Abstract

In prior research work, it has been demonstrated that Linux can starve CPU-bound processes in the presence of network I/O. The starvation of Linux CPU-bound processes occurs under the two Linux schedulers, namely the 2.6 O(1) scheduler and the more recent 2.6 Completely Fair Scheduler (CFS). In this paper, we analyze the underlying root causes of this starvation problem and we propose effective solutions that can mitigate such starvation. We present detailed implementations of our proposed solutions for both O(1) and CFS Linux schedulers. We empirically evaluate the effectiveness of our proposed solutions in terms of execution time and incoming traffic load. For our experimental study and analysis, we consider two types of mainboard architectures: Uni-Processing (UP) and Symmetric Multi-Processing (SMP). Our empirical results show that the proposed solutions are highly effective in mitigating the starvation problem for CPU-bound processes with no negative impact on the performance of network I/O-bound processes.


1. Introduction

Linux operating systems are widely deployed and used today. Any design flaw in the Linux kernel may drastically affect the systems, applications, and processes running on this kernel. Our prior research work [1] demonstrated that the Linux kernel can starve CPU-bound processes in the presence of network I/O-bound processes in the system. A CPU-bound process is defined as a user process that intensively uses the CPU without I/O requests whereas a network I/O-bound process is defined as a user process that intensively receives and/or sends network packets producing a high number of interruption requests in the system. This starvation
problem may affect the behavior of processes executing on the system. In previous work [1], we demonstrated the performance impact on the execution of user processes empirically by analyzing both Linux schedulers 2.6 O(1) and CFS under a variety of conditions such as different network I/O-bound processes, Network Interface Cards (NICs), and main-board architectures (uni-processor and symmetric multi-processor). We concluded that the starvation of CPU-bound processes occurs in Linux under all these different conditions, and for this reason, this drawback can affect many Linux-based computers in use today.

This paper is a major extension of our previous work presented in [1], and has distinct contribution and focus. The primary contribution of this paper is to propose a set of potential solutions that can address the starvation problem that is prevalent in both Linux schedulers: 2.6 O(1) and CFS. The proposed solutions have been carefully evaluated and analyzed in order to empirically demonstrate that the starvation problem can be mitigated in both schedulers 2.6 O(1) and CFS under mainboards equipped with a single processor and multiple processors. Furthermore, this work shows empirically that that the proposed solutions do not entail any negative impact on the performance of network I/O.

The remainder of the paper is organized as follows. Section 2 describes some related work about the starvation problem on the Linux kernel. Section 3 introduces a brief description of the root causes of the starvation problem we analyzed. Section 4 describes our proposed solutions to mitigate the starvation problem in both Linux schedulers 2.6 O(1) and CFS. The effectiveness of these solutions is studied and analyzed in Section 5. Section 6 presents further possible tuning and enhancements to the proposed solutions for both schedulers. Finally, Section 7 concludes our study.

2. Related Work on Starvation

A considerable amount of research work in the literature has focused on the Linux kernel and its performance. Aas [2] described how Linux 2.6 CFS and Linux 2.6 O(1) schedulers work in detail. The author noted the importance for tasks to be treated with a certain degree of fairness and to ensure that threads never starve in the Linux scheduler. Wong et al. [3] provided a detailed comparison of 2.6 O(1) and CFS Linux schedulers pointing out that CFS is more fair than 2.6 O(1) in terms of CPU time distribution without significantly affecting the performance of interactive processes. Moreover, the authors also demonstrated empirically that CFS is more efficient than 2.6 O(1) mainly because of the complex algorithms used to identify interactive tasks in 2.6 O(1). On the other hand, Turner et al. have shown that Linux CFS scheduling algorithm can still allow a task or a group of tasks to consume excess CPU share that can eventually cause an unacceptable utilization or latency in an otherwise idle system. They proposed a use of explicit upper bound on usage in addition to the lower bounds already provided by shares [4].
Kang et al. [5] experimentally showed the unexpected execution latency of real-time tasks in Linux 2.6 $O(1)$ scheduler, which, in many cases, are caused by starvation problems in the Linux scheduler. They proposed an alternative Linux scheduling algorithm based on a Weighted Average Priority Inheritance Protocol (WAPIP). WAPIP is a variation of the Priority Inheritance Protocol (PIP) [6], which assigns priorities to kernel-level processes at runtime by monitoring the activities of user-level real-time tasks. This algorithm significantly improves the latency of real-time tasks.

Kesavan et al. [7] investigated how to provide I/O service differentiation and performance isolation for virtual machines on individual multicore nodes in cloud platforms. As guest VM operating systems use adaptive resource management mechanisms such as TCP congestion avoidance and disk I/O schedulers, sharing I/O between VMs is fundamentally different from sharing I/O between processes. To address this problem, the notion of Differential Virtual Time (DVT) was used to provide service differentiation with performance isolation for VM guest OS resource management mechanisms.

Salah et al. [8][9] proposed an interruption handling scheme for the Linux kernel, which minimizes the overhead caused by heavy incoming network traffic. The goal of their research work was to avoid the starvation of the user processes by minimizing the time spent in handling interrupt requests. Moreover, they demonstrated empirically that their proposed scheme significantly improves the performance of general-purpose network desktops and servers running network I/O bound applications, when subjecting such network hosts to both light and heavy traffic load conditions [10].

Wu and Crawford [11] showed that Linux 2.6 $O(1)$ Scheduler can starve processes due to a misclassification of non-interactive network applications as interactive tasks such that they can unjustifiably obtain up to 95% of the CPU resources. The authors identified and analyzed the starvation problem in the Linux 2.6 $O(1)$ scheduler running on a single-processor mainboard. They proposed a generic solution to address the starvation problem based on a global minimum interactivity threshold to filter out all non-interactive processes in the system sending them to sleep.

Our previous research work [1] extended Wu and Crawford [11] research work. We studied the behavior of network processes at relatively normal network traffic loads, not exceeding 100 Mbps. As a result, our empirical results revealed the appearance of the starvation only for a particular traffic rate range close to $90Kpps$ for incoming traffic rate. Moreover, we empirically investigated the scope of the starvation problem by conducting several experiments with different network I/O-bound processes, network interface cards, and Linux kernels. We conducted a performance analysis using both single and multiple processor main board
architectures. Moreover, the root causes responsible for the starvation of CPU-bound processes were also identified and analyzed.

3. Root Cause of the Starvation Problem

This section is a summary of our previous research work [1], in which an in-depth analysis of both Linux schedulers: 2.6 $O(1)$ and CFS was carried out to determine the root causes behind the tendency of Linux schedulers to starve CPU-bound processes in the presence of network I/O-bound processes.

