On Linux Starvation of CPU-bound Processes in the Presence of Network I/O

1K. Salah
1Computer Engineering Department
Khalifa University of Science Technology and Research (KUSTAR)
Sharjah, UAE
Email: khaled.salah@kustar.ac.ae

2A. Manea
2Department of Information and Computer Science
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Email: manea@kfupm.edu.sa

3S. Zeadally
3Department of Computer Science and Information Technology
University of the District of Columbia
Washington DC, 20008
Email: szeadally@udc.edu

4,5Jose M. Alcaraz Calero
4Department of Communications and Information Engineering
University of Murcia
Murcia 30011, Spain
5Hewlett-Packard Labs
Cloud and Security Lab
Bristol, BS34 8QZ, UK
Email: jmalcaraz@um.es

Abstract

Process starvation is a critical and challenging design problem in operating systems. A slight starvation of processes can lead to undesirable response times. In this paper, we experimentally demonstrate that Linux can starve CPU-bound processes in the presence of network I/O-bound processes. Surprisingly, the starvation of CPU-bound processes can be encountered at only a particular range of traffic rates being received by network processes. Lower or higher traffic rates do not exhibit starvation. We have analyzed it under different network applications, system settings and network configurations. We show that such starvation may exist for the two Linux Scheduler, namely the 2.6 O(1) scheduler and the more recent 2.6 Completely Fair Scheduler (CFS). We instrumented and profiled the Linux kernel to investigate the underlying root causes of such starvation. In addition, we suggest possible mitigation solutions for both schedulers.

KEYWORDS: Operating System, Linux, Network I/O, CPU Scheduler, Performance, CPU-bound Processes

1 Introduction

Nowadays, many computer systems are using the Linux operating system around the world. Linux is being used in aerospace, military, government, surgery, and definitively, in many critical services. The Linux kernel is the core of this operating system. The kernel provides some basic services such as memory management, processes scheduling, input/output management, and others. Any problem in the kernel may drastically affect the proper execution of applications, which in turn, may impact services.

The design of operating systems entails many challenges, one of which is to ensure that processes do not encounter starvation while running. The starvation problem is defined as a system state where a process is perpetually denied to use system resources, and the program cannot finish its task without those resources. There are numerous causes for starvation. For example, starvation may arise when the operating system receives many interruption requests (Input/
Output (I/O) requests). In this case, the running processes may never receive Central Processing Unit (CPU) time since the kernel uses all the CPU time processing these interruptions. The process scheduler is a module of the kernel in charge of managing the CPU resources fairly among the different processes running in the system without affecting the inherent behavior of any process [1].

The main objective of this paper is to demonstrate that CPU starvation can occur in the current Linux kernel 2.6. Such starvation is exhibited by CPU-bound processes which are non-interactive processes that require intensive CPU computation. The problem particularly occurs when network I/O-bound processes run in the system at the same time. In contrast to CPU-bound processes, network-I/O processes are non-interactive processes that use network I/O intensively. Examples of CPU-bound processes may include password cracking, simulation, and mathematical computation such as that of PI and matrix manipulation. Examples of network I/O bound processes may include Web and FTP servers, online game servers and applications, and multimedia streaming servers and applications. In order to establish the scope of this problem, this paper takes into account many experimental conditions covering different types of network I/O-bound processes, network interface cards, single and multiple processor mainboard architectures, and two Linux schedulers in order to demonstrate the occurrence of the starvation problem under all these conditions. Moreover, an in-depth analysis of the measurement results have been done to identify the root causes of such starvation. Finally, proposed solutions that can mitigate such starvation are given for both Linux Schedulers, namely the 2.6 $O(1)$ scheduler and the more recent 2.6 Completely Fair Scheduler (CFS), henceforth referred as 2.6 CFS.

The rest of the paper is structured as follows. Section 2 describes related works on the starvation problem in the Linux kernel. The different experiments to detect the starvation problem in CPU-bound processes when there are network I/O-bound processes in the system are described in the Section 3. Section 4 describes an analysis of different Linux schedulers identifying the root causes of the starvation problem. Section 5 proposes some ideas in the design of the Linux schedulers in order to minimize the starvation problem. Finally, Section 6 makes some concluding remarks and presents future works.

2 Related Work

Several research works have focused on the study of the Linux kernels and the starvation problem. An in-depth documentation about how Linux 2.6 CFS and 2.6 $O(1)$ schedulers work is given in [2],[3]. The author points out the importance of treating tasks with a certain degree of fairness and to ensure that threads never starve in the scheduler.

One of the major changes in the Linux kernel 2.6 was an entirely new scheduling algorithm. This scheduling algorithm was advertised for its $O(1)$ time complexity, and thus it was widely called Linux 2.6 $O(1)$ Scheduler. This scheduler gained wide acceptance among the Linux community over the years. This scheduler was designed to achieve the following new features: i) $O(1)$ complexity on the time needed to choose the next process to run. ii)
Quick response to interactive processes under high system load. iii) An acceptable level of prevention of both starving and hogging. iv) Scalability and task affinity under symmetric multiprocessing environment. v) Improved performance when having a small number of processes [4], [5].

*Linux 2.6 O(1) scheduler* has recently been replaced in the *Linux kernel 2.6.23* with a new scheduling algorithm, namely, the *Completely Fair Scheduler (CFS)*. The CFS was designed to maximize CPU utilization and the system interactive performance by fixing the deficiencies found in the previous O(1) scheduler, especially its complex interactivity heuristics. The basic idea behind the Linux CFS is to emulate an ideal and precise multitasking uniprocessor CPU, which is able to run all the processes in the system in parallel and with an equal speed of 1/n, where n is the number of runnable tasks in the system, by making an equal and fair division of the CPU bandwidth among all tasks in the system. One of the main features of CFS, besides its “fair scheduling” algorithm, is its modular design, where processes are scheduled according to their scheduling classes. Another important difference is that this scheduler has no notion of a specific pre-set timeslice per process. Rather, the timeslice is totally dynamic and its calculation is implicitly accomplished by the CFS algorithm [6].

