Towards a Responsive, Yet Power-efficient, Operating System: A Holistic Approach *

Le Yan, Lin Zhong and Niraj K. Jha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{lyan,lzhong,jha}@princeton.edu

Abstract

Although computing hardware has become increasingly more powerful, computer responsiveness is still an important issue due to multi-tasking and software bloat. We propose a holistic approach for improving computer responsiveness through user focus-aware resource management for CPU, memory, disk I/O, and graphics processing. Previous approaches only address one or two of these problems simultaneously. To the best of our knowledge, our work is the first to address disk I/O scheduling for better responsiveness. It also offers better solutions for the other problems. We also exploit the user-perceived latency to perform dynamic voltage scaling of the CPU to reduce power consumption at run-time without sacrificing responsiveness. We implemented our approach in the Linux/X Window system (henceforth referred to as Linux/X) on an IBM Thinkpad R32 laptop with mobile Pentium 4-M processor, which has two performance levels with different frequency/supply voltage settings: high (30.0W at 1.8GHz/1.3V) and low (20.8W at 1.2GHz/1.2V). Experimental results show that user focus-aware resource management achieves a significant improvement in computer responsiveness and some in energy efficiency. For example, for $TuxRacer$, a video racing game, with $GpsDrive$, a navigation system, running in the background, it provides a reduction of 42.0% in user-perceived latency and 7.5% in energy consumption with respect to the Linux/X system.

1 Introduction

More people are working with computers and for more hours. Computer responsiveness is not only important for a pleasant user experience, but also critical for user productivity [1, 2]. As the computer hardware becomes increasingly powerful, user attention becomes the most precious computing resource [3]. An unresponsive computer simply wastes the most precious resource.

Computer responsiveness is important due to multi-tasking and software bloat [4]. Even if the hardware resource is more than adequate for each application, when multiple applications are running, the resource available to a specific application becomes inadequate. If an application is augmented with new features, extra resources may be consumed, outweighing the benefit of new features. Managing the hardware resource is primarily a task of the operating system (OS). As we know from our experience, however, existing OSs do not provide satisfactory responsiveness on a loaded and/or power-managed system. Motivated by the limitations of past work, we investigate how to tackle the unresponsiveness problem in Linux/X based computers by favoring the X server and processes under user focus, and managing the CPU, memory, disk, graphics, and power resources appropriately. Our solutions improve the responsiveness of a loaded and/or power-managed computer significantly. Unlike previous approaches on computer responsiveness, which address process scheduling and memory management only, we propose a holistic approach addressing process scheduling, memory management, disk I/O scheduling, dynamic voltage scaling (DVS) in the OS, and X client scheduling in the X Window system. As far as we are aware, this is the first work to address disk I/O scheduling for responsiveness. Our focus-aware resource management kernel module is available for download [5].

The paper is organized as follows. We address related work in Section 2 and present background material in Section 3. In Section 4, we discuss how to manage the CPU, memory, disk, graphics, and power resources to improve computer responsiveness. We also describe its implementation in the Linux/X system. In Section 5, we discuss the metrics for evaluating focus-aware resource management. We present experimental results in Section 6. We present discussions and conclusions in Section 7.

2 Related Work

In the past, interactive applications were primarily textual and spent a lot of time waiting for user input, and therefore involved low CPU usage. However, with the emergence of modern interactive applications, ranging from video games to sophisticated simulation and virtual reality environments, interactivity estimation based on either CPU load [6] or I/O load [7] information has become inaccurate. For example, $TuxRacer$ [8], a video racing game, involves high CPU usage, 96% on an average. It is both CPU-bounded and I/O-bounded. Favoring I/O-bounded processes over CPU-bounded ones cannot provide adequate support for modern interactive applications.

Unix-like OSs, including Linux, typically base their windowing system on the X Window system [9], in which a single X server handles graphics processing requests from all X clients. X client scheduling is, therefore, critical for any X Window system based computer to be visually responsive. This is addressed by the X server smart scheduler in [10]. However, the X server is just a process managed by the OS and may suffer resource starvation itself. The smart scheduler addresses this problem partially. Evans et al. [11]

*Acknowledgments: This work was supported in part by DARPA under contract no. DAAB07-02-C-P302 and in part by NSF under grant no. CCF-0428446.

propose to utilize user focus information to identify interactive applications in UNIX. They use a centralized interactive manager in the X Window system to identify interactive applications. The kernel then gives priority to the interactive applications in scheduling and throttles non-interactive applications from memory to reduce scheduling and page fault latencies, which are only two of the major contributors to user-perceived latency, as described in Section 3.1.

As power consumption has become a critical concern for mobile computers, dynamic power management (DPM) and DVS techniques have attracted a lot of attention. However, they may aggravate computer unresponsiveness, since applications have to run with less hardware capacity. Some works [12–15] target interactive applications and utilize user-perceived latency to evaluate their approaches. In [16], user-perceived latency is proposed to be used to drive DVS, which offers more room for DVS without harming computer responsiveness.