We set up a testbed where two computers (a sender and a receiver) were directly connected using a cross-over network cable under a 1Gb Ethernet link. The sender was running a traffic generation tool to send traffic to the receiver. The network generator tool used was Distributed Internet Traffic Generator (D-ITG) [12]. The receiver was running two different applications: a network I/O-bound application and a CPU-bound application. The former was in charge of receiving the traffic sent by the sender while the latter was used to measure the impact on the execution time of the CPU-bound application when traffic is received at different rates. Figure 1 shows this experimental setup. We used several Network Analysis Tools such as Ethereal [12], TCPDump [14] and ITGRecv [12] as network I/O-bound processes and we used the application Simplex as a CPU-bound process. Simplex [15] is an implementation of a commonly used nonlinear numerical method for optimizing multi-dimensional unconstrained problems by using an iterative search algorithm. A parameter can be provided to the Simplex application to set the number of iterations to be executed, which in turn, is directly related to the execution time of this application. This time increases when Simplex gets less CPU resources due to high load on the system or unfair scheduling of Simplex application. As the CPU gets busier, this execution time interval increases. During the experimental test, the sender stressed the receiver by sending it packets at different rates.

![Figure 1. Experimental Setup](image)

When the sender transmits packets at around 90Kpps (packet per second), almost all the execution time of the Simplex process was spent in the kernel mode processing the large number of interruptions requested by the network I/O-bound process. This causes a starvation problem for the Simplex process. This starvation occurs because of the large number of involuntary context switches that the Simplex application is forced to do. This demonstrates that the scheduler algorithm in the Linux kernel handling user interactivity is the root cause of the
problem. The starvation problem was shown to occur on both Linux schedulers: 2.6 \( O(1) \) and CFS, and the root causes of this starvation were studied on both of them. The root causes are described in more detail in the following subsections.

### 3.1 Linux 2.6 \( O(1) \) Scheduler

Figure 2 shows the process scheduling algorithm used in the Linux 2.6 \( O(1) \) scheduler [16]. There are two different arrays of queues for allocating processes: an active array and an expired array. The former contains processes that can receive CPU resources, whereas the latter contains processes that might wait till they are reallocated in the active array. Each array of queues is indexed by means of a priority index. All the processes available in a queue are associated to this priority. A process is selected to receive CPU resources according to its priority. When a process receives CPU resources, an algorithm is executed to dynamically recalculate its new priority value at run-time. This algorithm determines where this process should be queued after it has consumed its CPU execution time quantum.
The dynamic priority calculation algorithm takes into account the interactivity of the process to calculate the new priority. It considers two types of processes according to its interactivity: interactive or I/O-bound processes and non-interactive or CPU-bound processes. This algorithm uses the $\text{sleep\_avg}$ values associated with each process to determine its interactivity [16, 17]. This value determines an average time that the process sleeps, which is calculated from the individual sleep periods during which the process is put into sleep, namely $\text{sleep\_time}$ values. This scheduler uses an upper threshold in order to control the maximum value that a process $\text{sleep\_avg}$ value can reach. This value is controlled by the $\text{MAX\_SLEEP\_AVG}$ parameter of this 2.6 $O(1)$ scheduler. However, the design of the 2.6 $O(1)$ scheduler does not impose any lower limits on the process's $\text{sleep\_time}$ value [18, 19]. Hence, we empirically demonstrated that processes which have many short sleeping periods can be misclassified as interactive processes and thus receive more CPU resources than other processes. Network traffic analysis tools such as Ethereal and ITGRecv are some examples of those processes, since the scheduler is unable to estimate their dynamic priority level and interactivity status correctly.

The network analysis tools used in our experimental setup are usually non-interactive and are supposed to leave CPU processing power for other more interactive processes in the system. However, these processes have around 90% of their $\text{sleep\_time}$ value below 10 ms when the sender is stressing the system at traffic rate 90Kpps. As a result, they are misclassified as interactive. This causes their priority to be increased, so they can acquire even more CPU resources and cause other CPU-bound processes in the system to starve. In fact, these results match those obtained by Wu and Crawford [11], which showed in their theoretical and experimental work that "the 'relatively fast' non-interactive network process might frequently sleep to wait for packet arrival. Though each sleep lasts for a very short period of time, the wait-for-packet sleeps occur so frequently that they lead to interactive status for the process."

Several root causes are responsible for the starvation of the Linux scheduler around a particular traffic rate of 85 Kpps. First, the Linux 2.6 $O(1)$ scheduler misclassifies non-interactive network processes as interactive processes due to the heuristic algorithm used to carry out this classification. As a consequence of this misclassification, the network I/O-bound process is able to monopolize the CPU resources at certain traffic rates (85 Kpps in this experiment). Second, the reason why this monopolization can be done is because of the starvation heuristics [1] available in the scheduler to identify starvation in the system are unable to recognize the starvation of CPU-bound processes at this specific traffic load and to respond accordingly. Third, the misclassification of the network process into the interactive class leads to an incorrect assignment of dynamic priorities during the process execution, which in turn, may worsen the starvation problem for other processes.
Finally, all processes sleeping for small time intervals receive the advantages associated with their sleep times unless other circumstances limit their evaluation of the sleep time [1].

In summary, the analysis of the sleep_time value of the network I/O-bound process has shown that the root cause of the starvation problem is the sleeping behavior of this process. It causes the scheduler to increase its dynamic priority and treat it as an interactive process. This is because the scheduler design lacks the ability to recognize those non-interactive processes that have many short-term sleeps. It has been demonstrated that the repetitive short-term sleeps performed by the network I/O-bound process when it receives packets at particular traffic rate around 90Kpps causes a misclassification of this process as an interactive task which in turn causes a starvation problem for the CPU-bound process.

3.2 Linux CFS

The Linux CFS scheduler is focused on the running time of each task. It tries to emulate an ideal fair multitasking parallel processor, by dividing the processor time almost equally among all processes in the runqueue [20, 21]. Figure 3 shows the CPU time divisions done according to the number of processes available in the runqueue.

![Figure 3. CFS per-process clock with n processes in the runqueue](image)

The CFS scheduler provides many tunable parameters that allow for an immediate real-time interaction with the system scheduler [22]. One of these parameters is the sysctl_sched_wakeup_granularity. This parameter is directly related to the sleeping behavior of any normal process, i.e. non-real time process. Such a process is scheduled under the SCHED_OTHER scheduling policy. It is worth noting that we identified that the starvation problem in the 2.6 O(1) scheduler was related to the sleeping behavior of the network I/O-bound process. So, it is very probable that this sleeping behavior is the reason for the starvation problem in this scheduler as well. The parameter sysctl_sched_wakeup_granularity controls the wake-up granularity of the processes under SCHED_OTHER scheduling policy (normal processes). This wake-up granularity allows adapting workloads by means of delaying the preemption of processes and thus reducing over-scheduling. In kernel 2.6.24, this parameter is initialized according to the following formula [23]:

\[
\text{Wall Clock} = \begin{cases} 
1 & n = 1 \\
2 & n = 2 \\
4 & n = 4 \\
8 & n = 8 
\end{cases}
\]
sysctl_sched_wakeup_granularity = 10ms \times (1 + \log(ncpus)) \quad (1)

Where \( ncpus \) is the number of processors in the system. Thus, the default value of this parameter is 10 ms in single-processor systems. It is worth pointing out that the default value of this parameter has been changed in kernel 2.6.27 to 5 ms [24].