Kang et al [7] describe why the *Linux 2.6 O(1) scheduler* is not suitable to support real-time tasks. The authors experimentally prove unexpected execution latency of real-time tasks which in many cases are due to starvation problems. They also provide a new Linux scheduling algorithm based on weighted average priority inheritance protocol (WAPIP), a variation of the priority inheritance protocol (PIP) [8] which assigns priorities to kernel-level processes at runtime by monitoring the activities of user-level real-time tasks. The new algorithm improves significantly the latency of the real-time tasks. Li et al [9] also provide a non-preemptive algorithm suitable for soft real-time systems, now based on the usage of dynamic grouping of tasks with deadlines that are very close to each other and schedules tasks within the group.

Torrey et al [4] propose a new Linux scheduler and they compare the new proposal with the current *Linux 2.6 O(1) scheduler* providing comparable results in all response time tests and showing some inadvertent improvements in turnaround time. An advantage of this new scheduler is that there is no method for differentiating interactive tasks, which minimizes the overheads associated with the scheduler. Bozyigit et al [10] provided a modified version of the Linux kernel (including the scheduler) which enabling an integrated task migration for clusters of workstations.

Wong et al [11] provide a comparison of 2.6 O(1) and CFS Linux schedulers noting that CFS is fairer in CPU time distribution without compromising the performance for interactive processes significantly higher than O(1). Moreover, authors prove that CFS is more efficient than O(1) mainly due to the complex algorithms used to identify interactive tasks for the *O(1) scheduler*. Wang et al [12] also provides a comparison of the Linux kernels: 2.6 O(1), CFS and Rotating Staircase Deadline Scheduler (RSDL) analyzing and measuring fairness, interactivity and multi-
processors performance in micro, real and synthesis applications. Authors prove that there are notable differences in fairness and interactivity under micro benchmarks, while minor differences in synthesis and real applications.

Salah et al [13][14] provide an interruption handling scheme for the Linux kernel which minimizes the overheads caused by heavy incoming network traffic. Their past research efforts focused on avoiding the starvation of user processes as a result of the time spent to handle incoming interrupt requests. Moreover, they prove experimentally that their proposal improves significantly the results for general-purpose network desktops or servers running network I/O bound applications, when subjecting such network hosts to both light and heavy traffic loads [15].

Wu and Crawford [16] prove that the Linux 2.6 O(1) scheduler can starve processes due to a misclassification of non-interactive network applications as interactive tasks that can lead to them unjustifiably obtaining up to 95% of the CPU resources. The authors identify and analyze the starvation problem in the Linux 2.6 O(1) scheduler running Linux Operating System (OS) in a single-processor mainboard. The authors address the starvation problem under different network traffic processing rates. They provide a generic solution for the problem. This solution is a global minimum threshold to filter out all processes in the system sending them to sleep.

In contrast to Wu and Crawford [16] work, our research work presented in this paper studies the behavior of the network processes at relatively normal network traffic loads, not exceeding 100 Mbps. As a result, our research work provides new results delimiting the appearance of the starvation phenomenon only at a particular traffic rate range near to 90 Kpps (packet per second). Moreover, this paper extends the Wu and Crawford’s work by identifying empirically the scope of the starvation problem. To this end, several empirical experiments with different network I/O-bound processes, network interface cards and Linux kernels have been analyzed under both single and multiple processor mainboard architectures. Moreover, the root causes of starved CPU-bound processes are identified and analyzed. Finally, mitigation solutions for that can address this starvation phenomenon are presented.

3 Existence of the Starvation Problem under Different System and Network Configurations

This section proves experimentally the appearance of the starvation problem in CPU-bound processes when there are network I/O-bound processes running the Linux kernel 2.6 and it establishes the scope of the problem. Section 3.1 describes the experiments used to investigate the starvation problem and the metrics used to evaluate the starvation. Sections 3.3-3.6 present empirical results under different experimental conditions, using different network I/O-bound processes, network interface cards, schedulers, and micro-processor architectures.

3.1 Experimental Setup

The experimental setup used to identify the starvation problem is composed of two machines: a sender and a receiver connected by means of a 1 Gbps Ethernet crossover cable, as shown in Figure 1. Both machines run Fedora Core 5 Linux with the Linux kernel 2.6.22 which is the latest version of Linux with O(1) scheduler. The sender is an Intel
Xeon CPU (5 GHz) with 4GB RAM and a network adapter with BCM 5751 net controller. The receiver is a single-
processor Intel Pentium 4 CPU (3.2 GHz) with 512 MB RAM and a network adapter with BCM 5752 network
controller. Both machines have been booted at run level 3, and no services are running in the background in order to
minimize the impact of other system activities on performance and measurement. Moreover, the Ethernet link flow
control has been disabled ensuring that the network throughput is not throttled between the sender and the receiver.

![Figure 1. Experimental setup](image)

The receiver runs the *Simplex* [17] application to measure the CPU time available to user applications. *Simplex* is an
implementation of a commonly used nonlinear numerical method for optimizing multi-dimensional unconstrained
problems by means of an iterative search algorithm. *Simplex* is a CPU-bound application computationally heavy
with no disk or network I/O operations. A parameter can be provided to *Simplex* establishing 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*. The
more the CPU is busy, the larger is this execution time interval. Moreover, the receiver also runs a network traffic
analysis tool in order to run a network I/O-bound process at the same time. This tool might process the entire
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. Different combinations of network traffic analysis and generation tools
have been analyzed in the following subsections. For this experimental setup, 64 bit payload length UDP packets
ranging from 0 to 150 Kpps have been used to analyze the behavior of the receiver against different net traffic rates.
We have chosen 64-bit packets to generate the maximum packet rate of 150 Kpps. A packet of 64 bits would give
the smallest Ethernet frame.

There are several measures 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 [18] for *Simplex* 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 achieved. 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 slot expiration in the scheduler or due to the presence of a higher priority task than *Simplex* in the scheduler.

