

Interactivity vs. fairness in networked Linux systems

Wenji Wu *, Matt Crawford

Fermilab, MS-368, Batavia, IL 60510, United States

Received 18 September 2006; received in revised form 6 February 2007; accepted 3 April 2007
Available online 3 May 2007

Responsible Editor: I. Nikolaidis

Abstract

In general, the Linux 2.6 scheduler can ensure fairness and provide excellent interactive performance at the same time. However, our experiments and mathematical analysis have shown that the current Linux interactivity mechanism tends to incorrectly categorize non-interactive network applications as interactive, which can lead to serious fairness or starvation issues. In the extreme, a single process can unjustifiably obtain up to 95% of the CPU! The root cause is due to the facts that: (1) network packets arrive at the receiver independently and discretely, and 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. (2) The current Linux interactivity mechanism provides the possibility that a non-interactive network process could receive a high CPU share, and at the same time be incorrectly categorized as interactive. In this paper, we propose and test a possible solution to address the interactivity vs. fairness problems. Experiment results have proved the effectiveness of the proposed solution.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Linux; Process scheduling; Interactivity; Fairness; Networking

1. Introduction

Over the last several years, the Linux operating system has gained wide acceptance and is deployed in many scientific and commercial environments. Compared to previous versions, Linux 2.6 has made significant performance improvements in terms of interactivity, fairness, and scalability. Linux 2.6 is now preemptible, and has an $O(1)$ CPU scheduler.

The Linux 2.6 scheduler is prioritized and epoch-based [1–4]. The whole process scheduling is based on a data structure called *runqueue*. A runqueue is created and maintained for each CPU in the system. The per-CPU runqueue keeps track of all runnable tasks assigned to a particular CPU. Each runqueue consists of an active priority array and an expired priority array. All runnable processes begin in the active array, and are scheduled in priority order. In general, when a process expires it is moved to the expired array so that all runnable processes get an opportunity to execute. When the active array becomes empty, the expired and active arrays are switched. This unique active–expired array design

* Corresponding author. Tel.: +1 630 840 4541; fax: +1 630 840 8208.

E-mail addresses: wenji@fnal.gov (W. Wu), crawdad@fnal.gov (M. Crawford).

is credited with much of the overall system performance improvements.

One design goal of Linux 2.6 is to improve interactivity [5]. Processes such as text editors and command shells interact constantly with their users, and spend a lot of time waiting for keystrokes and mouse events. When inputs are received the process must be woken up quickly; otherwise, the user will find the system to be unresponsive and annoying. Typically, the delay must not exceed 150 ms [1]. Linux 2.6 provides excellent interactive performance by employing the following measures [1,2,4]: (1) its scheduler is a typical decay usage priority scheduler. Processes are scheduled in priority order, where effective priority has two components: static priority and dynamic priority bonus. The static priority reflects inherent relative importance of processes, which is expressed by processes' *nice* values. The dynamic priority bonus depends on CPU usage patterns; the scheduler favors interactive processes and penalizes non-interactive processes by adjusting the dynamic priority bonus. (2) To reduce scheduling latency, expired interactive processes are reinserted back into the active array, instead of the expired array. In addition, an interactive process' timeslice is divided into smaller pieces, preventing interactive processes from blocking each other. (3) Linux 2.6 is kernel-preemptible. Whenever a scheduler clock tick or interrupt occurs, if a higher-priority task has become runnable, it will preempt the running task as long as the latter holds no kernel locks. (4) Linux 2.6's clock granularity has reached 1 ms level.

Fairness is another design goal of Linux 2.6 [5]. Fairness is the ability of all tasks not only to make forward progress, but to do so relatively evenly. The opposite of fairness is starvation, which occurs if some tasks make no forward progress at all [6,7]. Linux 2.6 scheduler's active-expired array design is supposed to ensure fairness [1,2]. However, as described above, an expired interactive process is reinserted back into the active array instead of the expired array. This leads to the possibility of starvation for the processes in the expired array if the active array continues to hold runnable processes. To circumvent the starvation issue, when the first expired process is older than some limit, expired processes are moved to the expired array without regard to their interactive status. Usually, an interactive process does not consume much CPU time because most of time it sleeps waiting for user inputs. In general, the Linux 2.6 scheduler can ensure fairness among processes, and provide excel-

lent interactive performance at the same time. However, our experiments and analysis have shown that the current Linux interactivity mechanism tends to incorrectly categorize non-interactive network applications as interactive, which can lead to serious fairness or starvation issues. The interactivity mechanism allows the possibility that a non-interactive network process could consume a large CPU share, and at the same time be incorrectly categorized as interactive. Further, incorrectly labeled "interactive network applications" might block true interactive applications, resulting in degraded interactive performance.

Linux-based network end systems have been widely deployed in the High-Energy Physics (HEP) community at labs like CERN, DESY, Fermilab, and SLAC, and at many universities. At Fermilab, thousands of networked systems run Linux; these include computational farms, trigger processing farms, hierarchical storage servers, and desktop workstations. From a network performance perspective, Linux represents an opportunity since it is amenable to optimization and tuning due to its open source support and projects such as web100 and net100 that enable examination of internal states [8,9]. The performance of Linux-based network end systems is of great interest to HEP and other scientific and commercial communities. In this paper, we analyze the interactivity vs. fairness issues in networked Linux systems. Our analysis is based on Linux kernel 2.6.14. Also, it is assumed that the NIC (Network Interface Card) driver makes use of Linux's "New API," or NAPI [10,11], which reduces the interrupt load on the CPUs. The contributions of the paper are as follows: (1) We systematically study and analyze the Linux 2.6 scheduling and interactivity mechanism; (2) Our researches have pointed out that the current Linux interactivity mechanism is not effective in distinguishing non-interactive network processes from interactive network processes, and might result in serious fairness/starvation problems. Mathematical analysis and experiments results have verified our conclusions. (3) Further, we propose and test a possible solution to address the interactivity vs. fairness problems. Experiment results have proved the effectiveness of our proposed solution.

The remainder of the paper is organized as follows: In Section 2 the related researches on interactivity and fairness are presented. Section 3 analyzes Linux scheduling and interactivity mechanisms. In Section 4, we investigate the interactivity vs. fairness

problems in networked Linux systems through mathematical analysis. In Section 5, we show experiment results to further study the problems, verifying our conclusions in Section 4. In Section 6, we propose and test a possible solution to address the interactivity vs. fairness problems in network Linux systems. And finally in Section 7, we conclude the paper.

2. Related work

The schedulers of Unix variants such as BSD4.3, FreeBSD, Solaris, and SVR4 [12–15] are typical decay usage priority schedulers: processes are scheduled in priority order; higher priority processes are scheduled to run first. The priorities of I/O bound (interactive) processes grow with time, so that when they are awakened, they have higher priority than CPU-bound (non-interactive) processes, and are therefore scheduled to run immediately. In general, those schedulers provide excellent interactive response on general-purpose time-sharing systems for traditional interactive applications that have low CPU consumption. However, those schedulers are not effective in support of interactive multimedia applications (e.g., audio player, video player) that have high CPU usages. To address this problem, Etsion et al. [16] proposed the human-centered scheduling of interactive and multimedia applications on a loaded desktop. In their approach, the scheduler first estimates the “volume of user-interaction” associated with each process by monitoring relevant I/O device activity, and then the scheduler uses those estimates to prioritize interactive processes, without respect to their CPU usages. However, this method might not be appropriate for some network applications.

To ensure fairness, proportional-share schedulers [17–20] are usually employed to control the relative rates at which different processes can use the processor. Over the years, different proportional-share schedulers have been proposed. In [17], Waldspurger et al. proposed the lottery scheduling to enable flexible control over the relative rates at which CPU-bound workloads consume processor time. In [18], Goyal et al. proposed a hierarchical CPU scheduler for multimedia operating systems, which provides protection between various classes of applications.

In [21], Petrou et al. proposed a hybrid lottery scheduler, which aims to achieve responsiveness comparable to the FreeBSD scheduler while maintaining lottery scheduling’s flexible control over relative execution rates and load insulation. So far, no

research has been found to relate interactivity and fairness to network applications.

3. Linux scheduling and interactivity

Linux 2.6 is a *preemptive multi-processing* operating system. Processes (tasks) are scheduled to run in a *prioritized round robin* manner [1–4], to achieve the objectives of fairness, interactivity and efficiency. For the sake of scheduling, a Linux process has a dynamic priority and a static priority. A process’ static priority is equivalent to its *nice* value, which is specified by the user and not changed by the kernel. The dynamic priority is used by the scheduler to rate the process with respect to the other processes in the system. An eligible process with better (smaller-valued) dynamic priority is scheduled to run before a process with a worse (higher-valued) dynamic priority. The dynamic priority varies during a process’ life. It depends on the process’ scheduling history and its specified static priority, which we will elaborate in the following sections. There are 140 possible priority levels for processes (both dynamic priority and static priority) in Linux. The top 100 levels are used only for real-time processes, which we do not address in this paper. The last 40 levels are used for conventional processes.

3.1. Linux scheduler

As shown in Fig. 1, the whole process scheduling is based on a data structure called runqueue. Essen-

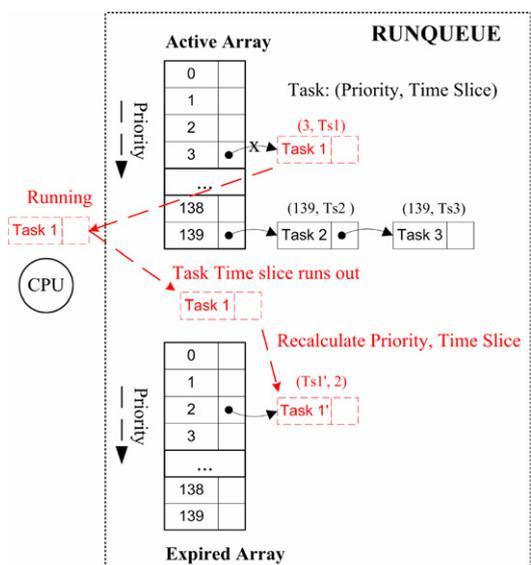


Fig. 1. Linux process scheduling.

tially, a runqueue keeps track of all runnable tasks assigned to a particular CPU. One runqueue is created and maintained for each CPU in a system. Each runqueue contains two priority arrays: *active priority array* and *expired priority array*. Each priority array contains a queue of runnable processes per priority level. Processes with higher dynamic priority are scheduled to run first. Within a given priority, processes are scheduled round robin. All tasks on a CPU begin in the active priority array. Each process' *timeslice* is calculated based on its *static priority*; when a process in the active priority array uses up its timeslice, it is considered *expired*. An expired process is moved to the expired priority array if it is not interactive. An expired interactive process is reinserted into the active array if possible. In either case, a new timeslice and priority are calculated. When there are no more runnable tasks in the active priority array, it is simply swapped with the expired priority array. An unexpired process might be put into a wait queue to sleep, waiting for expected events such as completion of I/O. When a sleeping process wakes up, its timeslice and priority are recalculated and it is moved to the active priority array. As for preemption, whenever a scheduler clock tick or interrupt occurs, if a higher-priority task has become runnable, it will preempt the running task as long as the latter holds no kernel locks.