Network process tools such as ITGRecv and the Simplex application are normal tasks. Thus, these tasks are to be preempted when their CPU time is expired. This checking is done by the function `check_preempt_wakeup` defined under `sched_fair.c` scheduler module [25]. This function takes into account the parameter `sysctl_sched_wakeup_granularity` to determine whether the task is to be preempted or not. Figure 4 shows a simplified version of the function `check_preempt_wakeup` [26].
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p) {
    struct task_struct *curr = rq->curr;
    struct cfs_rq *cfs_rq = task_cfs_rq(curr);
    struct sched_entity *se = &curr->se, *pse = &p->se;
    unsigned long gran;
    gran = sysctl_sched_wakeup_granularity;
    if (pse->vruntime + gran < se->vruntime)
        resched_task(curr);
};

Figure 4. Simplified version of the check_preempt_wakeup function

This function performs several checking for processes belonging to different scheduling policies including real-time, batch and group scheduling. Then, the function compares the values of vruntime (the CPU time consumed by a process) for the current process with those of the newly awakened process. If the difference of the vruntime values of the current process and the newly awakened process is more than the wake-up granularity threshold (i.e. sysctl_sched_wakeup_granularity), the awakened process preempts the current process. Otherwise, the current process continues getting extra CPU resources. Note how this wake-up granularity threshold parameter is used to adapt workloads, as it enables the usage of more timeslots before preempting the process.

The experimental setup we used helped to determine that the CPU-bound process is starved when the network I/O-bound process is being stressed at a traffic rate between 70 – 100 Kpps. The root cause is the sleeping behavior of the network I/O-bound process, which causes an excessive preemption of the CPU-bound process because the default setting of the parameter sysctl_sched_wakeup_granularity is 10 ms [17, 24]. This excessive preemption leads to an over-scheduling in the Linux scheduler and it causes the starvation of the CPU-bound process. In fact, the performance of the CPU-bound process improves as the sysctl_sched_wakeup_granularity parameter is increased. This fact shows that the root cause of the starvation problem is associated with this setting of the parameter (sysctl_sched_wakeup_granularity).

4. Proposed Solutions

In this section, we propose solutions to mitigate the starvation problem of CPU-bound processes in the presence of network I/O. Our proposed solutions improve the scheduler design in order to mitigate the effect of the short-term sleep behavior of network I/O-bound processes on the performance of CPU-bound processes. The main solution is based on the inclusion of a global lower threshold that can be used in conjunction with the value of sleep_time associated with the network process. To avoid the starvation problem, a process is sent to sleep when the sleep_time associated with the process is less than this threshold. This threshold may be
included in the 2.6 $O(1)$ scheduler. Moreover, the same solution can be implemented in the CFS scheduler by changing the default setting of the `sysctl_sched_wakeup_granularity` parameter, which has to be carefully chosen in order to mitigate the negative impact of network I/O-bound processes. The proposed solution for each scheduler is discussed in detail in the following subsections.

### 4.1 Linux 2.6 $O(1)$ Scheduler

The analysis of the `sleep_time` value of the network process has shown that the sleeping behavior of the network process can cause the scheduler to boost its dynamic priority and be treated as an interactive process. This is because the design of the scheduler lacks the ability to recognize non-interactive processes that have many `short-term` sleeps. One way to address this deficiency is to insert a `static global threshold` to control the minimum `sleep_time` value used in the calculation of the `average sleep time` (`sleep_avg`) associated with any process. The `average sleep time` associated with a process is the key value used to determine the dynamic priority assigned to a given process. If there are many `short-term` sleeps below this threshold, they are ignored in the calculation of the `average sleep time`, which in turn, avoids the misclassification of the network I/O-bound process as an interactive task.

Figure 5 shows the flow chart of the function `recalc_task_prio()` [27]. This is the scheduler function that calculates the priority associated with a given process at run-time. The shaded boxes shown in Figure 5 represent the changes included in this function in order to insert the `static global threshold` into the scheduler.

We modified the Linux 2.6 $O(1)$ scheduler code to include a `static global threshold` for the `sleep_time` associated with any process. This threshold determines if the `sleep_time` associated with a given process is taken into account when calculating the average sleep time of that process. This average sleep time is used by the `recalc_task_prio()` function to determine the dynamic priority associated with the process. The threshold has been defined as `MIN_SLEEP_TIME_MS`. To enable the configuration of this parameter from the user level, we have defined an entry in the `/proc` file system through which this value can be modified instantly without the need to reboot the kernel. The entry is referred to as `MIN_SLEEP_TIME_MS` on the `/proc/net` folder. The following echo command can be used to set the value of this parameter:

```
# echo 10 > /proc/net/network_min_sleep
```

It is to be noted that the `MIN_SLEEP_TIME_MS` value can also be read using the following command:

```
# more /proc/net/network_min_sleep
```
Figure 5. Flow chart of the proposed recalc_task_prio() function

The value of the MIN_SLEEP_TIME_MS threshold should be varied and analyzed in order to find the most appropriate value that alleviates the starvation problem without affecting the performance of other processes.
The analysis of the sleep_time values carried out in our previous work [1] demonstrated that the most suitable value for the sleep_time should be set between 10 and 15 milliseconds. The inclusion of the MIN_SLEEP_TIME_MS parameter enables control over the minimal value of the sleep_time that can be counted in the sleep_avg value used in the dynamic priority calculation. In particular, the MIN_SLEEP_TIME_MS should be set to a value greater than 10 ms to prevent the observed starvation problem.

4.2 Linux Completely Fair Scheduler (CFS)

Linux CFS comes with several tuning parameters which can be set through the sysctl command. In fact, the default settings of those parameters is not always optimal, and the performance of the scheduler can be greatly improved if those parameters are re-assigned more appropriate values for the current system load [28]. The sysctl_sched_wakeup_granularity parameter is a Linux CFS parameter which has a special importance under the SCHED_OTHER scheduling policy. It determines whether a process can receive more CPU resources before being preempted in order to adapt workloads and reduce the over-scheduling time associated with the preemption of any process. The default value of the sysctl_sched_wakeup_granularity parameter is 10 ms in kernel 2.6.24 and 5 ms in kernel 2.6.27.

The analysis carried out in our previous work [1] demonstrated that the most recommended setting of the parameter sysctl_sched_wakeup_granularity to eliminate the starvation problem should satisfy rule (2) below:

\[ \text{sysctl_sched_wakeup_granularity} > y \]  \hspace{1cm} \text{(2)}

where \( y \) is the sleep_time value associated with the network I/O-bound process, which lies between 10 and 15 ms when the network traffic rate is in the critical range of 70–100 Kpps. Thus, neither the old nor the new default setting of sysctl_sched_wakeup_granularity satisfies the derived rule.