### 3.2 Starvation with Different Network I/O-bound Processes

The goal of the experimental setup is to identify the starvation problem on CPU-bound processes in the presence of network I/O-bound processes. For this reason, the studies of different network I/O-bound processes become important to establish the scope of this problem. Several tool combinations have been utilized on both sender and receiver in order to identify the starvation problem under different experimental conditions.

The Linux scheduler used to carry out this experimental study has been the 2.6 O(1). The starvation problem was originally found in the 2.6 O(1). This scheduler is the one on which most interactivity research work has been focused on in the literature. In addition, the analysis of starvation has also been extended to the CFS scheduler.

The first combination is composed of the open source Distributed Internet Traffic Generator (D-ITG) [19] tool version 2.4.4 that generates traffic at the sender and the *ITGRecv* tool is also distributed in the same package for capturing this traffic in the receiver. The second and third combinations use *tcpdump* utility [20] and *ethereal* tool [21] as network I/O-bound processes respectively for capturing traffic in the receiver. In both cases, the KUTE [22] tool version 2.4 is used to generate the traffic in the sender. KUTE is a free traffic generation tool distributed as Linux kernel module that is able to send UDP packets at kernel-level mode. The main advantage of KUTE is that it does not suffer from the frequent context switching usually associated with user-level traffic generation tools and it enables us to increase the accuracy of measurements for departure times of packets. The second combination uses *Tcpdump 4.0* to simulate the network receiving activity. The third combination uses *Ethereal 0.99* to simulate the network receiving activity. Both have been executed to capture 2 GBit of traffic in 2 files (1 Gbit on each file). The results gathered when running *Simplex* CPU-bound process using these three different network I/O-bound process combinations are depicted in Figure 2.
Figure 2. CPU-bound process starvation with different network I/O processes

Figure 2 shows the performance of the Simplex process under three types of network I/O-bound processes, ITGRecv, tcpdump and ethereal. In general, the results show a common trend in all the network processes analyzed on which starvation problem arises at around 85-90 Kpps arrival rate. Figure 2(a) plots the Simplex execution time against the network traffic arrival rate. The Simplex execution time is increasing until a local maximum before 50 Kpps rate and then decreases. This local maximum may be due to an increased number of interrupts at this rate. After that point, the Simplex execution time increases sharply to an absolute maximum to a rate of 90 Kpps. In this case, the Simplex execution time takes more than 700 seconds to be executed when this process would not take more than 10 seconds on a free CPU (see Simplex execution time at 0 Kpps rate). The absolute maximum for all graphs indicates an apparent performance bottleneck at the rate of 90 Kpps for all the network I/O-bound processes analyzed. After 90 Kpps rate, Simplex execution time drops off and follows a regular linear trend whilst the network traffic is increased.

Figure 2(b) shows the Simplex user time against network traffic arrival rate. This plot follows a constant trend around 10 seconds. This is the time required for the execution of the Simplex process. The graph shows two maxima around 40 Kpps and 90 Kpps. This fact may be due to the increased number of interrupts at these rates. Note that the difference between these two maximum points and the rest of the points is small. The starvation problem is clearly shown at the rate of 90 Kpps, and the Simplex user time is almost constant for the remaining cases.
The *Simplex* kernel time is plotted against the network traffic arrival rate in Figure 2(c). The graph has a similar shape to Figure 2(a). This graph shows the origin of the starvation problem. It really originated in the kernel and it is not directly associated to *Simplex*. Note that the kernel time includes scheduling, context switches, and other activities. Figure 2(c) shows a small jump in *Simplex* kernel time at the rate of 40 Kpps and a very sharp jump at the rate of 90 Kpps, which clearly highlights the performance bottleneck at this rate. From the other results obtained, the number of involuntary context switches of *Simplex* is depicted in Figure 2(d). This result is a clear evidence of the starvation problem at the rate of 90 Kpps. Note that at 90 Kpps rate, the number of involuntary context switches reaches up to 2.5 million. Figure 2(e) depicts the percentage of CPU time available for *Simplex* at different network traffic rates. At the rate of 40 Kpps, there is a local minimum in the graph where *Simplex* cannot get many CPU resources due to the increased number of interrupts which have more scheduling priority than *Simplex*. At the rate of 90 Kpps, the percentage of *Simplex* CPU time is an absolute minimum reaching close to 30%. This shows a clear starvation problem at this rate where the *Simplex* process is almost starved. After 90 Kpps, the percentage of CPU resources available for *Simplex* is almost constant (around 50%).

This similarity in the *Simplex* behavior with different network I/O-bound processes shows that the starvation problem does not depend on the type of the network I/O process since it occurs for different types of network I/O processes, including *ITGRecv*, *Tcpdump* and *Ethereal*. The starvation problem has been identified in CPU-bound processes when a network I/O-bound process is running and it is receiving packets at rates approaching 90 Kpps.

### 3.3 Starvation with Different Network Interface Cards

Another experiment has been reproduced with different NICs. The goal of this section is to demonstrate that the starvation problem does indeed exist regardless of the type of NIC being used. The NIC used in Section 3.2 was *Broadcom BCM NIC (NetXtreme BCM5752 Gigabit Ethernet Card)*. In this experiment we use instead the *Intel NIC (82557/8/9 [Ethernet Pro 100])* NIC with exactly the same experimental conditions previously exposed in Section 3.2. The combination of *D-ITG* tools is used on both sender and receiver (*ITGRecv*). This combination has produced the highest starvation problem in the previous experiment. For this reason, this tool combination has been used in this experiment. Moreover, the execution time of *Simplex* has been reduced to get shorter simulation times.
Figure 3 shows the results of running Simplex for 1 second where the network traffic receiving rate is ranging between 0 and 150 Kpps. In this case also, the results show a peak of Simplex execution time at 90 Kpps. Both the user time and the kernel time are higher around 90 Kpps rate. However, note that the peak of involuntary context switches at the rate of 90 Kpps is not very sharp and the CPU availability shows a constant trend around 50% for rates higher than 60 Kpps showing a clear minimum at the rate of 90 Kpps. These differences can be due to the behavior associated to the different NIC drivers. Nevertheless, the increase of the kernel time and the Simplex execution time at 90 Kpps rate can help to conclude that the starvation problem is still present at this rate.
independent of the type of NIC utilized. It is worth noting that the *Simplex* execution time at 90 Kpps is 70 times more than the time required on a free CPU.

