment algorithm will assume Task A never goes to sleep at all. By implementing this algorithm, the fairness can be maintained without sacrificing interactivity. In contrast with CFS, O(1) scheduler favours interactive tasks and penalize non-interactive tasks. This may lead to a fairness issue.

3. Fairness Test and Result

Two micro-benchmarks are used to evaluate both schedulers. Fairness test [9] will be used to quantitatively measure how accurate are the schedulers distributing the CPU bandwidth in a fair manner. In the meanwhile, the modified Interbench [10] is used to evaluate the interactivity performance of both schedulers.

3.1. Fairness Test

Fairness test measures the scheduler's accuracy in distributing CPU bandwidth evenly among multiple processes. Multiple computational-intensive processes are created and each of them performs a similar integral calculation of pi value. The single-threaded pi calculation program uses discrete integration [11].

This micro-benchmark application is consisting of parent process (which is the main application itself) and *n* number of children processes. First, the parent process forks *n* number of children. Each child then registers signal and wait for signal from parent process so that they will start executing at the same time. After the parent process has ensured that each child is forked, it will signal the children to start executing. Each children process will start to perform the timing mechanism concurrently using the nanosecond-accurate `clock_gettime()` interface followed by the calculation of pi value respectively. Each child that finishes executing its respective calculation will stop the timing mechanism and the time duration of execution is recorded.

3.2. Simulation Environment

To measure fairness of both schedulers, a set of 8, 16, 24, and 32 children of pi processes are executed. Each set of processes will be executed concurrently and repeated for 100 times. The average values of results are then represented in line graphs showing the number of children against the time consumed to finish the calculation. The timing is grouped to the nearest 0.1 second. The spread area of data in the smoothed bell-shape line graph will indicate how high the standard deviation of the time consumption among tasks is. The more the time recorded concentrates on the average value, the lesser standard deviation it has.

The differential value for pi calculation is made very small to ensure that the approximation program takes at least a few seconds to complete. In this test, the differential value is set to ten millionths (1/100000000). The test was performed using a Intel Xeon X3210 2.13 GHz quad-core processor with 2 GB of main memory running on Linux 2.6.22 kernel (for O(1)) and 2.6.24.2 kernel (for CFS). Only one core is enabled throughout the test so that only one run queue is used.

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

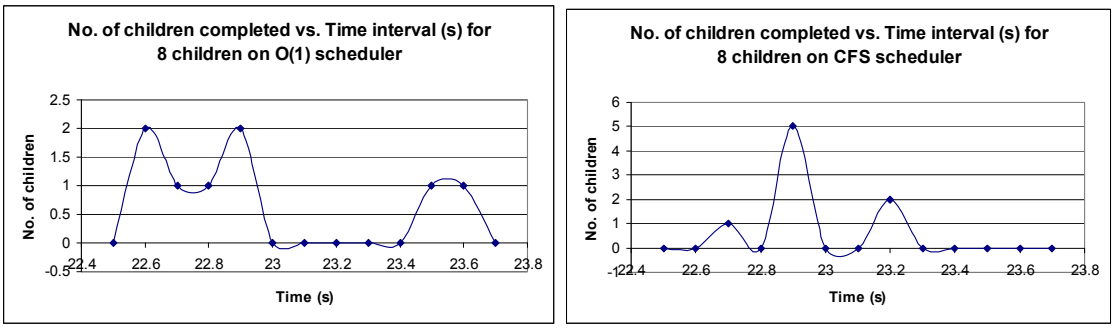

Figure 1 (a): Results of fairness measurement for 8 number of children on O(1) and CFS schedulers.

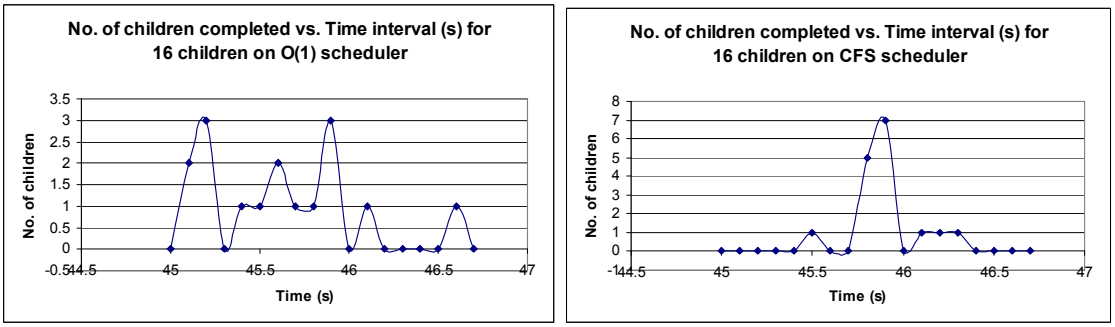

Figure 1 (b): Results of fairness measurement for 16 number of children on O(1) and CFS schedulers.