A simple solution to the starvation problem is to modify the default setting of sysctl_sched_wakeup_granularity so that is greater than the sleep_time values associated with the network I/O-bound process when the network traffic rate is between 70 and 100Kpps, i.e. greater than 15 ms. With this solution, the default value of the parameter sysctl_sched_wakeup_granularity is recommended to be established at 20 ms. Our previous analysis [1] also demonstrated that there is no noticeable gain in performance when the value of sysctl_sched_wakeup_granularity exceeds 20 ms.

5. Evaluation of our Proposed Solutions

This section evaluates the effectiveness of our proposed solutions to mitigate the starvation problem. The effectiveness is measured using different criteria; i) the solution effectively mitigates the starvation problem on
both schedulers 2.6 $O(1)$ and CFS; ii) the solution does not affect the network I/O throughput; and iii) the solution is effective on both uni-processor and symmetric multi-processor main-board architectures.

To demonstrate the effectiveness of the proposed solutions, they have been implemented in both Linux schedulers and their performances have been empirically analyzed. The following subsections analyze the effectiveness of the proposed solutions with respect to the criteria described above.

5.1 Experimental Testbed and Methodology

The experimental testbed used to assess the effectiveness of the proposed solution is composed of two machines: a sender and a receiver connected by a 1-Gbps Ethernet crossover cable. Conceptually, this scenario is analogous to the one previously shown in Figure 1. The sender runs a network generator tool called Distributed Internet Traffic Generator (D-ITG) [12]. The receiver runs the Simplex [15] application to measure the CPU time available to user applications. Simplex is configured to be executed in 1 sec. However, the busier the CPU gets, the larger is this execution time interval. Moreover, the receiver also runs a network traffic analysis tool, in order to have a network I/O-bound process running at the same time. This tool processes almost all the network I/O received, which in turn, is generated by a network traffic generator tool deployed at the sender. The network traffic generator tool generates packets according to many settings such as ratio packet/second, packet length, packet type (UDP/TCP) and others. For this experiment setup, a network stream of 64bit payload length UDP packets has been used to analyze the behavior of the receiver for different network traffic rates (between 0 to 150 Kpps). The network traffic analysis tool executed at the receiver is ITGRecv, which is part of the Distributed Internet Traffic Generator (D-ITG) open source tool [12].

The sender runs Fedora Core 5 Linux with the Linux kernel 2.6.16. It is an Intel Xeon CPU (5 GHz) with 4GB RAM and equipped with a network adapter with a BCM 5751 network controller. The receiver runs the modified Linux scheduler implementation to be analyzed. It is an Intel Pentium 4 CPU (3.2 GHz) with 512 MB RAM and equipped with a network adapter with a BCM 5752 network controller. Both machines have been booted at run level 3, and no services were running in the background in order to minimize the impact of other system activities on performance and measurement. Moreover, the Ethernet link flow control was disabled ensuring that the network throughput is not throttled between the sender and the receiver.

There are several characteristics associated with each running process useful to diagnose the starvation problem. These measures are the Process Execution Time, System Time, User Time, Involuntary Context Switches, and CPU Availability. We used the Linux time utility [29] on the Simplex application in the receiver to measure these metrics. Process Execution Time is defined as the time Simplex takes to complete its execution. System Time is the CPU time Simplex spends in kernel mode. User Time is CPU time Simplex
spends in user mode. **CPU Availability** is the percentage of CPU resources **Simplex** has consumed. It is computed as \((\text{System Time} + \text{User Time}) / (\text{Process Execution Time})\). Finally, the **Number of Involuntary Context Switches** is defined as the number of times **Simplex** has been preempted, or forced to give away the CPU resources involuntarily, either due to a time quantum expiration in the scheduler or because of the presence of a higher priority task than **Simplex** in the scheduler.

### 5.2 Empirical Analysis of the Proposed Solutions

This section presents our empirical results obtained with our proposed solutions implemented in both schedulers to determine their effectiveness in mitigating the starvation problem.

#### 5.2.1 Linux 2.6 O(1) Scheduler

In order to analyze the effect of the **global lower threshold** (referred as **MIN_SLEEP_TIME_MS**) variable inserted in the **Linux 2.6 O(1) scheduler** on the starvation problem, the value of this threshold has been varied from 15 to 1000 ms using the experimental setup described in section 5.1. The minimum value was experimentally demonstrated in our previous research [1] according to the sleeping behavior of the network I/O-bound process at the critical 90Kppp traffic rate. It is worth noting that this traffic rate is the only rate that causes the starvation problem. The maximum value is determined by another parameter available in the scheduler, namely **MAX_SLEEP_AVG**. This parameter is the upper threshold to control the maximum average sleeping time calculated for any process in the system. The performance of the CPU-bound **Simplex** application has been measured using the **time** utility for each value assigned to **MIN_SLEEP_TIME_MS**. Figure 6 shows the execution results including **process execution time, user time, system time, involuntary context switches** and **CPU availability** for every **MIN_SLEEP_TIME_MS** value analyzed. In order to compare the solutions proposed in this paper with the original **Linux 2.6 O(1) scheduler** code, the same experiment was carried out using the value 0 ms for the parameter **MIN_SLEEP_TIME_MS**, which, in effect, disables the threshold and runs the original scheduler code.
Figure 6. Performance of Simplex process with different MIN_SLEEP_TIME_MS values

Figure 6(a) clearly shows the effect of the global lower threshold. The Simplex process is starved when the network I/O-bound process received packets at the specific rates 40 Kpps and 75 Kpps in the case of the original Linux scheduler code. Note that at 75 Kpps, the Simplex execution time is 170 times higher than at 0 Kpps (no network traffic), which illustrates an important starvation problem. However, if the MIN_SLEEP_TIME_MS threshold is established at 15 ms there is a significant reduction in the Simplex execution time at these two traffic rates (40 and 75 Kpps). Our previous experimental results showed that 90% of the time durations Simplex sleeps for when there is a network I/O interruption is less than 10 ms. For this reason, if this threshold is established at this value, the dynamic priority calculation can be controlled and the network I/O process will not be misclassified as interactive. It is important to note that this threshold does not have any significant impact on the Simplex execution time at any other traffic rate as can be shown in Figure 6(a). This fact clearly shows the added value of the global lower threshold since it helps to stabilize the performance of the CPU-bound process. Moreover, the other values assigned to the MIN_SLEEP_TIME_MS threshold (from 30 to 1000 ms) have a very small impact on the Simplex execution time.