### 3.4 Starvation under a Different Linux Scheduler

To investigate whether the starvation problem is only present with the *Linux 2.6 O(1) scheduler* used in the previous experiments, we conducted a new experiment in which a different Linux scheduler is used. This experiment has been set up with the same experimental conditions described in Section 3.3 but now using the *Linux CFS* available in the kernel version 2.6.23 and thereafter. For our implementation, we used Linux version 2.6.30.

**Figure 4. Starvation phenomenon under Linux CFS**

![Graphs showing starvation phenomenon under Linux CFS](image-url)
All the plots available in Figure 4 show similar results to those obtained with the Linux 2.6 $O(1)$ Scheduler. In essence, the results show a peak in the Simplex execution time, user time, system time, and involuntary context switches around 90 Kpps rate. It is worth pointing out that the Simplex execution time is 120 times higher than the time Simplex requires on a free CPU. This fact demonstrates that this starvation problem is not an isolated fact associated to Linux 2.6 $O(1)$ scheduler but is available in both schedulers.

3.5 Starvation under Uni-Processor and Symmetric Multi-Processor Architectures

All the previous experiments have been executed on the Linux operating system on machines with a uni-processor (UP) architecture. We used the same experimental conditions described in Section 3.3 using now an SMP-configured Linux kernel to investigate the starvation problem under symmetric multi-processing (SMP) environment. In particular, we used two different SMP-configured Linux kernels: 2.6.22 and 2.6.30, which correspond to the implementation of the 2.6 $O(1)$ and the CFS schedulers. The execution time of the Simple application is shown in Figure 5. The results clearly show that the starvation problem still exists under both 2.6 $O(1)$ scheduler and CFS scheduler around the particular 90 Kpps network traffic receiving rate. This fact determines that the starvation problem is not specifically associated with UP architectures because it also arises with SMP-configured Linux kernels.

![Figure 5. Starvation Problem under SMP environment using (a) 2.6 $O(1)$ Scheduler (b) CFS](image)

4 Analysis of the Root Cause of the Starvation Problem

The experimental configurations described in Section 3 enable us to identify and investigate the scope of the starvation problem on CPU-bound processes in the presence of the network I/O-bound processes. As a result, it has been proven that the starvation problem arises on different experimental conditions. Note that the number of involuntary context switches that Simplex suffers under all these experimental conditions is a clear indication that the
starvation problem is directly related to the scheduling algorithm used in the Linux kernel. This section analyzes the root causes of the starvation problem. Moreover, since this problem appears in both schedulers: 2.6 \( O(1) \) and CFS, this section has been divided in two different subsections to analyze the root causes of both schedulers.

### 4.1 Linux 2.6 O(1) Scheduler

The Linux 2.6 \( O(1) \) scheduler is an implementation of a scheduling algorithm where the process execution priorities are calculated during the execution of the system [4, 5]. There is a computational trade-off on the inclusion of the dynamic priority calculation on the kernel design. On the one hand, the inclusion produces an overhead on the scheduling time since its value has to be dynamically calculated. On the other hand, the inclusion of the dynamic priority calculation in the Linux kernel allows changing the scheduling behavior adaptively based on the dynamic behavior of the process. It enables the optimization of the CPU usage when the processes suddenly change their behaviors at run-time. Linux 2.6 \( O(1) \) has included this dynamic priority feature to minimize the overhead required to execute this calculation. It uses a simple algorithm based on the calculation of delta priorities using the interactivity and the sleep times associated with the process. This dynamic priority calculation is done by means of the `recalc_task_prio()` function available in this scheduler. The intention of this analysis is to isolate the root cause of the starvation problem. So, different scheduling parameters have been altered and different scheduler features have also been altered to compare the results.

The dynamic priority calculation has been smartly disabled to investigate if this is related to the starvation problem. Figure 6 depicts the flow chart of the processes done inside of the `recalc_task_prio()` function. The deactivation of the dynamic priority calculation is achieved by avoiding any update in the average sleep time value of any process, i.e. `sleep_avg` [23, 24]. The highlighted box available in Figure 6 represents the update function deactivated.

This new version of the 2.6 \( O(1) \) scheduler has been used to run a new experiment on which all the experimental conditions described in Section 3.3 have been reproduced. In summary, a combination of \( D-ITG \) tools has been used at both the sender and the receiver; the sender ranges from 0 to 150 Kpps rates of 64-bit payload length UDP packet. The receiver also executed Simplex with the same number of iterations (equivalent to 1 sec. on a free CPU).
Figure 7 confirms that the starvation problem of CPU-bound processes in the presence of network I/O-bound processes has disappeared totally when the dynamic priority calculation is disabled in the Linux 2.6 O(1) scheduler. This fact proves that the starvation problem is directly related to the dynamic priority calculation mechanism of the scheduler.

Figure 7(a) plots the Simplex execution time against the network traffic arrival rate. The graph shows an increasing linear trend. A small positive slope ranges from 0 to 50 Kpps whereas a sharp slope ranges between 60 and 90 Kpps. Note that the trend is always increasing and there is no sharp jumps in the trend. This clearly shows that by disabling the dynamic priority calculation mechanism the starvation problem disappears. The reader can compare the Simplex execution times between both scenarios: activating and deactivating the dynamic priority calculation. For the case where the dynamic priority calculation is enabled, it is reached more than 700 seconds at the sharp peak between 70 and 100 Kpps (see figure 2(a)) whereas in the case where the dynamic priority calculation is disabled, it is reached only 1.5 second when the highest network load is being received in the system (150 Kpps). Thus, by disabling the dynamic priority calculation module in the scheduler, it has improved the scheduler behavior in our experiment.
Figure 7. Performance of Simplex when "dynamic priority calculation" is disabled

Figure 7(b) shows the involuntary context switches that Simplex does using our new version of the scheduler. In general, the involuntary context switches number remains below 50. This small number demonstrates that the starvation problem goes away when disabling the scheduler dynamic priority calculation module. Figure 7(c) depicts a constant linear trend for all the network traffic rates which ensures that there is no starvation in the system.

