2014 Fifth International Conference on Intelligent Systems, Modelling and Simulation

On the Fairness of Linux O(1) Scheduler

Jyothish Jose, Oravanpadath Sujisha
Exalture Software Labs Pvt Ltd, Cochin
Kerala, India - 682037
{*jyothish6190, sujishavpz*}*@gmail.com*

Malayamparambath Gilesh, Thayyil Bindima
Government Engineering College Wayanad
Kerala, India - 670644
{*gileshmp, bindima*}*@gecwyd.ac.in*

***Abstract*—The scheduling algorithm of Linux operating systems has to fulfill several conflicting objectives: fast process response time, higher throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low and high priority processes etc. The set of rules used to determine when and how to select a new process to run is called scheduling policy. Current Linux kernel uses Completely Fair Scheduler(CFS) which does not meet the complete requirements of a good scheduler, since its complexity is O(log n) and its scheduling policy can cause starvation of optimally threaded process in the presence of greedy threaded process. Previous scheduler was O(1) scheduler, which has a complexity of O(1). The O(1) scheduler had limitations like poor interactive performance and low throughput for background jobs and hence was replaced by CFS. In this paper we illustrate the reasons for poor interactive performance and low throughput for background jobs and also provide a simple solution to the above problems. Finally we explain an enhancement to the solution for better performance of the modified scheduler.**

***Keywords*-Unix OS, Process Scheduling, Threads, O(1) scheduler, CFS, Starvation.**

I. Introduction

The process scheduler is an important part of the kernel of an operating system. A scheduler must *fairly* allot the processor time to all the processes. Numerous processes, scalability concerns, trade-offs between latency and throughput, and demands from various workloads makes it difficult to find a single all-in-one scheduler algorithm. The CFS process scheduler of the Linux kernel comes very close to satifying all the demands and providing an optimal solution for all cases with perfect scalability and throughput.

The 2.6 Linux kernel introduced a new scheduler that is commonly referred to as the O(1) scheduler. Which means that the scheduler can perform the scheduling of a task in constant time. We had the 2.6.22 version of Linux under consideration. The O(1) scheduler used in the kernel faced several challenges. One of the major issues is the starvation of a CPU-bound process when large number of interactive processes are present in the run-queue. Consequently later versions of Linux replaced the O(1) algorithm by CFS (Completely Fair Scheduler) algorithm which has a time complexity O(log n).

Linux processes are preemptive. If a process enters the running state the kernel checks whether its dynamic priority is greater than the priority of the currently running process. Thus the CPU-bound processes get pre-empted in the presence of I/O-bound processes which have higher priority than CPU-bound processes. Through our work, we could solve the problem of starvation in Linux 2.6.22 kernel with the O(1) scheduler and fine tune the scheduler to improve the performance, maintaining the time complexity at O(1).

Rest of this paper is arranged as follows. In the first we discuss the existing O(1) scheduler. In the next section we discuss the starvation problem and a simple solution. Third section describes the implementation of a modified kernel followed by a section depicting the results.

II. O(1) Scheduler

The 2.6 Linux kernel introduced the new scheduler viz. 0(1) scheduler[1][4]. The scheduler could perform the scheduling of a task in constant time. Scheduling in constant time reduces the total system time required for executing multiple processes.

A. Schedule Policy

The schdule policy determines which process will run next on the CPU. Linux scheduling is based on the time-sharing technique, that is several processes run in *time multiplexing* because the processor time is divided into *slices*, each of which run the most eligible process in the queue. Timesharing relies on timer interrupts and is thus transparent to processes. The scheduling policy is also based on ranking processes according to their priority. The scheduling algorithm fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on.

B. Timeslice

The time slice is the numeric value that represents how long a task can run until it is preempted. Too short a time slice causes significant amounts of processor time being wasted on the overhead of process switching because the number of context switches increases and thus percentage of switching time dominates.

2166-0662/14 $31.00 © 2014 IEEE
DOI 10.1109/ISMS.2014.120

C. Priority

A common type of scheduling algorithm is priority-based scheduling. Processes with a higher priority run before those with a lower priority, whereas processes with the same priority are scheduled in a first-come-first-serve, preempting fashion. On some systems including Linux, processes with a higher priority receive a longer time slice as well. Thus a runnable process with time slice remaining and the highest priority runs always. Also, the Linux kernel implements two separate priority ranges.