3.2. Interactive scheduling

As we have said above, an interactive process needs to be responsive. The Linux kernel must provide the capabilities of interactive scheduling. To this end, it needs to:

- Perform process classification: differentiate interactive processes from non-interactive processes.
- Try to minimize the scheduling latency [7] for interactive processes.
 - Prevent non-interactive processes from blocking interactive processes.
 - Prevent interactive processes from blocking other interactive processes.

The interactivity estimator is designed to find which processes are interactive and which are not. It is based on the premise that non-interactive processes tend to use up all the CPU time offered to them, whereas interactive processes often sleep [1].

A *sleep_avg* is stored for each process: a process is credited for its sleep time and penalized for its runtime. A process with high *sleep_avg* is considered interactive, and low *sleep_avg* is non-interactive. The interactive estimator framework embedded into Linux operates automatically and transparently.

A process' dynamic priority varies during the process' life span. It depends on the process' interactivity status and its specified static priority. Linux assigns a dynamic priority to process P at time t as follows:

$$\text{dynamic_priority}(P, t) = \max\{100, \min\{\text{static_priority}(P) + 5 - \text{bonus}(P, t), 139\}\} \quad (1)$$

$$\begin{aligned} \text{bonus}(P, t) &= P \rightarrow \text{sleep_avg}(t) \\ &* \text{MAX_BONUS}/\text{MAX_SLEEP_AVG}. \quad (2) \end{aligned}$$

The constant *MAX_BONUS* is 10 and *MAX_SLEEP_AVG* is 1000 ms. $P \rightarrow \text{sleep_avg}(t)$ is the *sleep_avg* (in ms) for process P at time t , and it is limited to the range $0 \leq P \rightarrow \text{sleep_avg}(t) \leq \text{MAX_SLEEP_AVG}$. Therefore, *bonus*(P, t) ranges from 0 to 10. The quantity $5 - \text{bonus}(P, t)$ is also called the *dynamic priority bonus*. The more time a process spends sleeping, the higher the *sleep_avg* is, and the higher the priority boost.

From (1) and (2), it can be seen that Linux credits interactive processes and penalizes non-interactive processes by adjusting dynamic priority bonus. In this way, Linux allows interactive processes to preempt non-interactive processes when they have same, or nearly the same, static priorities.

When a process runs out its timeslice, the Linux kernel needs to determine its interactivity status. An expired interactive process is reinserted back into the active array, instead of the expired array. The interactivity threshold condition for process P is

$$\text{bonus}(P, t_E) \geq \text{static_priority}(P)/4 - 23, \quad (3)$$

where t_E is the moment that process P expires. For a process P with a default nice value of 0, the static priority is 120 [1,4] and the interactivity threshold is equivalent to $P \rightarrow \text{sleep_avg}(t_E) \geq 700$ ms.

If and only if the condition in (3) holds, P is deemed interactive. Reinserting an interactive process into the active array helps to increase responsiveness. If it was not done in this way, an interactive process in the expired array would have to wait for all the runnable processes in the active array to finish before regaining the CPU. However, keeping an expired interactive process in the active

array might lead to starvation for the processes in the expired array as long as the active array continues to hold runnable processes. To circumvent starvation, special interactivity rules have been made:

- **Rule 1:** If the time since the first process in the active array expired is greater than or equal to $STARVATION_LIMIT \times NR_{\text{running}} + 1$, any expired processes are moved to the expired array without regard to their interactive status. Here, the constant $STARVATION_LIMIT$ is 1000 ms, and NR_{running} is the number of processes in the runqueue.
- **Rule 2:** The interactivity is also ignored if a process in the expired array has a better static priority.

Furthermore, an interactive process P 's timeslice is divided into smaller pieces. Each piece has the size of $TIMESLICE_GRANULARITY(P)$, which is actually a macro that yields the product of the number of CPUs in the system and a constant proportional to $bonus(P, t)$ [1,4]. An interactive process does not receive any less timeslice, instead a task of equal priority may preempt the running process every $TIMESLICE_GRANULARITY(P)$. The process is then queued to the end of the list for its priority level. Processes at the same priority level run in round-robin fashion, so execution will rotate more frequently among interactive processes of the same priority, preventing them from blocking each other.

3.3. Sleep_avg scoring

The basic idea of $sleep_avg$ is to credit sleep time and penalize run time. However, the calculation of $sleep_avg$ is not a simple counter up and down. The current interactive status of the process is used to weight both sleep time and run time to introduce

some auto-regulation into the calculation [22]. The updating of $sleep_avg$ occurs at the moments that: (a) a process wakes up from sleep or blocking state, or (b) a process yields the CPU.

In the example of Fig. 2, at t_0 process P starts to run for a duration of t_r . At t_1 , P goes to sleep and yields the CPU to process Q . Then at t_2 , P wakes up and preempts Q . In general, the updating of $sleep_avg$ follows (4) and (5):

$$P \rightarrow sleep_avg(t_1) = \max\{0, P \rightarrow sleep_avg(t_0) - t_r * \alpha\}, \tag{4}$$

where α is a weighting factor for run time, $\alpha = 1 / \max\{1, bonus(P, t_0)\}$

$$P \rightarrow sleep_avg(t_2) = \min\{MAX_SLEEP_AVG, P \rightarrow sleep_avg(t_1) + t_s * \beta\}, \tag{5}$$

where β is a weighting factor for sleep time, $\beta = \max\{1, 10 - bonus(P, t_1)\}$.

However, when updating $sleep_avg$ for waking processes, special measures are taken to treat the following scenarios [1,4]: (a) Processes that sleep a long time are categorized as idle and will get minimally interactive status to prevent them suddenly becoming CPU intensive and starving other processes. (b) Processes waking from an uninterruptible sleep are limited in their $sleep_avg$ rise as they are likely to have been waiting on disk I/O, which is not a strong indicator of interactivity. (Most local disk I/O is associated with uninterruptible sleep.) (c) When an awakened process is put into a runqueue, there might be scheduling latency, which could be of a non-negligible duration. In this case, the time spent on the runqueue might or might not be credited as sleep time, depending on the state of the process when it was awakened. The state of the process is encoded within the process' *activated* field [1]. Let's assume that process P waits on runqueue for a period of t_w before it is scheduled

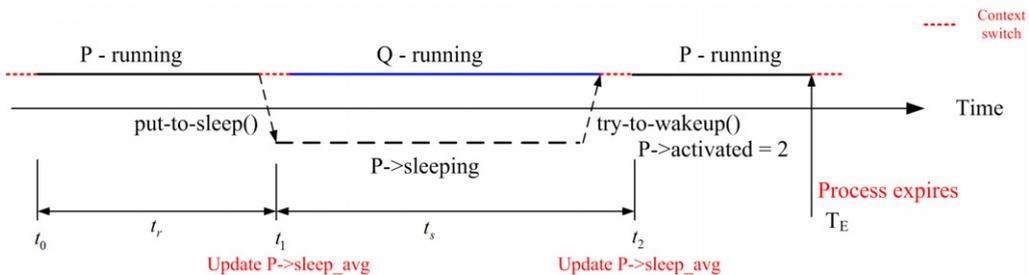


Fig. 2. Updating of $sleep_avg$.

Table 1
Credited sleep time vs. wait time on runqueue

$P \rightarrow \text{activated code}$	-1	1	2	0
Credited sleep time	0	$0.3 * t_w$	t_w	N/A

to run. The credited sleep time is as shown in Table 1. For example, a process might sleep to wait for data from network. Afterwards, when the process is woken up, its wait time on the runqueue is fully credited to the *sleep_avg* because its $P \rightarrow \text{activated code}$ is 2.

Since Linux only counts time in integral tick units, the Linux clock granularity might play a role when updating the *sleep_avg*: some sleep/run times are rounded up to the next whole tick, while others are rounded down. On average, these two effects tend to cancel out [23]. Furthermore, in Linux 2.6 the clock granularity is 1 ms level. In general, the *sleep_avg* is updated with reasonable accuracy.

4. Interactivity vs. fairness in networked Linux system

In previous sections we have discussed the Linux interactive scheduling mechanism: an expired interactive process is reinserted back into the active array, instead of the expired array. Interactive scheduling makes the Linux systems more responsive and interactive. However, interactive scheduling would bring the possibility of unfairness if the interactivity classification was inaccurate. For example, when a non-interactive process is incorrectly classified as interactive, reinserting it back into the active array will gain it extra scheduling runs, at the expense of other non-interactive processes. What's worse is that when a non-interactive process incorrectly gains interactive status, its dynamic priority is correspondingly enhanced, which might block some true interactive processes.

As remarked above, special measures have been taken to make interactivity classification accurate. Those measures are effective in preventing processes that mainly wait for disk I/O from being categorized as interactive [22]. However, our experiments and analysis have shown that the current interactivity classification mechanism is not effective in classifying network-related processes. It tends to classify applications like ftp and rcp as interactive when bandwidth is limited or the sender is slower than the receiver. Applications like ssh, telnet, and http clients are generally interactive applications; but

ftp, rcp, scp, and the like are not. If they are misclassified, it will raise scheduling fairness issues. In the following sections, we use a simplified model to analyze the fairness vs. interactivity issues.