Figure 7(d) shows a decreasing linear trend in the amount of CPU resources available for the Simplex process. In a perfect agreement with Figure 7(a), the decrease is gradual between 0 and 50 Kpps, and then it sharply steepens between 60 and 90 Kpps. From 100 Kpps, the amount of CPU resources available for Simplex is kept almost constant at around 50 percent.

The dynamic priority calculation mechanism uses the number of voluntary times (invocation in the code to sleep/wait functions) that the processes perform as a metric to calculate their priorities. Simplex is a CPU-bound user application and it does not have any voluntary call to sleep functions in its code. For this reason, both versions of the Linux 2.6 O(1) scheduler may manage Simplex equally. To enable or disable the dynamic priority calculation controlling the average sleep time of each process on the system should not have any effect on the Simplex because its average sleep time is always zero. Then, the reason why both schedulers provide totally different performance.
results is because of the frequency of voluntary *sleeps* done by the network I/O-bound processes. When a new network packet may be processed, the network I/O-bound process tries to sleep waiting the packet delivery/reception by the hardware driver. The network I/O-bound process is managed differently when the *dynamic priority calculation* is disabled in the scheduler. A detailed analysis about how this process is influenced by the *dynamic priority calculation* might help to identify the root causes of the starvation problem in the 2.6 O(1) scheduler.

An instrumentation code is introduced into the *Linux kernel* to enable the tracing of the *sleep time* values of the process (*ITGRecv*) in order to better understand how the scheduler handles the *dynamic priority* of the network process. The instrumentation code has been introduced in the scheduler code without affecting its original control flow shown in Figure 6. In addition, it also monitored other scheduler values such as total number of sleeps, average sleep time value (*sleep_avg*), dynamic priority and interactivity status of the network process. As a result, Figure 8 shows the profiling of the *ITGRecv* network I/O-bound process when the *dynamic priority calculation* is enabled.

Figure 8. *ITGRecv sleep behavior analysis*

Figure 8(a) plots the average of every sleep time done by the network process against the network traffic receiving rate. The figure shows that the average of sleep time is about 4 ms for all traffic rates less than 85 Kpps. After 85 Kpps, the network process never goes to sleep because of the time to process a network packet is higher than the rate
at which packets are arriving. For this reason, the time of sleeps cannot be defined and it has been conveniently set zero for all these traffic ranges.

Figure 8(b) shows the average value of the calculated dynamic priority of the network process at different traffic rates. The priority values range from 115 (being the highest priority) to 125 (being the lowest priority). From very low traffic rates and up to 80 Kpps, the dynamic priority is the maximum possible value, i.e. 115. After 80 Kpps, the dynamic priority starts dropping to 118 at almost 87 Kpps. This drop in the dynamic priority of the network process is due to the decrease in the number of sleeps the process experiences at this high traffic rate (see Figure 8(a)). This fact would result in a decrease of the sleep_avg value, and in turn, its dynamic priority. This explains why Simplex performance is greatly improved beyond this particular network traffic rate. However, the dynamic priority does not drop to the minimum possible value (125). This can be understood from the analysis of the sleep_avg value of the network process presented in Figure 8(c).

Figure 8(c) plots the average value of the sleep_avg of the network process against the network traffic arrival rate. The maximum value of sleep_avg is defined by an internal constant in the kernel set by default at 1000 ms. This figure explains why the dynamic priority of the network process is boosted to the maximum possible value of 115 when the network traffic ranges from 0 Kpps to 80 Kpps. This is due to the large number of short network sleeps (around 4 ms) that the sleep_avg value accumulates, which in turn boosts its value to the maximum possible value of 1000 ms. When the number of sleeps is reduced beyond the 85 Kpps barrier, the value of sleep_avg drops to 700 ms level and stays stable there. The reason why sleep_avg does not drop further is mainly because of the way the scheduler handles interactive tasks. When a task is classified by the scheduler as an interactive task, the scheduler increases the task’s sleep_avg value, and for this reason, the task is assigned a higher priority. This helps interactive tasks to be less likely to lose their interactivity status when they are running. Thus, even though the number of sleeps is reduced, the sleep_avg is never below 700 ms, which is the threshold for interactive tasks with the default nice value. The task classification is done using heuristics based on the number of sleeps done by the process. Since the network process carries out many sleeps, it is wrongly classified as an interactive process. Moreover, we also investigated the interactivity status of the network process at all traffic rates and we found that the network process is indeed classified as an interactive process for all traffic rates, which match with the results in Figure 8(c). These results provide hints on the root causes of the starvation problem related to the wrong identification of interactive tasks which might produce higher priorities in non-interactive network processes.

Another analysis has been done to compare the behavior of the network I/O-bound processes whether there is a CPU-bound process or not. This experiment records the number of time slots the ITGRrecv uses to process the packets at different traffic rates whether the Simplex is running or not.
Figure 9. ITGRecv time slot consumptions and expirations

Figure 9 shows the number of time slots ITGRecv uses to process packets when Simplex is executed and not executed. In general, the higher the traffic rate, the higher is the number of time slots. Low traffic rates (0-40 Kpps) are linearly increasing the number of time slots. Medium-low traffic rates (40-60 Kpps) cause the network process to produce more packets in a single CPU burst. This reduces the overhead of interrupts and context switching. After that, medium-high traffic rates (60-85) are sharply increasing the time slots used by the network process until 85 Kpps where it reaches a maximum value around 300 slots without Simplex running and 150 slots with Simplex running. It is expected that the increasing traffic rate causes the network process to consume more time slots to process the packets. Beyond 85 Kpps, the number of time slots remains constant while traffic rate increases. This is actually the maximum throughput that the system can achieve.