D. Data Structures Used

The O(1) scheduler was designed to accomplish specific goals[1]:

- Implement fully O(1) scheduling. Every algorithm in the new scheduler completes in constant-time, regardless of the number of running processes.
- Implement perfect SMP scalability. Each processor has its own locking and individual runqueue.
- Implement improved SMP affinity. Attempt to group tasks to a specific CPU and continue to run them there.
- Provide good interactive performance. Even during considerable system load, the system should react and schedule interactive tasks immediately.
- Provide fairness. No process should find itself starved or get high amount of timeslice.
- Optimize for the common case of only one or two runnable processes, yet scale well to multiple processors, each with many processes

The data structures are designed to assist the algorithm to accomplish the goals.

1) Runqueues: The basic data structure in the scheduler is the runqueue defined as *struct runqueue*. The runqueue is the list of runnable processes on a given processor; there is one runqueue per processor. Each runnable process is on exactly one runqueue. The runqueue additionally contains per-processor scheduling information. Consequently, the runqueue is the primary scheduling data structure for each processor.

2) Priority arrays: Each runqueue contains two priority arrays, the active and the expired array defined in *kernel/sched.c* as *struct prio_array*. Priority arrays are the data structures that provide O(1) scheduling. Each priority array contains one queue of runnable processors per priority level. Each queue contains list of the runnable processes at each priority level. The priority arrays also contain a priority bitmap used to discover the highest-priority runnable task.

3) Process descriptor: Each process descriptor includes several fields related to scheduling:

need_resched: A flag checked by ret_from_sys_call( ) to decide whether to invoke the schedule( ) function.

rt_priority: The static priority of a real-time process; valid priorities range between 1 and 99. The static priority of a conventional process must be set to 0.

counter: The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process.

cpus_runnable: A bit mask specifying the CPU that is executing the process, if any. If the process is not executed by any CPU, all bits of the field are set to 1. Otherwise, all bits of the field are set to 0, except the bit associated with the executing CPU, which is set to 1.

nice: Determines the length of the process time quantum when a new epoch begins. This field contains values ranging between -20 and +19; negative values correspond to *high priority* processes, positive ones to *low priority* processes.

E. Calculating Priority and Timeslice

1) Static task prioritization and the nice() system call: All tasks have a static priority, often called a *nice value* ranging from -20 to +19 as explained above. By default, tasks start with a static priority of 0, but that priority can be changed via the nice() system call. Apart from its initial value and modifications via the nice() system call, the scheduler never changes a task's static priority stored in *static_prio* variable. Static priority is the mechanism through which users can modify task's priority, and the scheduler will respect the user's input.

2) Dynamic task prioritization[3]: The Linux O(1) scheduler rewards I/O-bound tasks and punishes CPU-bound tasks by adding or subtracting from a task's static priority. The adjusted priority is called a task's *dynamic priority*, and is accessible via the task's priority variable. Priority of an interactive job gets boosted, whereas a CPU-bound job is penalised. In the Linux 2.6 scheduler, the maximum priority bonus is 5 and the maximum priority penalty is 5. Since the scheduler uses bonuses and penalties, adjustments to a task's static priority are respected. A CPU-bound job with a nice value of -2 might have a dynamic priority of 0, the same as a task that is neither a CPU nor an I/O job. If a user changes either's static priority, a relative adjustment will be made between the two tasks.

3) I/O-bound vs. CPU-bound heuristics: Dynamic priority bonuses and penalties are based on interactivity heuristics. This heuristic is implemented by keeping track of how long a task goes to sleep. In the Linux 2.6 scheduler, when a task is woken up from sleep, total sleep time is added to its *sleep_avg* variable. When a task gives up the CPU, voluntarily or involuntarily, the time the current task spent running is subtracted from its sleep_avg. The higher a task's sleep_avg, so is its dynamic priority. This heuristic is quite accurate since it keeps track of both time spent in sleeping as well as time spent in running. Since it is possible for a task to sleep quite a while and still use up its timeslice, tasks that sleep for a long time and then hog a CPU must be prevented from getting a huge interactivity bonus. The Linux

2.6 scheduler's interactivity heuristics prevent this because a long running time will offset the long sleep time.