Assume there is bulk data flowing from a sender to a receiver (as in ftp, for example). Process P is the data receiving process in the receiver. The network is relatively stable, and incoming packets are evenly spaced with a rate of N_i packets/s (pps). There is no other traffic directed to the receiver. This assumption holds for traffic patterns like voice over IP [24] or an ideal TCP self-clocking stream such as in [25]. In reality, the incoming traffic pattern is irregular. However, NAPI or “interrupt coalescing” will mask the arrival pattern and to some extent nullify its effect on the receiver. Similar conclusions are still expected to be valid, and are borne out by experiments. Also, let the NAPI driver's hardware interrupt time be T_{intr} , which includes NIC interrupt dispatch and service time; the software interrupt *softnet's* packet service rate be R_{sn} (pps); and process P 's data service rate is S_P (pps). When the network bandwidth is limited, or the sender's processing power is relatively slower than the receiver's processing power, we can assume that $N_i \ll R_{\text{sn}}$. Let process P have the default *nice* value of 0.

4.1. Single process receiver

Only process P runs on the receiver, no other processes. At time 0, P is waiting for network data from the sender (TCP or UDP).

As shown in Fig. 3, packets start to arrive at receiver at time 0. As an interrupt-driven operating system, the Linux execution sequence is: hardware interrupts \rightarrow software interrupts \rightarrow processes [1,2]. Packet 1 is first transferred to ring buffer, then the NIC raises a hardware interrupt to schedule *softirq* – *softnet*. Afterwards, the software interrupt handler (*softnet*) starts to move packet 1 from ring buffer to the socket's receive buffer of process P , waking up process P and putting it on the runqueue. During this period, new packets might arrive at the receiver. For example, packet 2 arrives during the period in Fig. 3. *Softnet* continues to process the packets within the ring buffer until it is empty. Letting T_{sn} be the duration that *Softnet* spends on the ring buffer, we see that

$$1 + [(T_{\text{intr}} + T_{\text{sn}}) * N_i] = T_{\text{sn}} * R_{\text{sn}}. \quad (6)$$

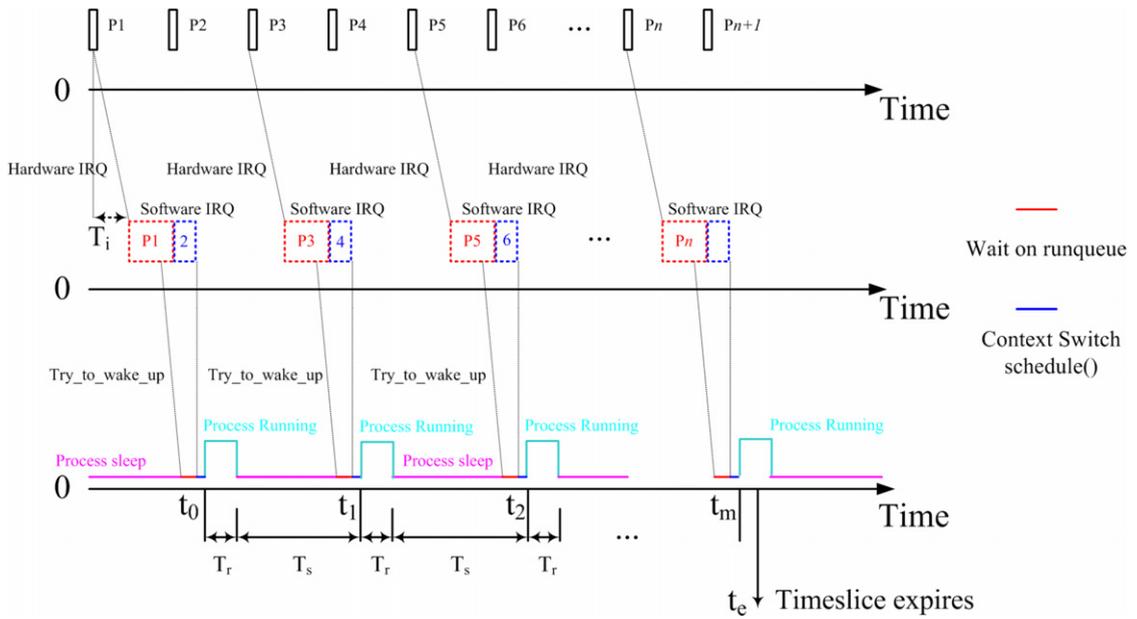


Fig. 3. Interactivity vs. fairness in networked Linux.

Here, $T_{sn} * R_{sn}$ is actually the number of packets that are handled together

$$T_{sn} = \left[\frac{1 + T_{intr} * N_i}{R_{sn} - N_i} * R_{sn} \right] / R_{sn}. \quad (7)$$

Then softirq yields the CPU. Process P begins to run, moving data from the socket's receive buffer into user space. Since there are $T_{sn} * R_{sn}$ packets in the receiver buffer, process P runs for a duration of $T_r = (T_{sn} * R_{sn}) / S_p$. Here, we are considering a relatively low incoming packet rate compared to the receiver's processing power. Before the next packet (P3 in Fig. 3) arrives at the receiver, process P runs out of data, and again goes to sleep, waiting for more. Either of two conditions could lead to a relatively low incoming packet rate: the network bandwidth from sender to receiver is low, or the sender's hardware is less powerful than the receiver's. If the next packet always arrives before process P goes to sleep, the sender will overrun the receiver. Incoming packets would accumulate in the socket's receive buffer. For TCP traffic, the flow control mechanism would take effect to slow down the sender.

When the next packet arrives at the receiver, the same scenario as described above occurs. The cycle repeats until process P stops. At time t_E , process P 's timeslice expires.

When incoming traffic wakes up process P , its wait time on runqueue is fully credited to the

sleep_avg. For the process being discussed, its $P \rightarrow activated$ code is 2. As shown in Fig. 3, process P runs for T_r and sleeps for T_s in each cycle.

Here

$$T_r = \frac{T_{sn} * R_{sn}}{S_p} = \frac{\left[\frac{1 + T_{intr} * N_i}{R_{sn} - N_i} * R_{sn} \right]}{S_p}, \quad (8)$$

$$T_s = \frac{\left[\frac{1 + T_{intr} * N_i}{R_{sn} - N_i} * R_{sn} \right]}{N_i} - \frac{\left[\frac{1 + T_{intr} * N_i}{R_{sn} - N_i} * R_{sn} \right]}{S_p}. \quad (9)$$

Following (4) and (5), it is easy to update $P \rightarrow sleep_avg(t)$ at time t .

From (8) and (9), it follows that

$$\frac{T_r}{T_s} = \frac{N_i}{S_p - N_i}. \quad (10)$$

Correspondingly, process P 's CPU share is

$$\frac{T_r}{T_r + T_s} = \frac{N_i}{S_p}. \quad (11)$$

Given the receiver and process P , S_p is fixed. Therefore, it can be derived from (3), (4), (5), and (10) that process P 's interactivity status would be strongly dependent on the packet arrival rate N_i , instead of interactive activities.

As illustrated in Fig. 3, we will count cycles of run and sleep beginning when the process wakes up. Cycle 1 starts at t_0 and ends at t_1 . Since an interval T_r of running is not more than 100 ms and

decreases $sleep_avg$ by αT_r , with $\alpha \leq 1$, $sleep_avg$ may fall to the next 100 ms bracket during the running portion of a cycle, but no further. This may increase β by 1, but no more. Referring to (4) and (5), we collect the possible changes of $sleep_avg$ in one cycle, $\Delta sleep_avg$, in Table 2.

From Table 2, we can surmise the following theorem.

Theorem 1. *Process P is the data receiving process in the receiver. The network is relatively stable, and incoming packets are evenly spaced with a rate of N_i (pps). And Process P's data service rate is S_P (pps). If $N_i/S_P < 0.9$, P will be categorized as interactive if it runs long enough.*

Proof. If $N_i/S_P < 0.9$, from (10) it can be derived that $T_r/T_s < 9$. From Table 2, it is seen that when $T_r/T_s < 9$, $\Delta sleep_avg > 0$ for any cycle. To categorize a process as interactive, it suffices to meet the condition in (3). Let us assume process P's initial $sleep_avg$ is $sleep_avg(0)$ when it is initially forked, and its nice value is 0.

- If $sleep_avg(0) \geq 700$ ms. Since $\Delta sleep_avg > 0$ for any cycle, process P will always be categorized as interactive.
- If $sleep_avg(0) < 700$ ms. Since $\Delta sleep_avg \geq 4T_s - T_r/6 > \frac{2}{2}T_s$ for any cycle, process P needs to run for some finite number of cycles n to achieve $\sum_{k=1}^n \Delta sleep_avg(k) > 700 \text{ ms} - sleep_avg(0)$.

Therefore, process P will meet the condition in (3) to be categorized if it is running long enough. \square

Theorem 1 shows that process P's interactivity status is strongly dependent on the packet arrival rate N_i , instead of its interactive activities. Clearly,

we can make the conclusions: network packets arrive at the receiver independently and discretely and the “relatively fast” non-interactive network process might frequently sleep to wait for packet arrival. Though each sleep lasts a very short period of time, the wait-for-packet sleeps occur so frequently that they lead to interactive status for the process.

The current Linux interactivity mechanism carries the chance that a non-interactive network process could consume a high CPU share, and at the same time be incorrectly categorized as interactive. For example, assuming $700 \text{ ms} \leq P \rightarrow sleep_avg(t_0) < 800$ ms, process P has gained interactive status. Based on Table 2, the change of $sleep_avg$ in each cycle is $4T_s - T_r/7$ (or $3T_s - T_r/7$). To keep the interactive status, it needs to meet the condition of $4T_s - T_r/7 \geq 0$ (or $3T_s - T_r/7 \geq 0$), Which is $T_r/T_s \leq 28$ (or $T_r/T_s \leq 21$). This condition can be easily met in normal network conditions. However, although process P keeps its interactive status, process P might still be using a high CPU percentage. When process P just meets the condition of $T_r/T_s \leq 28$ (or $T_r/T_s \leq 21$) to keep the interactive status, its CPU can reach as high as 96.55%. Table 3 shows process P's maximal CPU share at different scenarios while keeping its $sleep_avg$ in the indicated range.

4.2. Receiver plus other CPU load

In this case, process P runs on the receiver with M other non-interactive processes. All the processes have the same default nice value of 0.