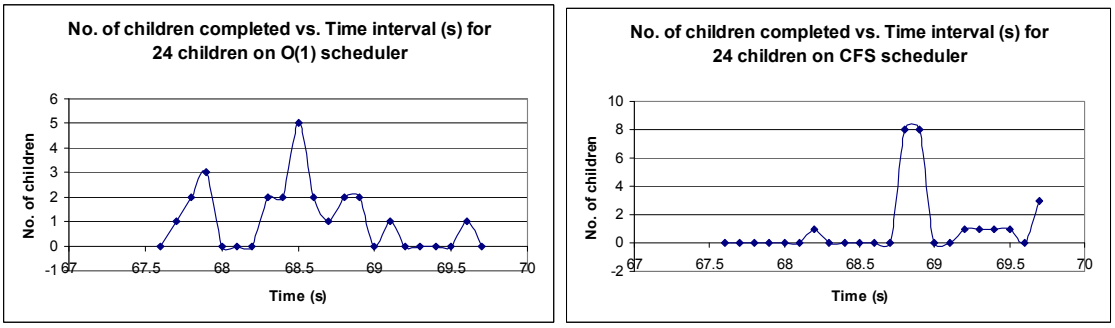

Figure 1 (c): Results of fairness measurement for 24 number of children on O(1) and CFS schedulers.

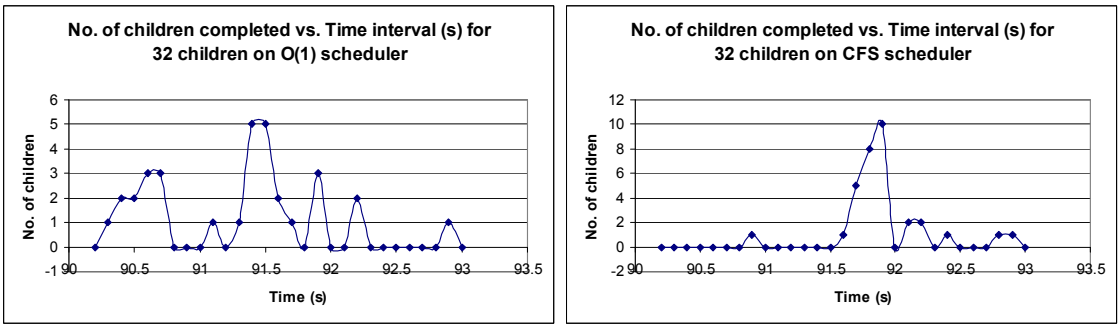

Figure 1 (d): Results of fairness measurement for 32 number of children on O(1) and CFS schedulers.

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

3.3. Results

Figure 1 (a-h) shows the result obtained from the test of a set of 8, 16, 24, and 32 children of pi processes on both schedulers. The smoothed bell-shape graphs for the O(1) scheduler shows that the standard deviation of the time consumed for each children to complete their pi processes are higher than CFS. In the O(1) scheduler, some children finish executing faster than others. This explains the wide spread of data across the graph. In contrast, the smoothed bell-shape graphs for CFS scheduler are significantly narrower and concentrated on its average value. This shows the CFS scheduler distributes its CPU time among each tasks in a fairer manner compared to O(1) scheduler.

There are many reasons behind the better task fairness achieved by CFS. One of the reasons is CFS tries to simulate an ideal multitasking CPU by dividing the time-slice into very fine granularity. Another factor that contributes in having fair CPU time in each task is the usage of nanoseconds accurate accounting. The time that should be entitled to each task to simulate an ideal multitasking CPU is calculated in nanoseconds precision. This accounting improves fairness in distributing CPU time among tasks.

4. Interactivity Test and Result

4.1. Interactivity Test

The interactivity of a program is defined as the scheduling latency or jitter that presents in tasks under different work load condition. This amount of latency influences the interactivity between user and program where high latency represents noticeable jerk. A customized Interbench is used to benchmark the interactivity of both CPU schedulers.