4) The effective_prio() function: The *effective_prio()* function calculates a task's dynamic priority. It is called by *recalc_task_prio()*, the thread and process wakeup calls, and *scheduler_tick()*. In all cases, it is called after a task's sleep_avg has been modified, since sleep_avg is the primary heuristic for a task's dynamic priority. The first thing effective_prio does is return a task's current priority if it is a real-time task. The function does not give bonuses or penalties to real-time tasks. The next two lines are key:

```
bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
prio = p->static_prio - bonus;
```

CURRENT_BONUS is defined as:

```
#define CURRENT_BONUS(p) \
NS_TO_JIFFIES((p)->sleep_avg) * \
              MAX_BONUS / MAX_SLEEP_AVG);
```

Essentially, CURRENT_BONUS maps a task's sleep average onto the range 0 to MAX_BONUS, which is 0-10. If a task has a high sleep_avg, the value returned by CURRENT_BONUS will be high, and vice-versa. Since MAX_BONUS is twice as large as a task's priority is allowed to rise or fall, it is divided by two and that value is subtracted from CURRENT_BONUS(p). If a task has a high sleep_avg and CURRENT_BONUS(p) returns 10, then 5 is subtracted from its static priority, which is the maximum bonus that a task can get. If a task had a sleep_avg of 0 and its CURRENT_BONUS(p) value is 0, the bonus value would get set to -5 and the task's static priority would get -5 subtracted from it, which is the same as adding 5. Adding five is the maximum penalty a task's priority can get.

Once a new dynamic priority has been calculated, the last thing that effective_prio() does is within the non-RT priority range. For example - if a highly interactive task has a static priority of -20, it cannot be given a 5 point bonus since it already has the maximum non-RT priority.

5) Calculating timeslice: Timeslice is calculated by simply scaling a task's static priority onto the possible timeslice range and making sure a certain minimum and maximum timeslice is enforced. The higher the task's static priority the larger the timeslice it gets. The task_timeslice() function is simply a call to the BASE_TIMESLICE macro which is defined as:

```
#define BASE_TIMESLICE(p) (MIN_TIMESLICE \
    + ((MAX_TIMESLICE - MIN_TIMESLICE) * \
      (MAX_PRIO-1 - (p)->static_prio) / \
      (MAX_USER_PRIO-1)))
```

Essentially, this is the minimum timeslice plus the the task's static priority scaled onto the possible timeslice range, (MAX_TIMESLICE - MIN_TIMESLICE).

6) Interactivity credits: Interactive credits help to control the rise and fall rate of the interactive status of tasks. Essentially, tasks get an interactive credit when they sleep for a long time, and lose an interactive credit when they run for a long time. A task's interactive credit value is stored in its *interactive_credit* variable. If a task has more than 100 interactivity credits it is considered to have high interactivity credit. If a task has less then -100 interactivity credits it is considered to have low interactivity credit.

F. The Main Scheduling Function

The main scheduler function *schedule()* pick a new task to run and switch to it. The function is called whenever a task wishes to give up the CPU voluntarily and if scheduler_tick() sets the TIF_NEED_RESCHED flag on a task. The function scheduler_tick() is called during every system time tick, via a clock interrupt.

1) The schedule() function[2]: The first thing that schedule() does is check whether it is called at the right moment. After that, it disables preemption and determines the length of time that the task to be moved out has been running. That time is then reduced if a task has high interactivity credit since it would be undesirable for a task that usually waits on I/O to lose interactivity status due to a single long period of CPU usage. Next, if the function is entered off of a kernel preemption interruptible tasks with a signal pending get a state of TASK_RUNNING and uninterruptible tasks get removed from the runqueue. This is because if a task can be interrupted and it has a signal pending, it needs to handle that signal. Tasks that are not interruptible should not be on the runqueue.

Now, it is time to look for the next task to run. If there are no runnable tasks in the runqueue, an attempt at load balancing is made. If balancing does not bring any runnable tasks, then a switch to the idle task is made. If there are runnable tasks in the runqueue but not in the active priority array, then the active and retired priority arrays are swapped.