Theorem 2. *Process P runs on the receiver with M non-interactive processes. All the processes have the default nice value of 0. Assume that the network is relatively stable, and P has already gained interactive*

Table 2
Changes of $sleep_avg$ in each cycle

$P \rightarrow sleep_avg(t_0)$	α	β	$\Delta sleep_avg$
$0 \leq P \rightarrow sleep_avg(t_0) < 100$	1	10	$10T_s - T_r$
$100 \leq P \rightarrow sleep_avg(t_0) < 200$	1	10 or 9	$10T_s - T_r$ or $9T_s - T_r$
$200 \leq P \rightarrow sleep_avg(t_0) < 300$	1/2	9 or 8	$9T_s - T_r/2$ or $8T_s - T_r/2$
$300 \leq P \rightarrow sleep_avg(t_0) < 400$	1/3	8 or 7	$8T_s - T_r/3$ or $7T_s - T_r/3$
$400 \leq P \rightarrow sleep_avg(t_0) < 500$	1/4	7 or 6	$7T_s - T_r/4$ or $6T_s - T_r/4$
$500 \leq P \rightarrow sleep_avg(t_0) < 600$	1/5	6 or 5	$6T_s - T_r/5$ or $5T_s - T_r/5$
$600 \leq P \rightarrow sleep_avg(t_0) < 700$	1/6	5 or 4	$5T_s - T_r/6$ or $4T_s - T_r/6$
$700 \leq P \rightarrow sleep_avg(t_0) < 800$	1/7	4 or 3	$4T_s - T_r/7$ or $3T_s - T_r/7$
$800 \leq P \rightarrow sleep_avg(t_0) < 900$	1/8	3 or 2	$3T_s - T_r/8$ or $2T_s - T_r/8$
$900 \leq P \rightarrow sleep_avg(t_0) < 1000$	1/9	2 or 1	$2T_s - T_r/9$ or $T_s - T_r/9$
$P \rightarrow sleep_avg(t_0) = 1000$	1/10	1	$T_s - T_r/10$

Table 3
Process P 's CPU share

$P \rightarrow \text{sleep_avg}(t_0)$	T_r/T_s	CPU share (%)
$700 \leq P \rightarrow \text{sleep_avg}(t_0) < 800$	21 or 28	95.45 or 96.55
$800 \leq P \rightarrow \text{sleep_avg}(t_0) < 900$	16 or 24	94.12 or 96
$900 \leq P \rightarrow \text{sleep_avg}(t_0) < 1000$	9 or 18	90 or 94.74
$P \rightarrow \text{sleep_avg}(t_0) = 1000$	10	90.91

status. For process P , if $N_i/S_P < 0.9$, no matter how many non-interactive processes run on the system, process P will have a CPU share of N_i/S_P , rather than $1/(M+1)$. The M non-interactive processes' total CPU share is: $(S_P - N_i)/S_P$, rather than $M/(M+1)$.

Proof. All processes begin in the active array, and are scheduled as described in Section 3. Since all processes have the same nice value, hence the same static priority, the second special interactivity rule to circumvent starvation for the processes in the expired array is not applicable here. Before process P is moved to the expired array, it will maintain its interactive status and have higher dynamic priority than the M non-interactive processes. This is due to the facts: (1) process P has already gained interactive status; (2) if it has $N_i/S_P < 0.9$, it has $\Delta \text{sleep_avg} > 0$ for any cycle, as proven in Theorem 1. Therefore, process P will not lose its interactive status. When process P expires, it will be reinserted in the active array, until the condition in the first special interactivity rule is satisfied.

Also, considering that Linux is preemptive: whenever a scheduler clock tick or interrupt occurs, if a higher-priority task has become runnable, it will preempt the running task as long as the latter holds no kernel locks. Therefore, no matter how many non-interactive processes run on the system, before process P is moved to the expired array, process P 's scheduling pattern is the same as that of the scenario discussed in Section 4.1, where only process P runs on the receiver. Correspondingly, process P 's CPU share won't change: it is N_i/S_P . The M non-interactive processes' total CPU share is: $(S_P - N_i)/S_P$. The M non-interactive processes can only run while process P sleeps.

According to the first special interactivity rule, "if the time since the first process in the active array expires is greater than or equal to $STARVATION_LIMIT \times NR_{\text{running}} + 1$, any expired processes are moved to the expired array without regard to their interactive status". In all, there are

$M+1$ processes in the runqueue, which implies $NR_{\text{running}} = M+1$. Also, $STARVATION_LIMIT = 1000$ ms. Let us denote the timeslice of a process with nice value η as $\text{timeslice}(\eta)$; $\text{timeslice}(0) = 100$ ms.

If it is the case that $N_i/S_P < 0.9$, then $(S_P - N_i)/S_P > 0.1$ and it follows that $STARVATION_LIMIT \times (M+1) \times (S_P - N_i)/S_P > M \times \text{timeslice}(0) + 1$. This implies that all the M non-interactive processes will expire and be moved to the expired array before the first special interactivity rule comes into effect. Since non-interactive processes are running only when process P is sleeping, at the moment when the last non-interactive process expires and is moved to the expired array, there is no runnable process in the active array. Then the active array is switched with the expired array, and a new cycle starts. Therefore, process P 's scheduling pattern is the same as that of the scenario discussed in Section 4.1. Correspondingly, process P 's CPU share won't change; it is N_i/S_P , instead of $1/(M+1)$. The M non-interactive processes can only run when process P is sleeping. The M non-interactive processes' total CPU share is $(S_P - N_i)/S_P$, instead of $M/(M+1)$. \square

From Theorem 2, it can be seen that networked Linux systems can have serious fairness problems. For example, if M is 10 and $N_i/S_P = 0.85$, then, process P 's CPU share would be as high as 85% while the total CPU shares of the 10 non-interactive process is only 15%. This establishes our conclusion that the Linux interactivity mechanism carries the chance that a non-interactive network process could consume a high CPU share, and at the same time be incorrectly categorized as interactive.

5. Experiments and analysis

To verify our claims in Section 4, we run data transmission experiments upon Fermilab's sub-networks, and the wide area networks between Brookhaven National Laboratory (BNL) and Fermilab (FNAL). In the experiments, we run *iperf* [26] to send data in one direction between two computer systems. *iperf*¹ on the receiver is the data receiving process P . The sub-networks used at Fermilab are as shown in Fig. 4a. The sender and receiver are attached to two Cisco 6509 switches connected to

¹ Iperf is multi-threaded; here we mean the iperf data transmission/reception thread.

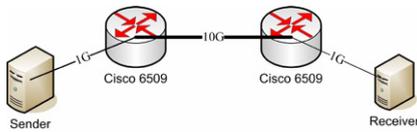


Fig. 4a. Fermilab sub-networks.

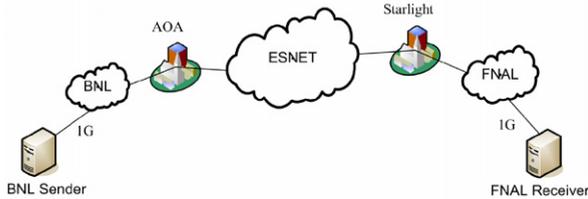


Fig. 4b. Wide area networks between BNL and FNAL.

each other by an uncongested 10-gigabit/second link. During the experiments, the background traffic in the network is low, and there is no packet loss or reordering in the network. For the network, the Round Trip Time (RTT) statistics are: $\text{min/avg/max/dev} = 0.134/0.146/0.221/0.25$ ms. In the experiments on local subnets, we use two different senders, one more powerful than the other. For simplicity, in the following sections, they are termed “Fast Sender” and “Slow Sender” respectively. The sender and receiver’s characteristics are shown in Table 4. Here, the *Fast Sender* and *Slow Sender* are relative to each other. At their full transmission capacities, both senders can saturate the Gigabit Ethernets.

The wide area networks between BNL and FNAL are as shown in Fig. 4b. During the experiments, data are transmitted from BNL to FNAL. There might be packet loss, or packet reordering in the wide area networks. The senders and receiver’s characteristics are shown in Table 5. The receiver is the same system as the one used on local subnets. For the network, the RTT statistics are: $\text{min/avg/max/dev} = 23.563/23.633/23.773/0.172$ ms.

In order to study the detailed interactive scheduling process, we have added instrumentation within Linux kernel. Specifically, (1) we keep track of the *sleep_avg* for each process at the moments its timeslice runs out; (2) we monitor the number of times that a process is reinserted into the active array due to its interactive status. For simplicity, it is termed “reinsertion count;” (3) we collect each process’ *stime* and *utime*² when it is terminated. Also to

² *stime*, *utime*: the time process spent in the kernel space and user space respectively.

study the effects of interactive scheduling on system performance, we create a non-interactive scheduling Linux, in which expired processes are inserted into the expired array, without regard to their interactivity status. In the following sections, we term “*WT*” for interactive scheduling, and “*NT*” for non-interactive scheduling.

To create non-interactive processes in the receiver, we run a purely CPU intensive application that executes a number of arithmetic operations in a loop. Non-interactive processes run as background loads. If there are m such processes in the receiver, it is termed as “*BLm*”. In all the experiments, the sender transmits one TCP stream to the receiver for 100 seconds. In the receiver, iperf is run as “*iperf -s -w 20M*”. All the processes are running with a nice value of 0. Further, since the transmission lasts for 100 seconds, in the receiver we calculate iperf’s CPU share as: $(\text{stime} + \text{utime})/100$ s. Consistent results were obtained across repeated runs. In the following sections, we present our experiment results.

5.1. Experiments over local subnets

Tables 6 and 7 show the iperf experiment results in the receiver for both slow sender and fast sender. In the experiments, the background loads are varied. For each group of data in the tables, we run the same experiments five times, and choose the group of data with highest throughput. The corresponding experiment results for iperf in the receiver are recorded. Those data include throughput, iperf’s CPU share, and reinsertion count. Also, in the experiments, we compare interactive to non-interactive scheduling. Iperf itself is not an interactive application. However, the experiment results in Tables 6 and 7 show that iperf’s interactive status is strongly dependent on the network conditions: iperf is more readily categorized as interactive with a slow sender than a fast sender. This verifies our claims in Section 4: when network packets arrive at the receiver independently and discretely, the “relatively fast” non-interactive network process might frequently sleep to wait for packet arrival. Though each sleep lasts a very short period of time, the wait-for-packet sleeps occur so frequently that they lead to interactive status for the process.