It is important to note that the first peak in the performance of the Simplex process around 30-40 KPPS is unrelated to the starvation issue. This peak can be interpreted by referring to the concept of Linux NAPI, which is a packet reception mechanism implemented in Linux 2.6 to alleviate the problem of receive livelock through “interrupt coalescing” [9]. In Linux NAPI, when the “first” packet arrives and is copied to the device DMA ring, an interrupt (i.e. RxInt) is generated to notify the CPU about the availability of the packet, the network device is added to the poll list, and all further packet arrival interrupts are disabled. Thus, new arriving packets are being copied to the device DMA silently. When the network process wakes up and processes all the packets in the DMA, the interrupts are re-enabled again [9]. At low network traffic load, i.e. between 0 Kpps and 40 Kpps, the number of sleeps per second is actually linear, such that there is almost one sleep for each arriving packet. This is because the arrival rate of the packets is very slow compared to the network process’s packet processing rate, such that one packet is completely processed and removed from the device DMA before the next packet actually arrives. Thus, NAPI re-enables the RxInt interrupts and the network process goes to sleep waiting for the next packet. However, beyond 40 Kpps, the rate of packet arrival exceeds the network packet processing rate resulting in new packet arrivals while the network process is still processing some previously arrived packets. The newly arrived packets are copied silently to the device DMA, and the network process handles them before going to sleep. Nonetheless, at this traffic range, the packet arrival rate is not too high such that the network process will eventually consume all packets in the DMA before its timeslice expires, and then
go to sleep again. This situation causes a slight increase, or mostly a balance, in the number of sleeps of the network process between 40 Kpps and 70 Kpps.

Figures 6(b), 6(c) and 6(d) show similar trends to the one shown in Figure 6(a). Figure 6(b) shows the user time of the Simplex process against the different MIN_SLEEP_TIME_MS values. It shows the significant improvement of the Simplex performance when this threshold is established up to 15 ms. Similarly, Figures 6(c) also shows the same performance improvement on the system time of Simplex process. However, the improvement is higher for the system time than for the user time. It is related to the root cause of the starvation problem which is directly related to the scheduling algorithms and the associated involuntary context switches executed during the system time. In fact, the reduction in the number of involuntary context switches of Simplex is shown in Figure 6(d). The global lower threshold established at 15 ms reduces the number of involuntary context switches by more than 2.25 million context switches at 75 Kpps, and consequently the system time is also reduced significantly.

Figure 6(e) shows the CPU availability for Simplex application based on different values established using the global lower threshold parameter. The results show that increasing the global threshold value allows Simplex to get more CPU resources. However, the results available in Figure 6(e) differ from the other results in Figure 6 because of the amount of CPU resources available for Simplex continues to increase for MIN_SLEEP_TIME_MS values greater than 15 ms. Increasing MIN_SLEEP_TIME_MS beyond 15 ms affects other processes in the system since their sleep times are not being counted in their sleep averages. This fact gives Simplex an unfair advantage of CPU resources at the expense of other processes in the system. In fact, there are almost no processes in the system that are treated as interactive when MIN_SLEEP_TIME_MS is set to 1000 ms and, as a result, Simplex can monopolize CPU resources. In this case, Simplex process always receives more than 70% of the CPU resources available in the system.

In addition, we also investigated the behavior of the solution when more than one CPU-bound process and more than one I/O-bound process are contending for the CPU resources. In particular, we use the same testbed as before, but in this case, we have used at the recipient side two ITGRecv network I/O-bound processes receiving two network streams with the same rate measured in KPPS. On the other hand, two Simplex CPU-bound processes are also running on the recipient host. The average time of the two simplex processes is plotted against the network arrival rate in KPPS for several settings of the MIN_SLEEP_TIME_MS parameter. The results show a similar behavior to the case of a single network I/O-bound process and a single CPU-bound process. Both Simplex processes are starved when the network I/O-bound processes received packets at the specific rates of 40 Kpps and 75 Kpps in the case of the original Linux scheduler code. On the other hand, when the MIN_SLEEP_TIME_MS threshold is established at 15 ms, there is a significant reduction in the execution
time of both Simplex processes at these two traffic rates (40 and 75 Kpps), which clearly demonstrates the effectiveness of the proposed solution.

![Figure 7. Average performance of two Simplex processes with two ITGRecv processes for different MIN_SLEEP_TIME_MS values](image)

To summarize, the results show a significant improvement in the performance of the CPU-bound process in the presence of a network I/O-bound process when the global lower threshold is inserted in the scheduler code. This improvement occurs when this threshold is set up to 10 ms. Moreover, there is only a slight increase in the improvement as the threshold value ranges between 15 and 1000 ms. However, the CPU-bound processes might get an unfair share of the CPU resources when increasing this threshold beyond 15 ms. For this reason, the recommended value for this threshold is 15 ms, which in turn, eliminates the starvation problem.

### 5.2.2 Linux CFS

To investigate the effect of the parameter `sysctl_sched_wakeup_granularity` on the starvation problem, its value was set in the range 0.1 to 30 ms using the experimental setup previously described in section 5.1. Note that this range takes the default value if this parameter is set to 10 ms. In addition, all the measurements were taken while the receiver was receiving network traffic at the critical rate around 90 Kpps. The results of this experiment are shown in Figure 8.
Figure 8. Performance of the Simplex process with `sysctl_sched_wakeup_granularity` ranging from 0.1 to 30 ms

Figure 8(a) plots the Simplex execution time against the range of values of the wake-up granularity parameter. In general terms, the Simplex execution time decreases when the wake-up granularity parameter increases. Moreover, Figure 8(b) plots the number of involuntary context switches Simplex is forced to do against the range of values of the wake-up granularity parameter. Both figures show very similar results. Three distinguishable regions become noticeable in Figures 7(a) and 7(b) and these include: 1-10 ms, 10-20 ms, and 20-30 ms. For simplicity, we refer these regions as small, medium, and large wake-up granularity regions, respectively.

The Small Wake-up Granularity Region produces a small improvement in the Simplex execution time. In fact, the Simplex execution time is only reduced by 3 s (from 112 s at 1 ms to 109 s at 10 ms). This fact can be directly mapped to the general sleeping behavior of the network process, which shows that more than 80% of the sleeps are shorter than 10 ms. Since the network process sleeps less than 10 ms for more than 80% of the time, the Simplex process is more susceptible to being preempted by the network process when the value of the parameter `sysctl_sched_wakeup_granularity` is less than 10 ms. This explains why the performance of Simplex does not have a significant improvement when the value of this parameter ranges from 1 to 10 ms.

In the case of the Medium Wake-up Granularity Region, the performance of the Simplex application improves significantly. The Simplex execution time sharply drops from 109 s at 10 ms to 2.5 s at 20 ms. Note that more than 80% of the network process sleeps are shorter than 10 ms. Thus, since the
sysctl_sched_wakeup_granularity value is higher than 10 ms, Simplex is less susceptible to being preempted by other awakened network processes and consequently, there is a decrease in the number of involuntary context switches. As a result, Simplex gets more CPU time and its performance improves rapidly.