At this point there is a runnable task in the active priority array. Next, the active priority array's bitmap is checked to find the highest priority level with a runnable task. Subsequently, dependent sleeping tasks on virtual SMT CPUs are given a chance to run. If there is a dependent sleeper, the current CPU switches to idle so the dependent sleeper can wake up and do what it needs to do.

If there has not been a switch to the idle task at this point, a check is performed to see if the task chosen to run next is not RT and has been woken up. If it is not an RT task and was woken up, it is given a slightly higher sleep_avg and its dynamic priority is recalculated. This is a way to give another small bonus to sleeping tasks. Once this check has been performed and a bonus possible awarded, the wakeup flag is cleared.

Now schedule() is ready to make an actual task switch. This point in the algorithm is a goto target, and whatever

task is pointed to by the next variable is switched to. Earlier decisions to schedule the idle task had simply set next to the idle task and skipped to this point. Here, the previous task has its TIF_NEED_RESCHED flag cleared, context switch statistical variables are updated, and the previous task gets its run time deducted from its sleep_avg. Also, an interactive credit is deducted from the previous task if its sleep_avg dropped below 0 and its credit is neither too high nor too low. This is because if its sleep_avg is less than 0 it must not have been sleeping very much. With this setup complete, the actual context switch is made so long as the previous task and the new task are not the same task. After the context switch, preemption is reenabled since it was disabled during the scheduling algorithm. The final part of the schedule() function checks to see if preemption was requested during the time in which preemption was disabled, and reschedules if it was.

III. Starvation Problem

The O(1) Scheduler uses interactivity heuristics to provide better interactive performance to the user. This heuristics gives better interactivity by boosting interactive processes and by punishing non-interactive tasks. This is done by giving priority boost to the task which sleeps more and by decreasing the priority with respect to the time the task run in the CPU[3]. By giving this I/O bound task priority boosting task which sleeps given a priority boost and CPU-bound tasks are punished. So when a I/O bound process wait it gets higher priority and again considered for scheduling over the lower priority tasks due to preemption. So the lower priority tasks will get starved for CPU time. This will affect the interactive performance of scheduler.

The dynamic priority calculation algorithm of the O(1) scheduler causes the starvation of CPU-bound tasks. For example consider there are some five IO-bound tasks present in the active array. At the same time there are some CPU-bound jobs are also present in the active array. Normally the IO-bound tasks have higher priority compared to CPU-bound jobs. So first IO-bound jobs are considered for the scheduling. After execution, IO-bound jobs are re-inserted to the active array with a priority boost. So they are again considered for the scheduling, since the priority is boosted. This process will continue until the whole timeslice of the five IO-Bound jobs get expired or the starvation_limit reaches. During this time the CPU-Bound jobs are waiting for its turn. So they are getting starved and to avoid this starvation a mechanism called *starvation_limit* is implemented. When the time of waiting for the longest starving process reaches the starvation_limit all the tasks in the active array are scheduled by their priority and inserted into expired array, regardless of whether they are IO-bound or CPU-bound. Once the active array gets empty, the pointers of active and expired array are swapped to make active to expired and expired to active. So the starving tasks in the active array get a chance to run. The mechanism of starvation_limit fails since the value cannot be too short, because too short value can cause too many context switches and it affects the scheduler efficiency adversely. The value of starvation_limit cannot be too large, because it can cause starvation. Assigning dynamic value to starvation limit based on the status of queues is difficult.

To the best of our knowledge, very little effort had been put to enhance the O(1) scheduler to overcome the major drawbacks like the fairness issue. Qiao ShiJiao et. al. [7] proposed an EP algorithm to estimate the total computing time of a task which has direct impact on the load balancing decisions. Yugui Luo et. al. [5] and Wong et. al.[6] compares the Linux schedulers in terms of the fairness but has not proposed a solution to improve the performance of the O(1) scheduler.