Figure 9 shows clearly how CPU-bound processes can starve in the presence of network I/O-bound processes. Whether Simplex is running or not, there is no difference in the obtained results at lower traffic rates (0-85 Kpps). This is mainly because the network process monopolizes the CPU such that Simplex is only able to be run when the network process yields the CPU. When the network process consumes more time slots, and thus more CPU resources, the Simplex process starts to starve. Figure 9 illustrates the difference between the two cases when the network priority decreases beyond 85 Kpps). When Simplex is running, the network process time slots are 150. This is half of the number of time slots compared to when Simplex is not running (300). This clearly shows that at these traffic rates, the CPU resources are divided fairly between Simplex and the network process. This is why it is perceived a boost in the performance of Simplex beyond 85 Kpps. Note the worst case occurs at 85 Kpps rate. At this rate, the network process needs many time slots to process the packets and the process is never penalized for using all these time slots due to its misclassification as an interactive task. This allows the network process to monopolize the CPU resources and causes other processes in the system to starve. However, beyond 85 Kpps, the scheduler heuristics for detecting starvation are working fine and regardless of the interactivity status of the network process, it is being preempted whenever its time slot is totally consumed.
In summary, several root causes have been identified: Firstly, *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). Secondly, the reason for which this monopolization can be done is that the starvation heuristics [1,25] 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 act accordingly. Finally, 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 raise the starvation problem in other processes. Finally, all processes sleeping a small time receive the advantages associated to their sleep times unless other circumstances limit their evaluation of the sleep time [26, 27].

4.2 Linux CFS

The *Linux CFS* scheduler is an implementation of a scheduling algorithm which tries to emulate an ideal fair multitasking parallel processor dividing the processor time equally among all the processes in the runqueue [28, 29]. Note that the time is homogeneously divided in slots according to the number of processes. In fact, *Linux CFS* provides many tunable parameters allowing for an immediate real-time interaction with the system scheduler [6]. These tunable parameters can be the starting point to analyze the starvation problem of CPU-bound processes in the presence of network I/O-bound processes.

The first parameter to be analyzed is the *sysctl_sched_wakeup_granularity*. This parameter controls the wake up granularity of the processes that are under the *SCHED_OTHER* scheduling policy. The processes under this policy are the normal processes such as *Simplex* and *ITGRecv* (there is different scheduling policies for real-time, batch, etcetera). This parameter tries to adapt workloads in the processes to delay them to be preempted and reduces over-scheduling. This parameter is initialized in the Linux kernel 2.6.30 using the formula [30] given by equation (1):

$$\text{sysctl_sched_wakeup_granularity} = 10\text{ms} \times (1 + \log(ncpus))$$

where *ncpus* is the number of CPUs in the system. The default value of this parameter is 10 ms in single-processor systems for earlier Linux version up to 2.6.24. Note that the default value of this parameter has been changed in kernel 2.6.30 to 5 ms [31]. Periodically, the scheduler checks whether the current task may be preempted or not. This checking is done in the function *check_preempt_wakeup* [32] defined in the *sched_fair.c* file of the *CFS* scheduler module. Figure 10 shows a simplified version of the skeleton of this function.

```c
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;
```
After several checking for processes belonging to different scheduling policies including real-time, batch, and group scheduling, the check_preempt_wakeup function compares the values of vruntime of the current process against the new wake-up process. Then, in case the difference of these vruntime values is higher than the wake-up granularity threshold, i.e. sysctl_sched_wakeup_granularity parameters, the wake-up process preempts the current process. Note how this parameter is used to adapt workloads since it enables the usage of more time slots before preempting the process.

The experimental analysis of the sleep behavior of the network process ITGRecv done in the Linux 2.6 O(1) Scheduler (previously described in Section 4.1) has shown that this process sleeps frequently for very short durations. To increase the value of sysctl_sched_wakeup_granularity in the CFS scheduler may reduce the number of times ITGRecv can preempt Simplex, which in turn, can improve the performance of Simplex.

Another experimental setup has been done in order to discover how sysctl_sched_wakeup_granularity parameter can influence the starvation problem. The experimental conditions used are exactly the same described in Section 3.4. In this case, the sender is always sending traffic at 95 Kpps rate, the result achieved in this scheduler (as shown in Figure 4(a)) is worse and the value of the sysctl_sched_wakeup_granularity parameter is varied at the receiver while the performance of Simplex is recorded. The command used to vary the value of the parameter sysctl_sched_wakeup_granularity is as follows:

```
# sysctl -w kernel.sysctl_sched_wakeup_granularity="1000000"
```

This command assigns the value of 10 ms to the sysctl_sched_wakeup_granularity parameter. Note that this parameter is measured in nanoseconds [31]. The sysctl_sched_wakeup_granularity value is varied from 0.1 ms to 30 ms. Figure 11 shows the results of this experiment in which higher values of the sysctl_sched_wakeup_granularity parameter produce better Simplex execution times.
Figure 11. Effect of wakeup_granularity in (a) Simplex execution time and (b) Involuntary context switches