Finally, the Large Wake-up Granularity Region does not produce much improvement in the Simplex execution time. In particular, the Simplex execution time is reduced from 2.54 s at 20 ms to 2.36 s at 30 ms. This behavior is reasonable since almost all the overhead in the Simplex application was caused by the network process and it has already been eliminated when the value of the sysctl_sched_wakeup_granularity parameter is set to 20 ms. In fact, only the 10% of the sleeps of the network process are longer than 15 ms. Thus, increasing the value of this parameter from 20 ms to 30 ms does not really produce appreciable significant impact on the Simplex execution time and the number of involuntary context switches.

In addition, Figure 8(b) shows the number of involuntary context switches Simplex is forced to do against the ranging values of the parameter sysctl_sched_wakeup_granularity. The results are similar to the results shown in Figure 8(a). The improvement in Simplex performance in the Small Wake-up Granularity Region is small, since the reduction of involuntary context switches is minimal. The involuntary context switches are sharply reduced in the Medium Wake-up Granularity Region producing a significant improvement in the Simplex performance. Finally, the Large Wake-up Granularity Region does not produce significant improvement in the Simplex performance since the number involuntary context switches are not affected in this region.

Moreover, to investigate how the suggested solution behaves under multiple network I/O-bound processes and multiple CPU-bound processes, we analyzed the behavior of the modified CFS Scheduler when two Simplex processes are running along with two ITGRecv processes at the recipient side, using the same experimental setup described above. Figure 9 shows the average execution time of the two Simplex processes against the range of values of the wake-up granularity parameter. Similar to the previous results, the Simplex execution time decreases when the wake-up granularity parameter increases. In addition, the three distinguishable regions (1-10 ms, 10-20 ms, and 20-30 ms) become again noticeable in the plot, which demonstrates that the proposed solution is also effective when multiple CPU-bound and network I/O-bound processes are present in the system.
To sum up, the results demonstrate the effectiveness of our proposed solution. The Simplex execution time is improved from almost 110 s when `sysctl_sched_wakeup_granularity` is set at 5-10 ms to 3 s when this parameter is set at 20 ms. This is the recommended value (20 ms) with our proposed solution because it eliminates the starvation problem in the Linux CFS scheduler.

### 5.3 Impact on Network I/O-Bound Processes

In this section, we demonstrate that our proposed solutions do not change the original performance of both types of schedulers. We present an experimental analysis of the performance of network I/O-bound process with the modified versions which implement the proposed mitigation solutions for both types of schedulers.

#### 5.3.1 Linux 2.6 O(1) Scheduler

To demonstrate that the proposed solution does not incur any overhead or have any side-effects on the performance of the network I/O-bound process, we conducted a detailed network performance analysis. Performance results have been collected when the `global lower threshold` has been set in the range 0 to 1000 ms in order to analyze the impact of this parameter on the network performance. The performance metrics used to analyze the performance of the network I/O-bound process are: network throughput and latency. The network throughput is measured as the number of packets received by the network interface card and delivered to the application. The network latency is measured using the round trip time, i.e. the time it takes for a packet to make a round trip from the sender to the recipient and then come back to the sender.

The experimental setup used to measure the network performance is composed of a sender and a receiver. The receiver runs the *Linux kernel* with our solution implemented. The sender sends network packets to the receiver.
which sends those packets back to the sender. The traffic generation tool and the traffic analysis tools are the same used in previous experiments, i.e. the Distributed Internet Traffic Generator (D-ITG) [12]. In this experiment setup, the performance results are collected at the sender side. The receiver runs Simplex during the experiment whilst the MIN_SLEEP_TIME_MS threshold is varied from 0 to 1000 ms. The network I/O process’s throughput and latency results are shown in Figure 10 and Figure 8 respectively.

Figure 10. Network throughput with different MIN_SLEEP_TIME_MS values

Figure 10 shows the variation of network throughput against variable network loads for different values of the MIN_SLEEP_TIME_MS threshold. Note that the 0 ms value in the parameter MIN_SLEEP_TIME_MS represents a scenario with the original scheduler design (without our solution implemented). There is almost no significant difference in the throughput of the network process when the MIN_SLEEP_TIME_MS value is varied between 0 and 100 ms. This demonstrates that the insertion of the global lower threshold does not cause any negative impact on the throughput of the network processes. Thus, the improvement in the performance of Simplex process when the parameter MIN_SLEEP_TIME_MS is established between 15 and 30 ms does not cause any degradation in the network process performance.
Figure 11. Network latency with different MIN_SLEEP_TIME_MS values

Similarly, Figure 11 does not show a significant difference in the network latency when the parameter MIN_SLEEP_TIME_MS varies between 0 and 100 ms. This demonstrates that the proposed solution does not increase the network latency for our recommended value for the parameter MIN_SLEEP_TIME_MS, namely 15 ms.

For the case where the parameter MIN_SLEEP_TIME_MS is set to 1000 ms, there is a significant impact on the network throughput and latency. In this case, all the processes are misclassified as non-interactive since the sleep_avg calculation is disabled because of the value of the threshold, since the default setting of the maximum value that a process sleep_avg value can reach, which is controlled by the MAX_SLEEP_AVG parameter of this 2.6 O(1), is 1000 ms. Consequently, the network I/O-bound process may be drastically affected as a result of this misclassification. This fact is clearly noticeable in both Figures 8 and 9, where there is a clear decrease in the network throughput and a clear increase in the network latency.

As a result, the performance analysis of the network process with our proposed solution shows that the solution has no negative impact on the performance of the network process. In addition, the performance analysis also shows that it is not recommended to disable the calculation of the sleep_avg time because it would severely affect the performance of networked applications.

5.3.2 Linux CFS

In order to demonstrate that the proposed solution does not cause any extra overheads or side-effects on the performance of the network I/O-bound process, a detailed network performance analysis was also conducted. The experimental setup to carry out the network analysis is similar to the one previously described in section
5.3.1. In this case, the value of the parameter `sysctl_sched_wakeup_granularity` is varied between 1 to 30 ms including the default value in Linux kernel 2.6.24, i.e. 10 ms. The results of the experiment are shown in Figure 12 and Figure 13.

![Figure 12. Network throughput with different sysctl_sched_wakeup_granularity](image1.png)

Figure 12 shows the network throughput for variable network loads for different values of the parameter `sysctl_sched_wakeup_granularity`. The results show a similar network throughput for all the different values of the parameter `sysctl_sched_wakeup_granularity`. This fact demonstrates that there is no effect on the throughput of the network process when the value of the wake-up granularity parameter ranges between 10 ms to 20 ms, and the Simplex process is not affected by the starvation problem.