A. Proposed Solution

The starvation of CPU-bound jobs can affect throughput of background jobs adversely. In order to improve performance of background jobs while maintaining a good degree of interactive responsiveness, we propose a modification to the original O(1) algorithm. The reason for low throughput of CPU-bound jobs in the original algorithm was the inefficient calculation of dynamic priority. The dynamic priority algorithm is implemented to get higher user responsiveness. This is done by boosting interactive jobs and penalizing cpu-bound jobs. The dynamic priority depends upon the interactiveness of a process. As said earlier, a highly interactive process gets high dynamic priority and a low priority otherwise. The dynamic priority is calculated by subtracting the bonus from the static priority value and the value of bonus is proportional to sleep_avg. The sleep_avg is calculated by adding the I/O wait time and subtracting the execution time from the current sleep_avg value. So a task with large I/O wait time will get higher sleep_avg and also higher dynamic priority. A task which is continuously executing, will get small value of sleep_avg and the corresponding dynamic priority will be low.

The starvation problem is fixed by applying a small modification to the existing algorithm. The dynamic priority calculation algorithm is modified for getting higher throughput for cpu-bound tasks. In the modified algorithm, the dynamic priority is calculated by considering the queue waiting time of a task inclusive of the time spent in either of the queues. Here sleep_avg is calculated using the iøwait time and array wait time to the current sleep_avg and subtracting the cpu-run time from the sleep_avg. By this modification a task which is sleeping more time will get higher dynamic priority and the task which is starving more will also get higher priority. The task which uses more cpu time will get less dynamic priority.

B. Feasibility of The Solution

Main design goal of the new algorithm is to prevent persistent starvation of CPU-bound processes, improve the interactive performance of the scheduler and maintain the scheduler complexity to O(1). The algorithm should not contain any loops, since it can increase the running time of the algorithm from O(1). Apart from these constraints, there is no major changes done to the data structures, because it can affect the efficiency of the algorithm. There are only two augmentation to the *task_struct* structure, i.e two additional variables - *CPU_enter_time* and *CPU_exit_time* which are not dependent on the existing algorithm. These variables store the timestamp of last schedule to CPU and that of the entry to the runqueue respectively. The values in nanseconds can be be obtained by an explicit call to the sched_clock() function. At this point we could calculate *array_wait*, the time since last schedule, by subtracting the CPU_exit_time from the CPU_enter_time. The array_wait value is also considered in calculating the sleep_avg. None of the operations above mentioned are iterative and or require major data structure changes. Consequently time complexity of the algorithm remains same and so will be the efficiency. Instead, an increase in the performance of the new algorithm is shown in the results.

IV. Implementation

The existing O(1) scheduler was modified and incorporated to the Linux 2.6 kernel and the new kernel was recompiled. Modification of the scheduler detailed below.

A. Modification of the Existing Algorithm

In this simple modification two variables were added to the *task_struct*, which is the process descriptor i.e. cpu_enter_time and cpu_exit_time. When a process is selected by the scheduler, the time is recorded in the variable cpu_enter_time by calling the named sched_clock(), which gives the current system time. Similarly the time at which, the process is exiting the cpu can also be calculated and can be stored in the variable cpu_exit_time. The difference between the cpu_enter_time and cpu_exit_time - array_wait - is the time the task spent on the runqueue. Because the time period between a process exiting the cpu and re-entering the cpu will be spent waiting in the runqueues. The sleep_avg is calculated in milli seconds. But the array_wait is obtained in nano seconds. So we have to convert nano seconds to milli seconds. This can be done by right shifting the nano second value by 20. This array-wait time in milli seconds is added to the sleep_avg value to get new sleep_avg value. Thus a task having longer array_wait will get higher sleep_avg and thus higher bonus.

B. Evaluation of new Linux Kernel

Interbench 0.30[9] is used to evaluate the performance of the new scheduler. After installing[8] the new kernel, Interbench-0.30 was used to evaluate the performance. The results of running benchmarks shows the interactive performance of the new kernel. It is designed to measure the effect of changes in Linux kernel designer system configuration changes such as CPU, I/O scheduler and file system changes and options. With careful benchmarking, different hardware can be compared. It is designed to emulate the CPU scheduling behavior of interactive tasks and measure their scheduling latency and jitter.