Figure 11(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 whilst the wake-up granularity parameter increases. Moreover, Figure 11(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 provide similar results. Three distinguishable regions become clear in Figures 11(a) and 11(b): 1-10 ms, 10-20 ms, and 20-30 ms. For simplicity, we refer to 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, Simplex execution time is only reduced 3s (from 112s at 1ms to 109s at 10ms). This fact can be directly mapped to the general sleeping behavior of the network process previously plotted in Figure 8(a), which shows that more than 80% of the sleeps are shorter than 10 ms. To better understand the relationship between the observed behavior and the network process sleep behavior, we first explain how CFS manages these processes. Initially, both Simplex and ITGRecv start together at the same time and they have the same values. The Linux CFS algorithm strives to give an equal share of CPU time to both processes. At the end of any scheduling period (sched_period), CFS might have provided the same CPU time to both processes (the CPU time consumed by a process is stored in its vruntime value). Therefore, at the start of the next scheduling period, both processes have almost equal values of vruntime (also referred to as x). We assume the network process starts its execution during a small time (referred as ε). Thus, its vruntime value becomes x + ε. Then, Simplex starts its execution whereas the network process is sleeping. Let us assume that the network process sleeps during a time referred to as y. When the network process wakes up, Simplex has been executing for almost y ms. Thus, its vruntime value becomes x + y. At this point, the Linux CFS has to decide whether the network process should preempt Simplex or not. This is done in the check_preempt_wakeup function according to the condition available in the formula labeled as (2):

\[ \text{vruntime}_{\text{Network}} + \text{sysctl_sched_wakeup_granularity} < \text{vruntime}_{\text{Simplex}} \]  \hspace{1cm} (2)

If the condition available in the formula (2) is fulfilled, Simplex is preempted by the network process. Otherwise, Simplex may continue to execute. Replacing the variables in the formula labeled as (2) with their values results in the formula labeled as (3):

\[ x + \varepsilon + \text{sysctl_sched_wakeup_granularity} < x + y \]  \hspace{1cm} (3)

Simplifying x in (3), and isolating the wake-up granularity value, the formula labeled as (4) is obtained:

\[ \text{sysctl_sched_wakeup_granularity} < y - \varepsilon \]  \hspace{1cm} (4)
Consequently, *Simplex* is preempted if the value of the wake-up granularity is less than the sleep time of the network process \( y \) minus a small value \( \varepsilon \) which in turn is the time the network process has been executed on the CPU. Since it is known that \( \varepsilon \) is always greater than zero, eliminating \( \varepsilon \) from \( (4) \) the equation has no effect. Thus, \( (4) \) can be simply written as the formula labeled as \( (5) \):

\[
\text{sysctl\_sched\_wakeup\_granularity} < y
\]

\( (5) \)

Then, since the network process sleeps more than 80% of the times less than 10 ms, *Simplex* is more susceptible to be preempted by the network process when the value of the parameter \( \text{sysctl\_sched\_wakeup\_granularity} \) is less than 10 ms. This explains why the performance of *Simplex* is not improved when the value of the \( \text{sysctl\_sched\_wakeup\_granularity} \) parameters ranges from 1 ms to 10 ms and this also explains the high rate of involuntary context switches.

In the case of the *Medium Wake-up Granularity Region*, the performance of *Simplex* improves. The *Simplex* execution time sharply drops from 109s at 10ms to 2.5s at 20ms. This result is because more than the 80% of the network process sleeps are shorter than 10 ms. Thus, since the \( \text{sysctl\_sched\_wakeup\_granularity} \) value is higher than 10 ms, *Simplex* is less susceptible to be preempted by other wake-up network processes and consequently, there is a decrease in the number of involuntary context switches. Note that the condition depicted in formula labeled as \( (6) \) is harder to be fulfilled when the \( \text{sysctl\_sched\_wakeup\_granularity} \) value increases. Therefore, *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 2.54s at 20 ms to 2.36s at 30 ms. This behavior is reasonable since almost all the overhead in *Simplex* was caused by the network process and it has already been eliminated when the value of the \( \text{sysctl\_sched\_wakeup\_granularity} \) parameter is 20 ms. In fact, only the 10% of the sleeps of the network process are longer than 15 ms. Thus, from 20 ms to 30 ms, this parameter does not really produce a significant impact on the *Simplex* execution time and on the number of involuntary context switches.

After the analysis above, we can conclude that the root causes for which the starvation problem arises in CPU-bound processes when there is a network I/O-bound process running in the system using the *Linux 2.6 CFS scheduler*:

First, the starvation problem is related to the default value (10 ms) associated to the \( \text{sysctl\_sched\_wakeup\_granularity} \) scheduler parameters. This default value is not an appropriate value as shown in the formula labeled as \( (6) \) and in the Figure 11. The reason is because this value does not stop the network I/O-bound process from frequently preempting the CPU-bound processes. Second, the result of this experiment demonstrates the relationship between the performance of the CPU-bound processes and the value of the \( \text{sysctl\_sched\_wakeup\_granularity} \) parameter. Finally, the starvation problem of CPU-bound processes is directly
related to the sleeping behavior of the network processes at certain traffic rates (95 Kpps in this experiment) together with the value of the `sysctl_sched_wakeup_granularity` parameter.

5 Solutions to the Starvation Problem

Once this starvation problem has been identified, scoped and analyzed, this section describes some ideas in the design of the schedulers to mitigate the starvation problem available in both schedulers analyzed: Linux 2.6 O(1) and 2.6 CFS. This section has been divided in subsections in order to describe the solutions related to each scheduler.

5.1 Linux 2.6 O(1) Scheduler

The analysis of the *sleeping* behavior of the `ITGRecv` network process carried out in Section 4.1 has shown that this process is misclassified as an interactive process and it also receives high values of *dynamic priority* which leads to the starvation of other processes. The main reason is because the scheduler design cannot recognize non-interactive processes with many *short-time* sleeps. A simple solution may be to insert a *global static threshold* in the scheduler in order to control the minimum *sleep_time* value to be taken into account in the *average sleep* time of any process, as Wu and Crawford [16] have already proposed. This *global static threshold* has been implemented using the `/proc` filesystem. The value of the threshold may be varied and analyzed in order to find the most suitable value that minimizes the starvation problem whereas the general performance of the scheduler is not altered. Although this is a simple solution, it also has an important drawback because the processes running in the system may have different characteristics and behaviors and a single unique threshold may affect the performance of specific processes. Indeed, Wu and Crawford [16] also noted that this solution might affect certain network applications, such as multimedia applications (e.g. Voice over Internet Protocol (VoIP) applications) which usually send and receive packets periodically. In addition, the differences in the network environment (Local Area Network (LAN) or Wide Area Network (WAN)) as well as the differences in the network load at different times are very significant factors in shaping the sleeping behavior of many network process. Moreover, the observed starvation problem is only present at a specific network traffic load ranging between 70 and 100 Kpps. All these factors are ignored in this solution.