For better comparison and presentation, we show the reinsertion count of different experiment scenarios in Fig. 5. In the case of the slow sender, the reinsertion count is around 800 at different background loads; as for the fast sender, the highest

Table 4
Senders and receiver features for experiments upon fermilab’s sub-networks

	Fast sender	Slow sender	Receiver
CPU	Two Intel Xeon CPUs (3.0 GHz)	One Intel Pentium IV CPU (2.8 GHz)	One Intel Pentium III CPU (1 GHz)
System memory	3829 MB	512 MB	512 MB
NIC	Sysconnect, 32 bit-PCI bus slot at 33 MHz, 1 Gbps, twisted pair	Intel PRO/1000, 32 bit-PCI bus slot at 33 MHz 1 Gbps, twisted pair	3COM, 3C996B-T, 32 bit-PCI bus slot at 33 MHz, 1 Gbps, twisted pair

Table 5
Sender and receiver features for experiments upon wide area networks

	BNL sender	FNAL receiver
CPU	One Intel Pentium IV CPU (3.2 GHz)	One Intel Pentium III CPU (1 GHz)
System memory	1 G	512 MB
NIC	Intel PRO/1000, 32 bit-PCI bus slot at 33 MHz, 1 Gbps, twisted pair	3COM, 3C996B-T, 32 bit-PCI bus slot at 33 MHz, 1 Gbps, twisted pair

Table 6
Iperf experiment results in the receiver (slow sender)

Load	Scheduler	Throughput (Mbps)	CPU share (%)	Reinsertion count
BL0	WI	436	78.489	780
	NI	473	87.569	0
BL1	WI	443	81.573	815
	NI	285	49.923	0
BL2	WI	438	80.613	801
	NI	185	33.022	0
BL4	WI	430	79.217	785
	NI	113	20.025	0
BL8	WI	440	81.093	811
	NI	64.7	11.117	0

reinsertion count is only 47. As the experiment runs for 100 seconds, and the timeslice for a process with default nice value of 0 is 100 ms, there cannot be more than 1000 expirations of iperf’s timeslice. When eliminating factors of process sleep time and system interrupt time by noting iperf’s CPU share, reinsertion count of 800 implies that iperf is categorized as interactive almost all the time.

Experiment results in Tables 6 and 7 also verify the correctness of Theorem 2: interactive scheduling can lead to the fairness issue. As for non-interactive scheduling, when the number of background processes increases, iperf’s CPU share is correspondingly reduced. Basically, if the $M + 1$ processes run in the system, each process has its share of $1/(M+1)$. However, under interactive scheduling, iperf’s CPU shares are dependent on the network

Table 7
Iperf experiment results in the receiver (fast sender)

Load	Scheduler	Throughput (Mbps)	CPU share (%)	Reinsertion count
BL0	WI	464	99.228	7
	NI	478	99.975	0
BL1	WI	241	49.995	7
	NI	241	50.197	0
BL2	WI	159	34.246	8
	NI	160	32.826	0
BL4	WI	97.0	20.859	8
	NI	105	20.175	0
BL8	WI	74.2	15.375	47
	NI	58.3	11.143	0

conditions. With a slow sender, iperf’s CPU shares stays near 80%, no matter how many background processes there are. This is in accord with Theorem 2. With a fast sender, iperf’s CPU share is similar to what it receives under non-interactive scheduling. For better presentation, we show the results of CPU shares in Fig. 6. In the Figure, “FWT” represents fast sender and interactive scheduling in the receiver; “SWT” represents slow sender and interactive scheduling in the receiver; “FNT” represents fast sender and non-interactive scheduling in the receiver; “SNT” represents slow sender and non-interactive scheduling in the receiver.

To further probe the interactivity vs. fairness issues, we randomly choose two groups of experiment results. The experiments are run with background load of BL8, one with fast sender, and the other with slow sender. The experiment results are given in Figs. 7–10.

Figs. 7 and 8 give iperf’s *sleep_avg* in the receiver for slow and fast sender respectively. For the slow sender (Fig. 7), it can be seen that iperf’s *sleep_avg* is always greater than 700 ms. It means that iperf is categorized as interactive all the time. However, for the fast sender (Fig. 8), iperf is categorized as non-interactive most of the time. This is the reason that with a fast sender, iperf’s CPU share is similar to what it is under non-interactive scheduling. These

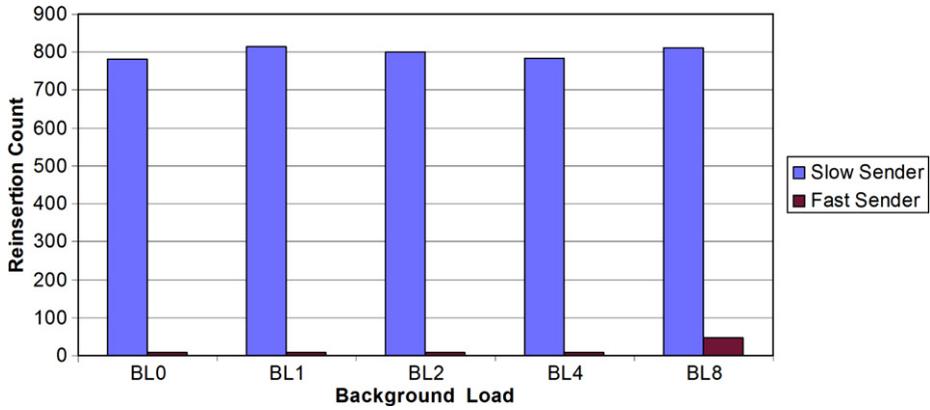


Fig. 5. Comparison of reinsertion count.

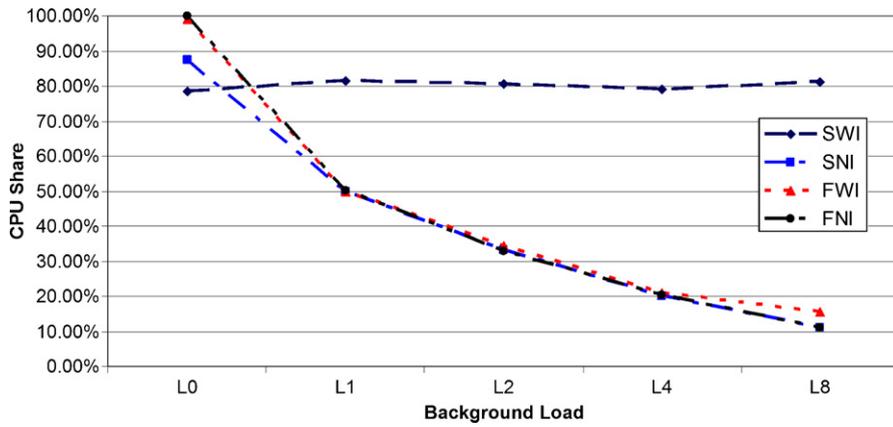


Fig. 6. Comparisons of CPU shares.

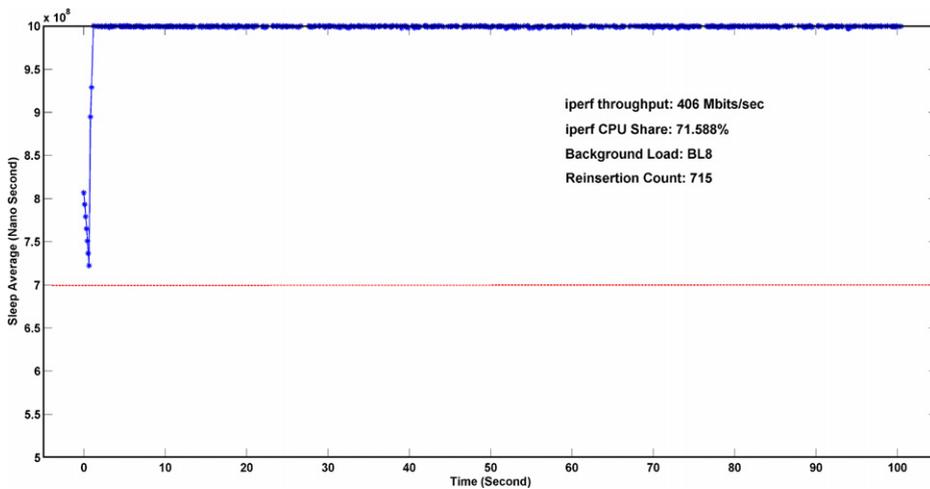


Fig. 7. Iperf's *sleep_avg* in the receiver (slow Sender).

experiment results agree with our analysis in previous sections. It further demonstrates that the cur-

rent interactivity classification mechanism is not effective in classifying network-related processes,

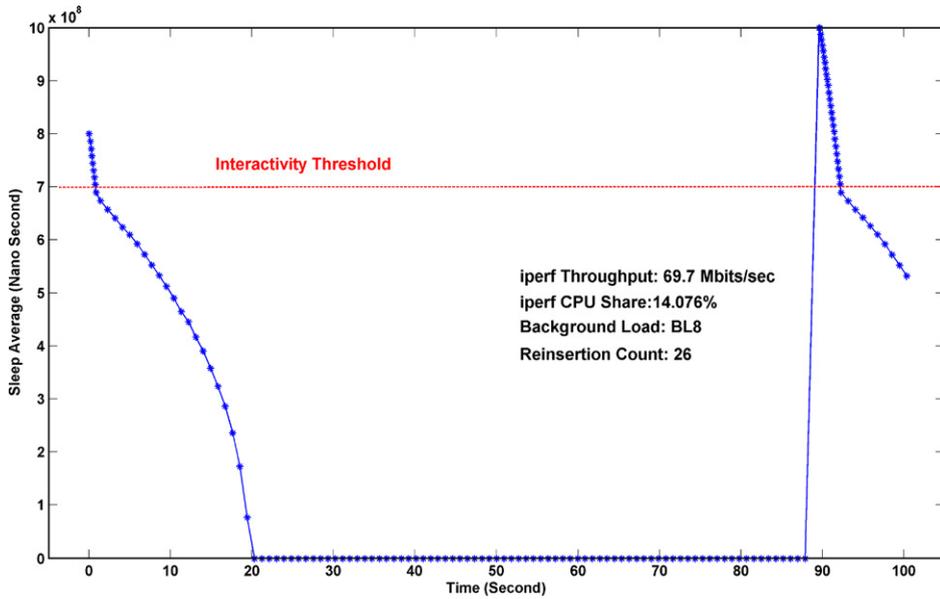


Fig. 8. Iperf's *sleep_avg* in the receiver (fast Sender).