Interbench helps best to reproduce a fixed percentage of CPU usage on the machine currently being used for the benchmark. It saves this to a file and then uses this for all subsequent runs to keep the emulation of CPU usage constant. It runs a real time high priority timing thread that wakes up the thread or threads of the simulated interactive tasks and then measures the latency in the time taken to schedule. As there is no accurate timer driven scheduling in Linux the timing thread sleeps as accurately as Linux kernel supports, and *latency* is considered as the time from the sleep till the simulated task gets scheduled. Each benchmarked simulation runs as a separate process with its own threads, and the background load also runs as a separate process.

X is simulated as a thread that uses a variable amount of CPU ranging from 0 to 100%. This simulates an idle GUI where a window is grabbed and then dragged across the screen. **Audio** is simulated as a thread that tries to run at 50ms intervals that then requires 5% CPU. This behavior ignores any caching that would normally be done by well designed audio applications, but has been seen as the interval used to write to audio cards by a popular Linux audio player. Audio is simulated by running SCHED_FIFO by using the real time benchmarking option. **Video** is simulated as a thread that tries to receive CPU 60 times per second and uses 40% CPU. This would be quite a demanding video playback at 60fps. Like the audio simulator it ignores caching, drivers and video cards and are simulated with the real time option.

C. Fine Tuning the Scheduler

The performance of the modified O(1) scheduler is not improved under interactive task such as video or audio. Reason for the problem is that of considering non interactive tasks as interactive tasks. sleep_avg is the variable in the process descriptor, which is used to calculate interactive credit of a task. sleep_avg decides whether a task is I/O bound or CPU bound. High value of sleep_avg indicates that a process is I/O bound and small value of sleep_avg shows that a process is CPU bound.

In the modified scheduler, the sleep_avg is calculated by summing up the values such as I/O wait time and array wait time and subtracting CPU run time from the sleep_avg value. In this case for a process which is CPU bound, which is waiting in the run-queue for a long time will get a high value for array wait time, then the value of sleep_avg will be large and its interactive credit will be high. So a CPU

bound process that waits in the array for a long time will have high sleep_avg and will be considered as i/o bound process. So the I/O bound process are not considered for scheduling since the presence of high sleep_avg valued CPU bound processes. This is the reason for not improving the performance.

This can be solved by subtracting the value of array wait time from sleep_avg before calculating the interactive credit of a task. In the modified algorithm, the sleep_avg is used to calculate the dynamic priority of a task. After the dynamic priority is calculated, sleep_avg is used to calculate the interactive credit of the task, which is used to calculate the interactivity of a task. The value of array wait time is only needed to calculate the dynamic priority and after that the value of array wait time is subtracted from the sleep_avg, so as the value of array wait time will not affect the interactive credit.

The array wait time is added to the sleep_avg just before the calculation of dynamic priority of a process and the same amount of dynamic priority is subtracted just before the interactive credit is calculated so as to it will not affect the interactive credit. By doing this a CPU bound process will not be considered as a interactive task.

V. Results

In order to measure the performance improvement, we modified the O(1) scheduler of the Linux 2.6.22.14 version. Rresults of the modified scheduler were compared on the Linux 2.6.22.14 against the original version of the kernel. Current Linux version uses a scheduler called CFS. So we compared the performance of the modified scheduler with the performance of current scheduler as well. In short, we had three linux distributions for comparison viz. 2.6.22.14 (with original O(1) scheduler), 2.6.22.14(with modified O(1) scheduler) and 2.6.23(with CFS scheduler). Figures [1,2,3] shows the results the latency of Audio, Video and X workloads in the presence of other workloads of Interbench with the three schedulers. The reduction in latency for modified O(1) against the CFS and O(1) are clearly distinguished in all but a couple of the cases.

The analysis of the results of the benchmarking of the schedulers such as O(1) scheduler, CFS scheduler and O(1) scheduler modified shows that the overall performance of the modified scheduler is improved. The starvation of CPU bound threads is fixed and the interactive performance of the scheduler is improved. Though the overall performance of modified O(1) scheduler is improved by a large extent, the results show that performance deteriorates in a couple of cases. From the preliminary, it was infered that the calculation of sleep_avg has more influence than expected and hence we fine tuned the calculation of sleep_avg.