Thus, another more effective solution is to insert a *Local Dynamic Threshold* associated to each process in the scheduler. This *Local Dynamic Threshold* is locally defined in each process with the intention of fulfilling the requirements of the different types of network processes under all the network conditions without any compromise. To this end, this threshold may be dynamically calculated according to the number and duration of *sleeps*, the network environment, the network load, and etcetera. This is a more flexible solution which could minimize starvation while improving the way the scheduler manages the network processes.

In addition, another solution is to improve the scheduler heuristics so as to be able to realize that there is a starvation problem under the circumstances analyzed in Section 4.1. Current heuristics are not able to detect the starvation problem when the traffic rate ranges between 70-100 Kpps.
Finally, another solution is to improve the scheduler heuristics to be able to classify correctly the interactivity of all the network processes. The correct classification may minimize the starvation problem because the non-interactive network processes receive a penalty in the number of time slots when they try to starve other processes.

5.2 Linux CFS

The analysis of the Linux CFS scheduler carried out in Section 4.2 has shown how the `sysctl_sched_wakeup_granularity` parameter is directly related to the starvation problem in CPU-bound processes when there is a network I/O-bound process running. The default value of this parameter causes the sleeping behavior associated with network processes to starve CPU-bound processes at traffic rates ranging between 70 and 100 Kpps.

A simple solution to the starvation problem associated with this scheduler could be achieved by modifying the global setting of the `sysctl_sched_wakeup_granularity` parameter. The analysis of the scheduler has shown that the most suitable value for this parameter to solve the starvation problem may follow the condition depicted in the formula (6): \( \text{sysctl_sched_wakeup_granularity} < y \), where \( y \) is the network process’s sleep time value. The \( y \) value is mostly between 5 ms and 15 ms when the network traffic rate is in the critical range 70 – 100 Kpps. It is worth noting that neither the old nor the new default setting of `sysctl_sched_wakeup_granularity` satisfies the formula labeled as (6).

To modify the default setting of `sysctl_sched_wakeup_granularity` parameter to be greater than most of the network process sleep time values when the network traffic rate is between 70 – 100 Kpps may solve the problem (> 15 ms). Moreover, the analysis carried out in Section 4.2 has shown that there is no gain in the performance when this value is higher than 20 ms. Thus, to modify the default setting of the parameter `sysctl_sched_wakeup_granularity` to 20 ms can avoid the starvation problem.

To use a global parameter to control the wake-up granularity produces some limitations. It is worth noting that even those processes belonging to the same scheduling queue are heterogeneous in their nature and behaviors. Therefore, a global unique value to control the `sysctl_sched_wakeup_granularity` might not suit the needs of all processes. For example, multimedia applications send/receive packets periodically producing a special sleeping behavior, which might be affected by the new value of `sysctl_sched_wakeup_granularity` parameters. Moreover, other factors such as the network environment (LAN or WAN) and the network load at different times are significant in the profiling of the sleeping behavior of many network processes but will be simply ignored if a unique global value is used.

Another solution is to insert a *Global Dynamic Wakeup-Granularity* in the scheduler. This parameter is automatically calculated by the scheduler taking into account the heterogeneity in the network processes and conditions. This is a value dynamically calculated when the network conditions change. The algorithm may take into account metrics such as the number of sleeps, average sleep time and network traffic rate. This approach hides the complexity of this parameter from the end-user since it is calculated without any user intervention. Moreover, the upper and lower thresholds may also be managed in order to avoid any further starvation problem.
6 Conclusions and Future Work

We have empirically proved that Linux can starve CPU-bound processes in the presence of network I/O-bound processes. We have shown that the starvation problem only arises when network processes send information at a particular range of traffic rates, while lower and higher traffic rates do not cause the starvation problem. The scope of the starvation problem has been analyzed under different network I/O-bound applications, system settings, and network configurations. Moreover, this scope has been analyzed on both single and multiple processor mainboard architectures. We have experimentally proved that the starvation problem exists in both Linux 2.6 schedulers: \textit{O(1)} and \textit{CFS} under all the experimental conditions analyzed. Moreover, the underlying root causes of the starvation problem in both schedulers have been identified and analyzed. Finally, solutions to mitigate the starvation problem have been proposed for both Linux schedulers. As a future study, we plan to implement the proposed solutions for both Linux schedulers and then evaluate their effectiveness in mitigating such a starvation phenomenon.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and feedback which enhanced the quality of the paper considerably.

References

A. He received the B.S. and M.S. degree in Computer Science from King Fahd University of Petroleum and Minerals, Saudi Arabia. His research interests include operating systems, parallel computing architectures, large-scale scientific computation and numerical analysis.

K. Salah is an associate professor at the Department of Computer Engineering, Khalifa University, Sharjah, UAE. He received the Ph.D. degree in Computer Science from Illinois Institute of Technology, USA, in 2000. His primary research interests are in the area of operating systems, computer networks, network security, performance evaluation and modeling.

A. Manea is currently a Ph.D. student in the Energy Resources Engineering Department at Stanford University, USA. He received the B.S. and M.S. degree in Computer Science from King Fahd University of Petroleum and Minerals, Saudi Arabia. His research interests include operating systems, parallel computing architectures, large-scale scientific computation and numerical analysis.
S. Zeadally received his Bachelor’s Degree in Computer Science from University of Cambridge, England, and the Doctoral Degree in Computer Science from University of Buckingham, England, in 1996. He is an Associate Professor at the University of the District of Columbia. He is a Fellow of the British Computer Society and a Fellow of the Institution of Engineering Technology, UK.

J. M. Alcaraz Calero holds a Ph.D. in computer Science by University of Murcia. Currently, he is a lecturer at University of Murcia, Spain and research staff at Hewlett-Packard Laboratories, UK. His research areas include cloud computing, virtualization, security, operating systems, policy-based systems and semantic web.