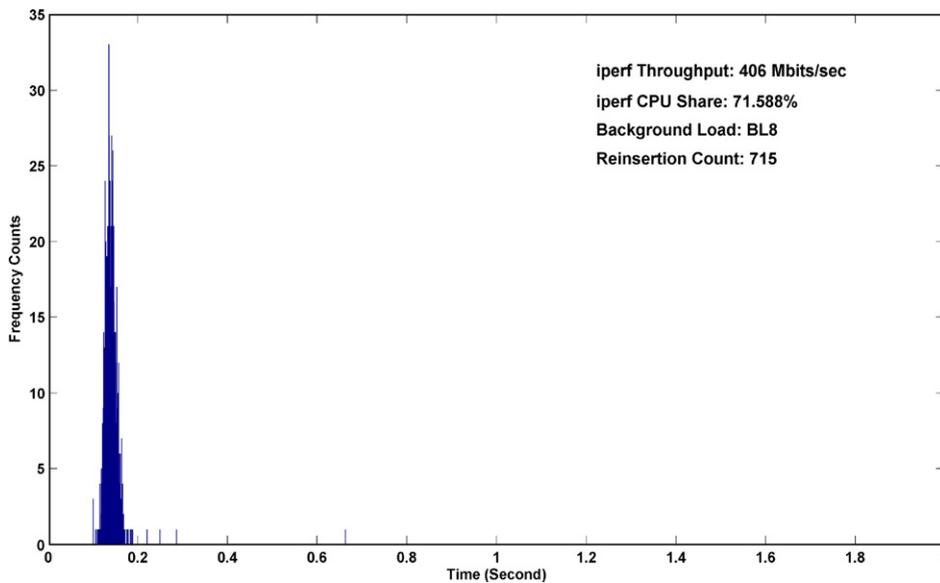


Fig. 9. Histogram of time intervals between consecutive timeslice expiration instants for iperf in the receiver (slow sender).

which are strongly dependent on the network conditions.

Figs. 9 and 10 give the histograms of time intervals between consecutive timeslice expiration instants for iperf in the receiver. These results verify the correctness of Theorem 2 from another perspective. Fig. 7 shows that with the slow sender iperf is always categorized as interactive. Therefore, each time iperf's timeslice expires, it is reinserted into

the active array, instead of the expired array. Also, due to its interactive status, iperf gains a priority bonus, resulting in higher dynamic priority than other non-interactive processes. Those non-interactive processes only run during the periods that iperf sleeps. Considering that facts that (1) with a nice value of 0, the timeslice is 100 ms; (2) iperf might sleep to wait for data, most of the time intervals between consecutive timeslice expiration instants

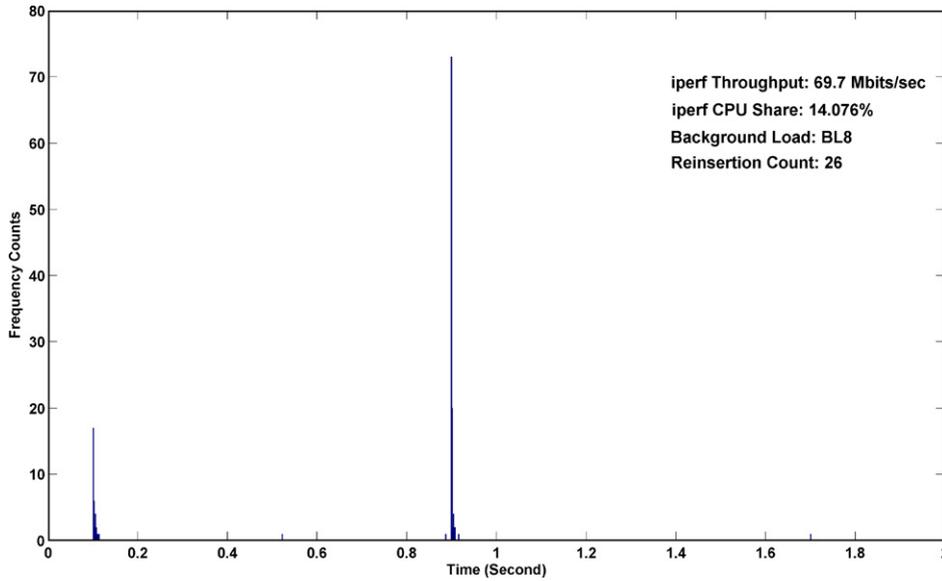


Fig. 10. Histogram of time intervals between consecutive timeslice expiration instants for iperf in the receiver (fast sender).

in Fig. 9 are between 100 ms and 200 ms. However, Fig. 10, the fast sender case, shows another story. This is due to the fact that iperf is non-interactive

most of time with a fast sender (Fig. 8). Once iperf’s timeslice expires, it will be moved to the expired array and can only regain the CPU after all eight non-interactive processes finish their timeslices. That is why the majority of the time intervals between consecutive timeslice expirations for iperf are greater than 900 ms.

Table 8
Iperf experiment results in the receiver

Load	Scheduler	Throughput (Mbps)	CPU share (%)	Reinsertion count
BL0	WI	325	75.877	713
	NI	304	65.68	0
BL1	WI	277	59.472	593
	NI	248	47.063	0
BL2	WI	274	58.996	588
	NI	195	31.922	0
BL4	WI	278	64.144	620
	NI	116	19.645	0
BL8	WI	273	58.788	586
	NI	79.8	9.717	0

5.2. Experiments over wide area networks from BNL to FNAL

We repeat our experiments over the wide area networks from BNL to FNAL. Experiment results also verify the claims of previous sections. Table 8 shows the iperf experiment results in the receiver. Fig. 11 gives the comparison of CPU shares. It

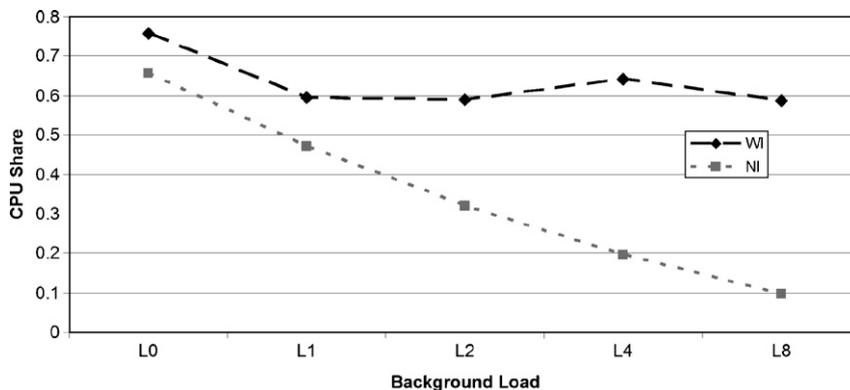


Fig. 11. Comparisons of CPU shares.

shows that the fairness issue also arises in wide area networking.

Figs. 12 and 13 give the results of one random wide area network experiment from BNL to FNAL. The background load of the experiment is BL8. Fig. 12 gives iperf’s *sleep_avg* in the receiver. It can be seen that iperf is also categorized as interactive all the time due to network conditions. Fig. 13 shows the histogram of time intervals between consecutive timeslice expiration instants for iperf in the receiver. It gives similar results as Fig. 9.

6. A possible solution

Our experiments and analysis described above have shown that the current interactivity classification mechanism is not effective in distinguishing non-interactive network processes from interactive processes, resulting in serious fairness/starvation problems. To summarize, the causes of this are: (1) network packets arrive at the receiver independently and discretely; the “relatively fast” non-interactive network process might frequently sleep to

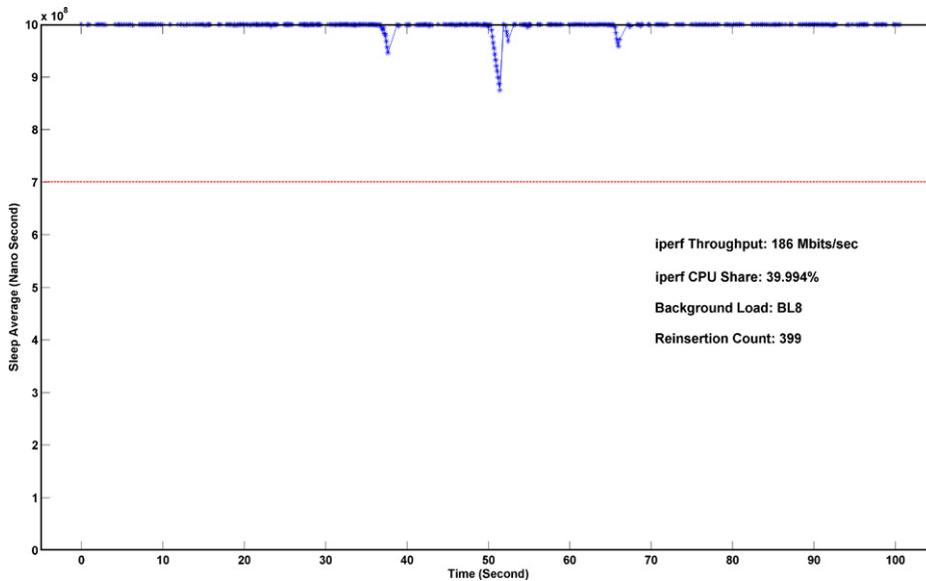


Fig. 12. Iperf’s *sleep_avg* in the receiver.

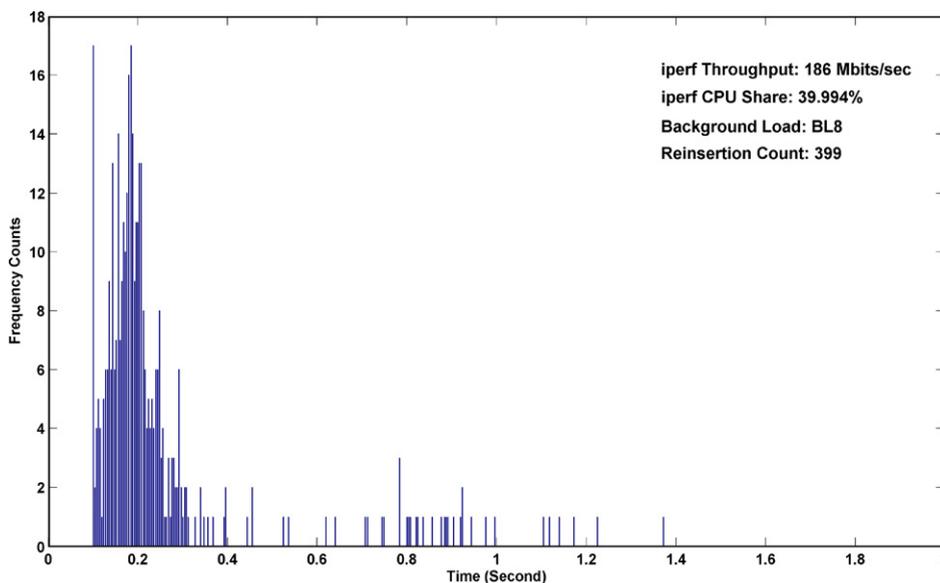


Fig. 13. histogram of time intervals between consecutive timeslice expiration instants for iperf in the receiver (WAN).