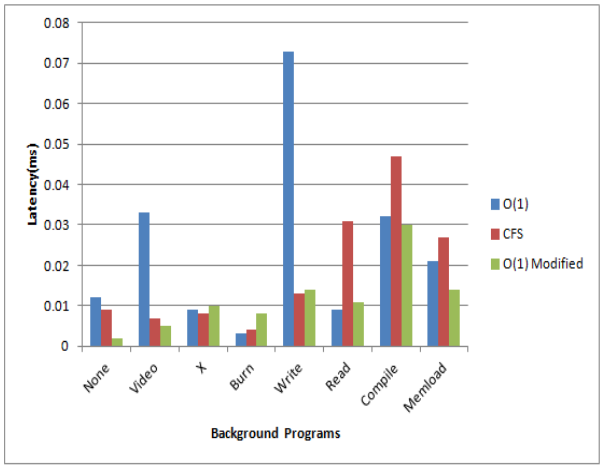

Figure 1. Latency of Audio in the presence of other loads.

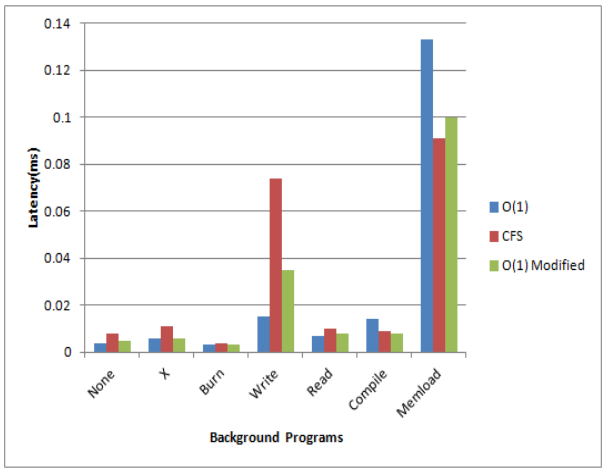

Figure 2. Latency of Video in the presence of other loads.

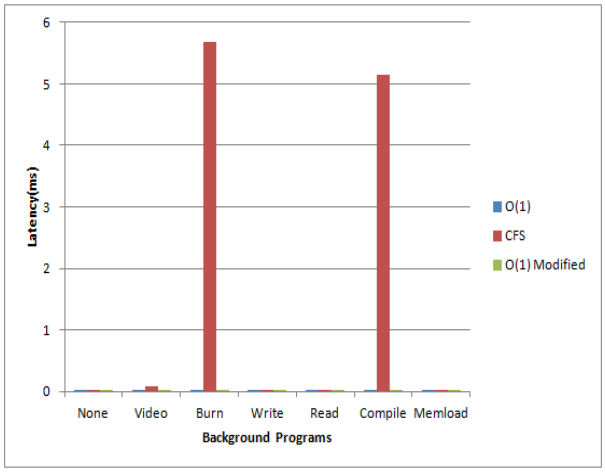

Figure 3. Latency of X in the presence of other loads.

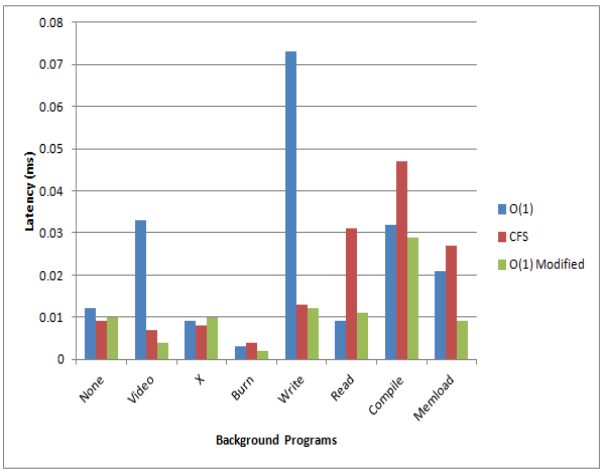

Figure 4. Audio in the presence of other loads.