![Figure 13. Network round trip delay with different sysctl_sched_wakeup_granularity](image2.png)

Figure 13.
Similarly, Figure 13 shows the network latency variation with variable network loads for different values of the parameter `sysctl_sched_wakeup_granularity`. There are some differences in the latency when the value of the parameter `sysctl_sched_wakeup_granularity` is set to 30 ms for traffic loads between 0 and 40Kpps. In all the cases, the differences are small around 5 ms. Nevertheless, all the graphs match showing almost no difference in the network latency beyond 40 Kpps. Moreover, the overall result shows no significant change in the network latency when the parameter `sysctl_sched_wakeup_granularity` is established between 10 to 20 ms, which in turn, is the suitable range for this parameter on which the Simplex process do not suffer from the starvation problem.

The performance analysis of the network process behavior shows that the proposed solution has no negative impact on the performance of the network process. As a result, it is recommended to change the default setting of the CFS scheduler parameter `sysctl_sched_wakeup_granularity` to 20 ms.

### 5.4 Solution Performance under SMP

The proposed solution on both Linux schedulers has been analyzed on a Symmetric Multi-Processor (SMP) main-board in order to demonstrate their effectiveness under SMP environments. An experimental setup similar to the one previously described in section 5.1, was used. However, in this case the Linux kernel was configured with the `SMP` options [30] enabled. Now, the receiver is a Pentium 3.2 GHz Core 2 Dual processor. Moreover, the sender generates network traffic at 90 Kpps, since it is the critical traffic rate at which the starvation problem arises. This particular value of the traffic rate at which the starvation problem takes place has been shown to be directly related to the sleeping behavior of the network processes at certain traffic rates, together with the default value of the `sysctl_sched_wakeup_granularity` parameter [1]. In particular, the default value (10 ms) of the `sysctl_sched_wakeup_granularity` scheduler parameters has been shown in [1] to be inappropriate, as it does not stop the network I/O-bound process from frequently preempting the CPU-bound processes.

The starvation problem might not appear on a dual processor machine in which only one CPU-bound process and one network I/O-bound process is running. This is because the scheduler usually runs the network I/O-bound process in one core and the CPU-bound process on the other. This can be easily detected using the following Linux command:

```bash
$ mpstat -P ALL
```

For this reason, this experiment runs two Simplex processes in the receiver to ensure that one core executes the one Simplex process, whereas the other core executes the second Simplex process and the network I/O-bound
process. The results provided in this section are those collected from the core managing both the CPU-bound Simplex process and the network I/O-bound process.

5.4.1 Linux 2.6 O(1) Scheduler

The global lower threshold has been varied from 0 to 1000 ms in order to analyze the effectiveness of the proposed solution in an SMP environment using the previously described experimental setup. The experimental results are shown in Figure 14.
Figure 14. Simplex execution under SMP environment with different MIN_SLEEP_TIME_MS thresholds

Figure 14(a) shows the starvation problem with the original Linux 2.6 O(1) scheduler configured with SMP options enabled. Note that the original Linux scheduler code is recovered by setting the parameter MIN_SLEEP_TIME_MS to 0 ms. The Simplex process needs almost 40 s to be executed against 1 s when the CPU is free. The Simplex execution time is drastically reduced from 38 s to 7 s when the MIN_SLEEP_TIME_MS is varied between 0 and 20 ms. After 20 ms, the graph shows that there is not significant improvement in the Simplex execution time when the MIN_SLEEP_TIME_MS is set to values that are higher than 20 ms. Figure 14(b), 12(c), 12(d) show the same trend. Moreover, Figure 14(e) shows how the amount of CPU time available for Simplex increases sharply when the MIN_SLEEP_TIME_MS ranges between 0 and 20 ms. However, the gain in CPU time decreases logarithmically beyond 20 ms.

In summary, the experimental results above demonstrate the presence of the starvation problem in Linux SMP environments running 2.6 O(1) scheduler. Moreover, those results show the similarity in the sleeping behavior of the network I/O-bound process under the symmetric multi-processor and uni-processor environments. This is clear from the sharp decrease in the graph shown in Figure 14(a) when MIN_SLEEP_TIME_MS ranges between 0 and 20 ms, which indicates that almost all of the sleep requests of the network I/O-bound process are shorter than 20 ms. Finally, the experimental results further demonstrate the effectiveness of the proposed solution on symmetric multi-processor architectures.

5.4.2 Linux CFS

The sysctl_sched_wakeup_granularity was varied from 0 to 100 ms in order to analyze the effectiveness of the proposed solution in an SMP environment using the previously described experimental setup. The experimental
results are shown in Figure 15. Note that the value of this `sysctl_sched_wakeup_granularity` parameter can be dynamically established using the following Linux command:

```
# sysctl -w kernel.sysctl_sched_wakeup_granularity=10000000
```
Figure 15. Simplex execution under SMP environment with different sysctl_sched_wakeup_granularity
Figure 15(a) shows the starvation problem when the original Linux CFS scheduler is configured with SMP options enabled. Note that the parameter `sysctl_sched_wakeup_granularity` was set to 10 ms on the original Linux CFS scheduler. However, the severity of the problem is lower than that obtained in the uni-processor environment, where the Simplex process needs 3.7 s when the parameter `sysctl_sched_wakeup_granularity` is set to 10 ms (default value) against the 109 s needed under the uni-processor environment. This can be attributed to the design improvement in the CFS scheduler fairness under SMP environment, which allows the scheduler to achieve better fairness on even a coarser granularity level, i.e. processor cores level, and thus reduces the severity of the starvation [31]. Nonetheless, the results for both SMP and uni-processor environments demonstrate a significant reduction in the Simplex execution time when the parameter `sysctl_sched_wakeup_granularity` ranges from 1 to 20 ms.

Figures 13(c) and 13(d) show the system time and the involuntary context switches of Simplex, respectively. They follow the same trend as in Figure 15(a). However, the user time shown in Figure 15(b) has a different behavior, which is almost a constant behavior around 1 s. Moreover, the CPU resources available to the Simplex process shown in Figure 15(e) follow a logarithmic trend. Note that the significant improvement in the Simplex performance is observed when the parameter `sysctl_sched_wakeup_granularity` ranges from 1 to 20 ms. On the other hand, parameter values greater than 30 ms can affect the performance of network I/O-bound process since Simplex unfairly receives more than 90% of the CPU resources.

The results of this experiment demonstrate the presence of the starvation problem in the Linux CFS scheduler configured with the SMP option enabled, although its severity is much less. To eliminate the starvation problem, it has been shown that the same recommended value of the parameter `sysctl_sched_wakeup_granularity` in uni-processor environments, i.e. 20 ms, should be also used to avoid the starvation problem under SMP environments. In fact, those results demonstrate the effectiveness of our proposed solution for both uni-processor and symmetric multi-processor environments.