Table 9
Wait-for-packet sleep statistics for iperf data transmission experiment

Experiment	<2 ms (%)	<5 ms (%)	<10 ms (%)	<15 ms (%)	<20 ms (%)	Mean (ms)	Throughput (Mbps)
BNL -> FNAL (1)	68.32	83.82	97.79	99.84	99.88	2.2214	263
BNL -> FNAL (2)	68.72	85.08	98.85	99.92	99.95	2.0071	221
FNAL -> FNAL (1)	99.78	99.85	99.93	99.93	99.93	0.2285	383
FNAL -> FNAL (2)	99.70	99.79	99.88	99.88	99.89	0.2259	438

wait for network packets. Though each sleep lasts for a short period of time, they occur more than frequently enough to lead to interactivity status. (2) The current Linux interactivity mechanism provides the possibilities that a non-interactive network process could consume a high CPU share, and at the same time be incorrectly categorized as interactive. To resolve the interactivity vs. fairness issues there might be two basic approaches. One approach is to completely overhaul the interactivity mechanism. However, the current mechanism has been proven effective for traditional non-networked applications. Major modifications would be likely to affect those applications. Clearly, this approach might be complex and time-consuming. The second approach is to reduce or eliminate those *sleep_avg* updates triggered by short inter-packet sleeps under non-interactive conditions. We pursue the latter course.

Usually, network applications can be classified into the following categories:

- (a) Interactive network applications like ssh, telnet, and web browsing. Since those applications involve human interactions, the wait-for-packet sleeps in the receiver usually last for hundreds of milliseconds or even seconds to wait for user inputs. For example, in [16], Etsion et al. have reported that standard typing at a rate of about 8 characters per second. In the extreme case, if a packet was sent out for each character typed, the inter-packet space would be average around 125 ms.
- (b) Non-interactive network applications. Some non-interactive network applications, like ftp,³ gridftp, and scp, involve bulk data transmission. As explained above, due to packet-switched network's packet delivery nature: network packets arrive in the receiver indepen-

dently and discretely. The “relatively fast” network process in the receiver might frequently sleep to wait for network packets. Though each wait-for-packet sleep is short, they are very frequent. Iperf also belongs to this category. Table 9 gives the wait-for-packet sleep statistics for a group of data transmission experiments in Section 5. It shows that most wait-for-packet sleeps last for a few milliseconds or less.

- (c) Multimedia network applications. For these applications, network packets are transmitted and received periodically. For example, VOIP packets are transmitted and received every 20 ms. These applications are categorized as “soft real-time” so other measures should be taken, regardless of the issues investigated here, to guarantee their CPU shares and responsiveness. Possibilities include (1) In Linux 2.6, making use of *chrt* [27] to classify these applications as real-time. Linux 2.6 provides two real-time scheduling policies, SCHED_FIFO and SCHED_RR, which support soft real-time behaviors [1,2]. (2) When developing these applications, specifically requesting real-time support. Linux 2.6 provides a family of system calls to support such capabilities [2]. However such an approach might reduce application portability [28]. (3) Making use of a proportional-share scheduler [18,20] to provide protection between various classes of applications. This paper mainly address the interactivity vs. fairness issues for network applications of categories (a) and (b).

Table 9 gives us insight on how to distinguish interactive network applications from non-interactive ones: for a truly interactive application, the wait-for-packet sleeps usually last for tens or hundreds of milliseconds or more; however, the inter-packet sleeps for bulk data transmission applications usually last for a few milliseconds or less. Accordingly, to resolve the interactivity vs. fairness issues in networked Linux systems, our strategy is as

³ FTP implementations usually are multi-processed or multi-threaded: one process/thread is in charge of FTP control channel, which may be interactive; other processes/threads are in charge of data transmissions. Here, we mean FT P's data transmission processes/threads, and similarly for gridftp.

Table 10
Iperf experiment results in the receiver (slow sender)

Load	Scheduler	Throughput (Mbps)	CPU share (%)	Reinsertion count
BL0	O-WI	436	78.489	780
	N-WI	455	87.467	8
BL1	O-WI	443	81.573	815
	N-WI	304	56.38	221
BL2	O-WI	438	80.613	801
	N-WI	260	47.493	254
BL4	O-WI	430	79.217	785
	N-WI	181	32.682	177
BL8	O-WI	440	81.093	811
	N-WI	109	19.748	104

follows: when the sleep duration does not exceed some minimal value, *sleep_avg* for the network process will not be updated; *sleep_avg* is only updated when the sleep exceeds the threshold. We have modified the Linux kernel, and call this floor value the “Interactive Network Threshold”. The value is configurable through a new item in the /proc filesystem, /proc/sys/kernel/interactive_network_threshold, and its unit is milliseconds. It can be set according to the network conditions and the system’s purpose. If the system is mainly used for local area networks, a relatively small value such as 5 ms is quite enough. If the system is used for

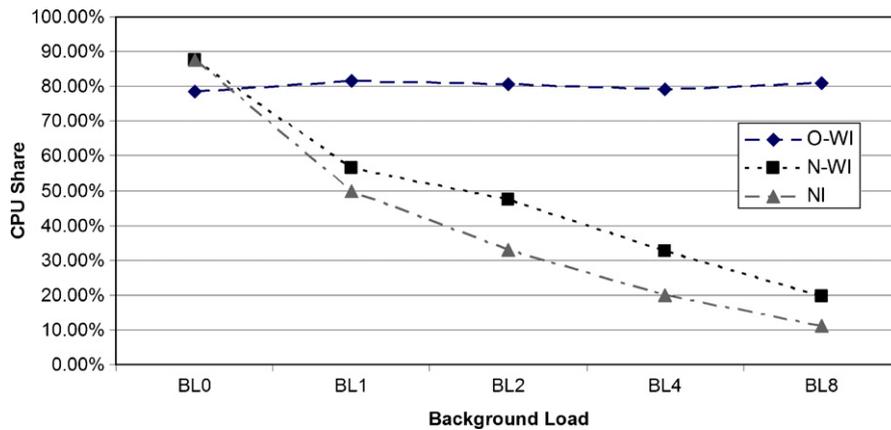


Fig. 14. Comparisons of CPU shares.

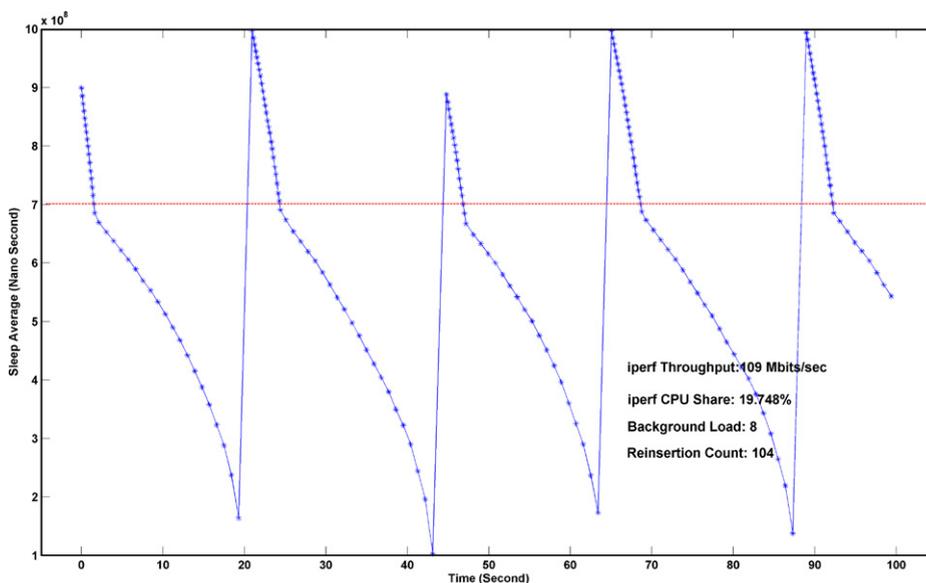


Fig. 15. Iperf’s *sleep_avg* in the receiver (slow sender).

wide area networks, and the packet jitter is high, `interactive_network_threshold` could be configured even higher. Usually high packet jitter implies low throughput; it would not cause serious fairness issues in the receiver. Therefore, `interactive_network_threshold` need not be too high. In our implementation, the default `interactive_network_threshold` is set at 30 ms. If system owner does not care about the interactivity

vs. fairness issues at all, it can be set as 0. If processing of streaming media such as VOIP is competing with other system loads and has not been protected as suggested above, an `interactive_network_threshold` of 15 ms may be better.