Interbench simulates various interactive tasks with the presence of background tasks called load. The benchmark measures the scheduling latency between the time the interactive task requesting the CPU resource and the time it actually receives it. Among the background loads available in Interbench are simulations of video load, full load, writing to disk load, reading from disk load, compiling kernel load and heavy memory swapping load. Interbench also allows load customization. In this paper, the focus is to evaluate the interactivity performance of desktop scheduling algorithms. Thus, only CPU bound loads are used because this paper is focus on CPU schedulers instead of Input/Output (I/O) schedulers.

This micro-benchmark is customized into three tests which I/O disks reading and writing loads are excluded. All the tests are using X-window simulation as the interactive task (front load) with the presence of background load. The background loads used are audio, video and full load which represent low, medium and high load respectively.

Audio, which is simulated to use up 5% of CPU time, is used as low load test. This load is waked up every 50ms using 5% of CPU load. Meanwhile, video is simulated to consume 40% of CPU time for medium load test. This load is waked up every 1/60 seconds to simulate a 60 frame per second video. Full load test is simulated to consume 100% of CPU resource. The X-window interactive task is simulated in such a way that it has variable amount of load where the CPU resource required ranged from 0% to 100%. It simulates an idle Graphical User Interface (GUI) window that is dragged across the screen by a user. Each test is executed for duration of 60 seconds.

4.2. Simulation Environment

The program first benchmarks the system to measure how many meaningless loops the system can run in one millisecond. This is done so that fixed percentage of CPU usage on the system can be reproduced. It saves this to a file and then uses this for all subsequent runs to keep the emulation of CPU usage constant.

For each samples, latency between the starting time of invoking the X-window interactive task and the actual time it starts executing is recorded. This latency represents the time taken by scheduler to allocate the CPU resource to the interactive task. The maximum latency for each test is also recorded because maximum latency represents jitter that occurs when interactive task is running. The latency value is accumulated after a sample has finished executing. After all the samples have finished executing, the average latency is calculated by dividing the total latency accumulated by the number of samples it completes throughout the duration of a test. Each test is repeated for 100 times and the average values are taken. This is shown in Table 1.

Table 1: X-window simulation with the presence of different types of background loads.

Test No.	Interactivity tasks (load)	Background tasks (load)
1	X-window (0 to 100%)	Audio (5%)
2	X-window	Video (40%)

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

	(0 to 100%)	
3	X-window (0 to 100%)	Full load (100%)

The test was performed using the same test bed as mentioned in Section3.2.

4.3. Results

The results of the interactivity test are shown in Figure 2 (a-b). Figure 2(a) shows that the X-window interactive task has lower average latency in CFS than O(1) scheduler when low load presents. For both medium and high load, the interactive task has lower latencies when running on O(1). Although CFS has higher average latency, the latency is well below human's reaction time to visual stimulus, which is 180 to 200ms [12]. The user of the computer is unlikely to notice the insignificant difference of average latency in CFS and O(1) schedulers because visual stimulus takes 20 to 40ms to reach the brain.

CFS scheduler has lower maximum latency recorded in all three types of background loads. Again, those differences in jitter are not significant to be noticed and reacted by user.

The interactive tests show that although the interactive task has slightly higher average latency time when running on CFS compared to O(1) scheduler, it is still acceptable because it is insignificant and not noticeable by human.

In CFS, the algorithm used is proportional share algorithm where the lag time of any active task is bounded by `period`. This is because every runnable task will be attended in `period` time. In contrast, the O(1) scheduler uses time-dependent priority algorithm where each task is given the same base time slice for the same priority. However, the lag time is bounded by the number of active tasks.

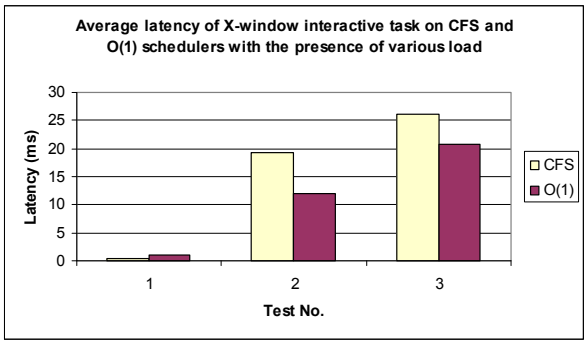

Figure 2 (a): Bar chart of average latency of X-window interactive task on CFS and O(1) schedulers with present of various background loads.