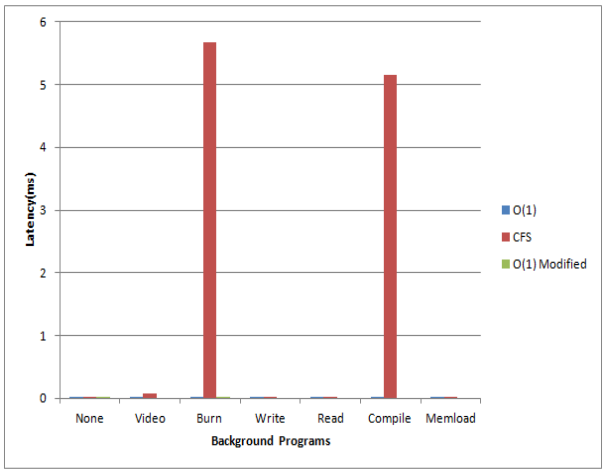

Figure 6. X in the presence of other loads.

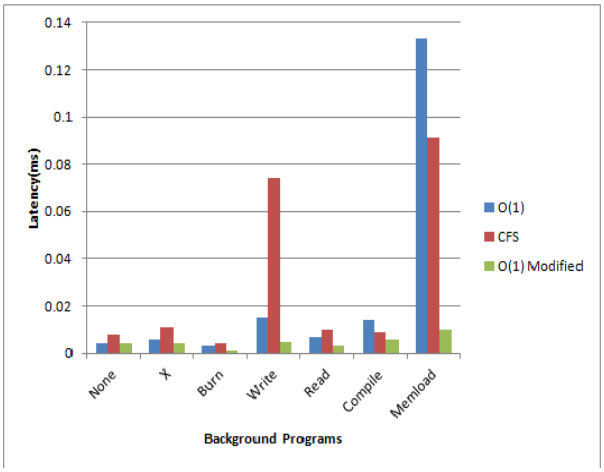

Figure 5. Video in the presence of other loads.

A. *Performance Analysis After Tuning*

The kernels after tuning the scheduler, was compiled and simulated again to measure performance. The results in Figures [4,5,6] shows that the fine tuning improves performance of the modified O(1) scheduler in terms of latency and thus would give higher throughput.

VI. Conclusion

As we have shown, the modified O(1) algorithm is better than both O(1) algorithm and CFS algorithm implemented in the Linux 2.6.22 kernel. It can be seen that complex data structure are not added and the algorithm does not introduce additional iterative operations. Hence, the computational complexity of the new algorithm remains unchanged. There is still open space for future work in terms of providing user fairness and preventing starvation in case of highly multi threaded processes.

Acknowledgment

We would like to thank Dr. B. Anil, Prof. A. Anvar and Prof. V. P. Sreejith of Government Engineering College Wayanad for their invaluable support and guidance.

References

[1] Josh Aas, *Understanding the Linux 2.6.8.1 CPU scheduler*, http://joshaas.net/linux/linux_cpu_scheduler.pdf, Available as on 21 December 2013.

[2] Daniele Bovet and Marco Cesati, *Understanding the Linux kernel*, OReilly (2006).

[3] Robert Love, *Linux kernel development 2nd Ed*, Pearson India (2010).

[4] Ma Wei-feng, Wang Jia-hai *Analysis of the Linux 2.6 kernel scheduler*, Computer Design and Applications (ICCDA), 2010 International Conference on, pp {V1-71 to V1-74}.

[5] Yigui Luo and Bolin Wu, *A comparison on interactivity of three Linux schedulers in embedded system*, Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on, pp 494-498

[6] Wong, C. S. and Tan, I. K T et. al. *Fairness and interactive performance of O(1) and CFS Linux kernel schedulers*, Information Technology, 2008. ITSim 2008. International Symposium on, 2008

[7] Qiao ShiJiao and Xu Hong and Yuan XingDe, *The improve of load balancing strategy on Linux system based on EP*, Computer Science and Network Technology (ICCSNT), 2011 International Conference on, pp 1477-1480

[8] Nixcraft, *How to Compile Linux kernel 2.6*, www.cyberciti.biz/tips/compiling-linuxkernel-26.html39., as on September 2011

[9] Con Kolivas, *Getting the interbench 0.30*, http://ck.kolivas.org/apps/interbench/, as on 17 Dec 2012

[10] Lan Yuqing, Xu Hao, Liu XiaoHui. *The Research of Performance Test Method for Linux Process Scheduling*, Information Science and Engineering (ISISE), 2012 International Symposium on, pp 216-219