We repeat the data transmission experiments as described in Section 5 on the Linux updated with the new interactivity parameter described as above. We compare the new experiment data with those

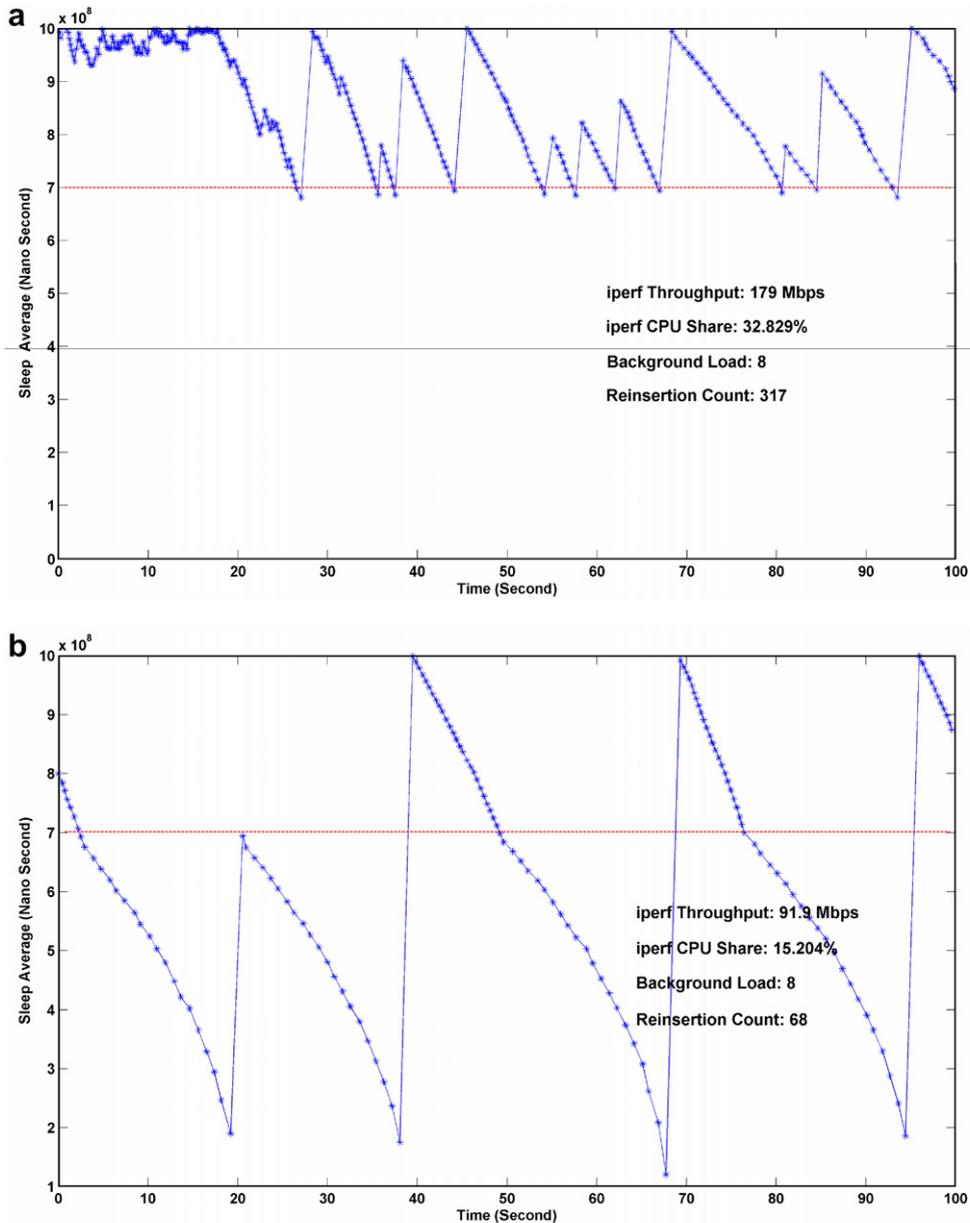


Fig. 16. Iperf's `sleep_avg` in the receiver for experiments from BNL to FNAL (a) `interactive_network_threshold` = 10 ms (b) `interactive_network_threshold` = 30 ms.

obtained in Section 5. The old experiment will be prefixed with “O-”, the new data with “N-”.

Table 10 shows the iperf experiment results in the receiver for experiments over Fermilab’s sub-networks. Since the fairness issue is not serious with the fast sender, the experiments are run only with the slow sender. The `interactive_network_threshold` is set as 5 ms. For better comparison and presentation, we show the comparisons of CPU shares in Fig. 14. It can be seen that: with the updated interactivity algorithm, iperf’s CPU share decreases as the background load increases; the reinsertion count of N-WI is much reduced compared to O-WI. Since `interactive_network_threshold` is set so low, it won’t affect the scheduling of true interactive network applications. The experiment results imply that our proposed solution is effective in resolving the fairness issues while maintaining the interactivity performance for true interactive network applications.

Fig. 15 shows iperf’s `sleep_avg` in the receiver with the updated interactivity algorithm for a randomly chosen experiment (`interactive_network_threshold=5 ms`, BL8). Compared with Fig. 7, it can be seen that most of the time iperf is not categorized as interactive. When it is not, it doesn’t gain extra runs at the expense of other non-interactive processes. This explains why iperf’s CPU share is effectively decreased when the background load is increased. It further verifies the effectiveness of our proposed solution. However, it still can be seen from Fig. 15 that iperf’s `sleep_avg` might jump from a low value to a much higher value, leading to the interactive status (also in Fig. 8). This is caused by the scheduling delay: when a low-dynamic-priority iperf wakes up upon packet arrival, it might wait on the runqueue for a relatively long time before it is scheduled to run, which is fully credited to the `sleep_avg`. Since the scheduling delays of interactive network processes cannot be differentiated from those of non-interactive processes, the influence of this type of scheduling delays is hard to eliminate. This is also the reason that the CPU shares in the N-WI runs are higher than in NI.

Similar results are obtained in experiments over the wide area networks from BNL to FNAL. Fig. 16 shows iperf’s `sleep_avg` in the receiver for two random experiments from BNL to FNAL with the new interactivity algorithm. In Fig. 16(a), `interactive_network_threshold` is set as 10 ms, while it is set as 30 ms in Fig. 16(b). It can be seen that for wide area networks, since the

packet jitter is higher, the `interactive_network_threshold` needs to be correspondingly configured higher. Setting `interactive_network_threshold` to 30 ms effectively improves the system’s fairness, while not affecting true interactive network applications’ performance. In Fig. 16, we also see scheduling delays causing jumps in `sleep_avg`.

7. Conclusions

Our researches have pointed out that the current Linux interactivity mechanism is not effective in distinguishing non-interactive network processes from interactive network processes, and results in serious fairness/starvation problems. Mathematical analysis and experiments results have verified our conclusions. Further, we propose and test a simple scheduler modification to address the interactivity vs. fairness problems in networked Linux systems. Experiment results have proved the effectiveness of our proposed solution. The improvements in fairness come at a cost: the network throughput for a given process may be reduced, while the CPU share, response time, or network throughputs of other processes are improved. This will be a desirable trade-off in some environments, but perhaps not in all.

Acknowledgements

We thank the editor and reviewers for their comments, which helped improve the paper. Also, we would like to thank Dr. Dantong Yu and Dr. Dimitrios Katramatos of Brookhaven National Laboratory. Without their sincere help, the wide area network experiments between BNL and FNAL were impossible.

References

- [1] D.P. Bovet et al., Understanding the Linux Kernel, third ed., O’Reilly Press, 2005, ISBN 0-596-00565-2.
- [2] R. Love, Linux Kernel Development, second ed., Noval Press, 2005, ISBN 0672327201.
- [3] C.S. Rodriguez et al., The Linux(R) Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures, Prentice Hall PTR, 2005, ISBN 0131181637.
- [4] www.kernel.org.
- [5] “Goals, Design and Implementation of the new ultra-scalable O(1) scheduler”, Linux Documentation, sched-design.txt.
- [6] A. Silberschatz et al., Operating System Concepts, seventh ed., John Wiley & Sons, 2004, ISBN 0471694665.

- [7] R. Love, Interactive kernel performance: kernel performance in desktop and real-time applications, in: Proceedings of the Linux Symposium, July 23–26, 2003, Ottawa, Canada.
- [8] M. Mathis et al., Web100: Extended TCP instrumentation for research, education and diagnosis, *ACM Computer Communications Review* 33 (3) (2003).
- [9] T. Dunigan et al., A TCP Tuning Daemon, *SuperComputing* (2002).
- [10] M. Rio et al., A Map of the Networking Code in Linux Kernel 2.4.20, March 2004.
- [11] J.C. Mogul et al., Eliminating receive livelock in an interrupt-driven kernel, *ACM Transactions on Computer Systems* 15 (3) (1997) 217–252.
- [12] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986, ISBN 0132017997.
- [13] U. Vahalia, *UNIX Internals: The New Frontiers*, Prentice Hall, 1995, ISBN 0131019082.
- [14] J. Mauro et al., *Solaris Internals Core Kernel Architecture*, first ed., Prentice Hall PTR, 2000, ISBN 0130224960.
- [15] M.K. McKusick et al., *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley Professional, 2004, ISBN 0201702452.
- [16] Y. Etsion et al., Process prioritization using output production: scheduling for multimedia, *ACM Transactions on Multimedia Computing, Communications, and Applications* 2 (4) (2006) 318–342.
- [17] C.A. Waldspurger et al., Lottery scheduling: flexible proportional-share resource management, in: Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation, Monterey, CA, November 1994.
- [18] P. Goyal et al., A hierarchical cpu scheduler for multimedia operating systems, in: Proceedings of the 2nd OSDI Symposium, October 1996.
- [19] J. Nieh et al., Virtual-time Round-robin: An O(1) Proportional Share Scheduler, in: Proceedings of the 2001 USENIX Annual Technical Conference, USENIX, Berkeley, CA, 2001, pp. 245–259.
- [20] K. Jeffay et al., Proportional share scheduling of operating system services for real-time applications, in: *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.
- [21] D. Petrou et al., Implementing lottery scheduling: matching the specialisations in traditional schedulers, in: Proceedings of the 1999 USENIX Technical Conference, pages 1–14, Monterey, CA, USA, June 1999.
- [22] <http://kerneltrap.org/node/780>.
- [23] Y. Etsion et al., Effects of clock resolution on the scheduling of interactive and soft real-time processes, in: Proceedings of ACM SIGMETRICS Conference, Measurement and Modeling of Computer Systems, June 2003, pp. 172–183.
- [24] J. Davidson et al., *Voice over IP Fundamentals*, second ed., Cisco Press, 2006, ISBN 1587052571.
- [25] V. Jacobson, Congestion avoidance and control, in: Proceedings of ACM SIGCOMM, Stanford, CA, August 1988, pp. 314–329.
- [26] <http://dast.nlanr.net/Projects/Iperf/>.
- [27] E. Siever et al., *Linux in a Nutshell*, fifth ed., O'Reilly Media, Sebastopol, CA, 2005, ISBN 0-596-00930-5.
- [28] Y. Etsion et al., Desktop scheduling: how can we know what the user wants?, in: Proceedings of the 14th international workshop on Network and Operating systems support for Digital Audio and Video, Cork, Ireland, 2004, pp. 110–115.



Wenji Wu holds a B.A. degree in Electrical Engineering (1994) from Zhejiang University (Hangzhou, PRC), and doctorate in computer engineering (2003) from the University of Arizona (Tucson, USA). He is currently a Network Researcher in Fermi National Accelerator Laboratory. His research interests include high performance networking, optical networking, and network modeling and simulation.



Matt Crawford leads the Wide Area Systems group in Fermilab's Computing Division. He holds a bachelor's degree in Applied Mathematics and Physics from Caltech and a doctorate in Physics from the University of Chicago. He currently manages the Lambda Station project, and his professional interests lie in the areas of scalable data movement and access.