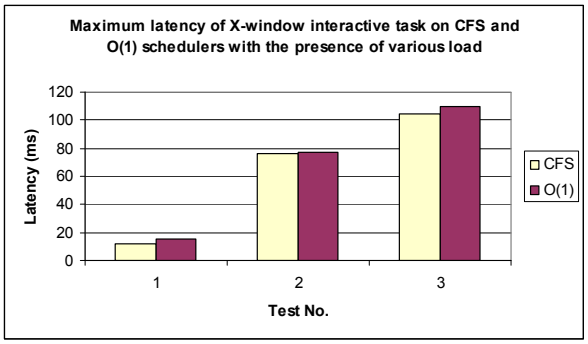

Figure 2 (b): Bar chart of maximum latency of X-window interactive task on CFS and O(1) schedulers with present of various background loads.

5. Conclusion

Results from fairness and interactivity tests show that the CFS has the advantage of being fairer in CPU bandwidth distribution without compromising interactivity performance significantly. This can be explained by implementation of fairly divided time slice given to every task and the nanoseconds accurate accounting. It is also more efficient than O(1) because CFS does not have complex algorithm to identify interactive tasks. Results from empirical evaluation of both schedulers show that CFS does achieve its main design goal.

6. Acknowledgements

We wish to acknowledge Linux kernel developers: Ingo Molnar and Peter Zijlstra for consultation and assistance on CFS. Also, we would like to thank Intel Penang Design Centre for providing hardware and research funding.

978-1-4244-2328-6/08/$25.00 © 2008 IEEE

7. References

[1] A. Silberschatz, P.B. Gailvin, G. Gagne, “Operating System Concepts,” 7th Edition, John Wiley & Sons Inc., 2005.

[2] B. Schneiderman, "Designing the User Interface: Strategies for Effective Human-Computer Interaction," 3rd Edition, Addison Wesley Longman, 1998.

[3] D.P. Bovet, et. al., "Understanding the Linux Kernel,” 3rd Edition, O'Reilly Press, ISBN 0-596-00565-2, 2004.

[4] R. Love, "Linux Kernel Developement," 2nd Edition, Noval Press, ISBN 0-672-32720-1, 2005.

[5] I. Molnar, "Modular Scheduler Core and Completely Fair Scheduler [CFS]," http://lwn.net/Articles/230501, Last accessed: 1st April, 2008.

[6] I. Molnar, “CFS Scheduler,” http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt, Last accessed: 1st April, 2008.

[7] J. Andrew, "Interview: Ingo Molnar," http://kerneltrap.org/?q=node/517, Last accessed: 1st April, 2008.

[8] L. Turbak, “Red-Black Trees,” http://cs.wellesley.edu/~cs231/spring01/ps4.pdf, Wellesley College, Last accessed: 1st April, 2008.

[9] I.K.T. Tan, C.S. Wong, J.W. Lam, and R.D. Kumari, "Measuring Operating Systems' Tasks Fairness for CPU Resource Scheduling," in *IASTED International Conference on Advances Computer Science and Technology*, Langkawi, Malaysia, April, 2008.

[10] C. Kolivas, "The homepage of Interbench, The Linux Interactivity Benchmark," http://members.optusnet.com.au/ckolivas/interbench/, Last accessed: 2nd April, 2008.

[11] D. Turner, "Calculating Pi in Parallel," Introduction to Parallel Computing and Cluster Computers, Ames Laboratory, http://cs.wellesley.edu/~cs231/spring01/ps4.pdf, Last accessed: 2nd April, 2008.

[12] R.J. Kosinski, “A Literature Review of Reaction Time,” http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm#Number%20valid, Last accessed: 10th April, 2008.

[13] I. Stoica et. al., “A Proportional Share Resource Allocation Algorithm for Real-time and Time-shared Systems,” in *Proc. 17th IEEE Real-Time Systems Symp.*, Dec. 1996.

[14] J.L. Hellerstein, “Achieving Service Rate Objectives with Decay Usage Scheduling,” in *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, pp. 813-825, August 1993.

[15] L.L. Fong, and M.S. Squillante, “Time-Function Scheduling: A General Approach for Controllable Resource Management,” Working Draft, IBM T.J. Watson Research Center, New York, March 1995.

[16] C.S. Wong, I.K.T. Tan, R.D. Kumari, W. Fun, "Towards Achieving Fairness in the Linux Scheduler", in *ACM SIGOPS Operating Systems Review: Research and Developments in the Linux Kernel,* vol. 42, issue 5, July 2008.

978-1-4244-2328-6/08/$25.00 © 2008 IEEE