6. Further Tuning and Enhancements

This paper has proposed effective solutions to address the starvation problem observed and has empirically demonstrated their effectiveness. However, further enhancement and tuning may be achieved under particular conditions. The following subsections describe some possible improvements that can be introduced to our proposed solutions for both Linux schedulers.
6.1.1 Linux 2.6 O(1) Scheduler

Our solution proposed using a *global lower threshold* has some performance issues. i) The processes available on a system have different behaviors. For this reason, the management of all these processes using a single unique threshold would definitely affect the performance of some processes. In fact, Wu and Crawford [11] implemented and analyzed a *global threshold* solution and they clearly noted that this solution might significantly affect certain network applications, such as multimedia network applications (e. g. VOIP applications), which usually send and receive packets periodically. ii) The sleep time of the network processes depends on the type of the network under which the process is running: local-area networks or wide-area networks [11]. For this reason, the global threshold must be updated according to the type of the network, and therefore, the end user has to be aware of all these complications related to the appropriate value of the global threshold. Note that the *global threshold* solution ignores the different network conditions that a networked system may experience in practice. iii) The observed starvation problem is only present at a specific network traffic load rate, i.e. between 70 and 100 Kpps and this fact is simply ignored in the *global threshold* solution.

An improvement to the proposed solution may be a *Local Per-Process Lower Threshold* solution. This solution is based on the definition of a *threshold* for each process, such that it efficiently accommodates all different types of network processes under different network conditions. This threshold can be determined using a heuristic approach to select the best value for the lower threshold associated with every non-interactive network process, while analyzing the behavior of the network process at different traffic rates. In summary, this solution may be designed as follows:

1. Define a $MIN\_SLEEP\_TIME\_MS$ threshold for each non-interactive network I/O-bound process. Those processes can be identified using their original priority and interactivity classification, as per the original O(1) scheduling algorithm [16]. Thus, the initial priority of the process would determine if it is interactive or not. The default threshold value can be initialized to 0 in order to be disabled by default.

2. For every non-interactive network I/O-bound process, determine the sleeping behavior of the process at different network loads. In particular, collect the number of sleeps and the sleep durations of the network process, and relate this information to the network traffic rate.

3. The collected information can be used to determine the smallest value of the $MIN\_SLEEP\_TIME\_MS$ threshold. This value can be used to establish a value which covers 80% of the sleeps requests. Once the value is known, the starvation problem that may arise would be avoided.
Finally, the value of the `MIN_SLEEP_TIME_MS` can be updated dynamically according to the new information collected about the sleeping behavior of the network process, taking into account network applications with special requirements, as well as changes in the network conditions.

### 6.1.2 Linux CFS

The proposed solution to the starvation problem under CFS Scheduler has similar limitations to the solution proposed under the 2.6 O(1) scheduler. In particular, the limitations can be briefed in the following points: i) A global parameter (i.e. `sysctl_shced_wakeup_granularity`) is not suitable to control all heterogeneous processes running on the system, even processes scheduled under the same scheduling policy. ii) Several important factors in shaping the sleeping behavior of the network process, such as the network environment and real-time network load, are not taken into account in the proposed solution. iii) The specific network traffic load rate under which the starvation problem is present, i.e. between 70 and 100 Kpps, is not taken into account in the proposed solution. iv) Finally, the proposed solution requires the end user to be aware of the proper configuration of the parameter `sysctl_sched_wakeup_granularity`.

A further improvement to our proposed solution may be through the use of a `Local Wake-up Granularity` parameter. This solution is based on the local definition of the parameter `sysctl_sched_wakeup_granularity` related to each process, such that it effectively takes into account all the different types of network processes under various network conditions. Moreover, the end user does not need be directly involved in setting up the proper value of this parameter, since the value of this parameter can be automatically determined using a proper heuristic algorithm and be dynamically updated at run-time based on the network conditions.

Note that the global parameter `sysctl_sched_wakeup_granularity` is used to delay the preemption of processes, in order to adapt to workloads and avoid over-scheduling. For this reason, the algorithm that can be used to automatically determine the value associated with the per-process parameter `local_wakeup_granularity` should be based on the sleeping behavior of the local process. In particular, we propose the formula (3) for calculating the per-process `local_wakeup_granularity` parameter:

\[
local\_wakeup\_granularity_p = \frac{\alpha \times num\_sleeps_p}{sched\_period \times load\_weight_p}
\]

where `local_wakeup_granularity_p` is the wakeup granularity parameter of the process `p`, `num\_sleeps_p` is the number of sleeps of the process `p`, `sched\_period` is the default scheduling period, `load\_weight_p` is the priority associated to the process `p`, and `\alpha` is a positive constant used as a weight factor to determine the importance of the sleep requests in the calculation of this parameter.
The following algorithm describes a possible implementation of the Local Wake-up Granularity improvement on a per-process basis. The solution tracks the number of sleeps for every scheduling period of all the processes. The number of sleeps is directly proportional to the value of the parameter local_wakeup_granularity. This ensures that processes which tend to sleep many times in very short periods, i.e. the scheduling period, are being penalized gradually for their sleeping behavior. In addition, the process priority is taken into account to efficiently manage high-priority processes.

**Algorithm: Per-process sched_wakeup_granularity**

- **Input:** p : process for which the wake up granularity is calculated  
  num_sleeps[p]: number of sleeps associated to the process  
  sched_period: the scheduling period time during which CFS tries to run each task once  
  load.weight[p]: the priority (nice value) associated to the process p  
  α: a positive constant

- **Output:** sched_wakeup_granularity[p]: wake up granularity for the process p

\[
\text{num_sleeps}[p] \leftarrow 0
\]

1. When p wakes up from a sleep  
   \[
   \text{num_sleeps}[p] = \text{num_sleeps}[p] + 1
   \]
   calculate the new value of sched_wakeup_granularity[p] using the formula (3)

2. For every new sched_period  
   \[
   \text{num_sleeps}[p] \leftarrow 0
   \]

7. **Conclusions**

We have studied and investigated the root causes responsible for the starvation of CPU-bound processes under the Linux kernel in the presence of network I/O-bound processes. We have analyzed the starvation problem for both on both Linux schedulers: 2.6 \(O(1)\) and CFS. We have proposed and implemented mitigation solutions for both Linux schedulers. Our mitigation involves the incorporation of a Global Lower Threshold in the Linux 2.6 \(O(1)\) Scheduler and the adjustment of the Wake-up Granularity parameters in the Linux CFS Scheduler. Our experimental results and analysis have shown that the proposed solutions are effective since they eliminate the starvation problem. At the same time, our proposed solutions do not impact negatively the performance of network I/O-bound processes. Our proposed solutions have been successfully validated for two types of mainboard architectures: uni-processor and symmetric multi-processor. Lastly, we have discussed tuning approaches that can further improve our proposed solutions for both Linux schedulers.
8. Acknowledgements

We thank the anonymous reviewers for their comments and suggestions which helped us to improve the content and quality of this paper.

References