3 Computer Responsiveness

In this section, we present some background material on computer responsiveness and discuss how the Linux/X system tackles it.

3.1 User-perceived Latency

A user can only interact with one application at a given time. Such an application is said to be *under user focus*. How responsive the application under user focus is and how swiftly the focus can be switched determine computer responsiveness. Most experiences of unresponsiveness come from interactive applications, in which the user expects immediate visual causality between user input and computer response. Human beings generally do not feel any delay between the "cause," $e.g.$, pressing the mouse button, and the "effect," $e.g.$, a menu window popping up, if the delay is below the human-perceptual threshold (a value of 50-100ms is commonly used [17, 18]).

The time it takes a computer to execute code to respond to a user input is called the user-perceived latency. If the code is executed in an uninterrupted fashion, the latency is minimal and determined purely by hardware capacity. However, in real life, the code is executed intermittently due to hardware resource sharing and OS maintenance, leading to a longer user-perceived latency. In computers, sharing of CPU, memory, disk, and graphics resources increases the user-perceived latency in different ways.

Scheduling latency: Scheduling latency is the delay between a process becoming runnable and actually running on the CPU. During this interval, another process or the kernel itself may be running instead.

Page fault latency: A page fault occurs when a process requests data that belong to its address space, but are not currently in memory due to its limited size. The OS is triggered to fetch the page from the hard disk into the memory. While handling a page fault, the OS may schedule another process to run on the CPU until the requested data are available. The page fault latency is the amount of time between a page fault occurring and the process resuming execution.

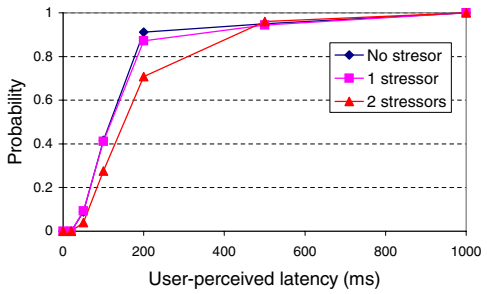
Disk latency: Since the data bandwidth of a hard disk is much smaller than that of the memory, it forms the bottleneck in handling page faults and could cause even more delay when multiple processes access the hard disk simultaneously. The process issuing the disk I/O request blocks until the requested data are available. The disk latency is the interval between a process issuing a disk request and resuming execution. Unlike CPU speed and memory capacity, which are governed by Moore's Law, the seek and access times of hard disks have improved at a much lower pace [19]. Therefore, disk latency has become critical for computer responsiveness.

X client scheduling latency: In X Window system based computers, a single X server handles all graphics processing requests. The CPU quantum for the X server process is used to process requests from different processes (X clients). X client scheduling is, therefore, critical for any X Window system based computer to be visually responsive.

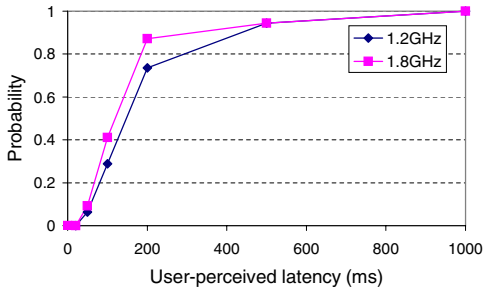
Normally, a computer should be able to handle a well-implemented application responsively. However, resource competition increases in two scenarios and introduces unresponsiveness. The first is a loaded system on which an application under user focus is running with some resource-hungry applications (*stressors*) in the background. Fig. 1(a) shows the cumulative distribution function (CDF) of the user-perceived latency for $TuxRacer$ with tar, the Linux archiving utility, running in the background, on an IBM Thinkpad R32 Linux/X based laptop with mobile Pentium 4-M processor (having two performance levels with different frequency/supply voltage settings: *high* (30.0W at 1.8GHz/1.3V) and *low* (20.8W at 1.2GHz/1.2V)). The distribution shifts towards longer latencies as the number of stressors increases, indicating that the user-perceived latency increases and the user may perceive time lags. In the second scenario, power-saving DPM/DVS techniques may aggravate resource competition. Fig. 1(b) shows the changes in the user-perceived latency when the laptop utilizes Intel SpeedStep technology [20] to change its performance level from *high* to *low* for $TuxRacer$. As the processor scales its frequency/supply voltage, the distribution of the user-perceived latency shifts towards longer latencies. As the computer becomes less responsive, user productivity degradation may increase system energy consumption for completing a task even with a smaller average power consumption. The above scenarios demonstrate the importance and necessity of guaranteeing responsiveness for computers, especially mobile computers, on which multiple applications may run simultaneously, and energy efficiency is important.

3.2 The Inadequacy of Linux/X

Linux has been historically and organizationally separated from the windowing system. Most Linux computers come with XFree86 [21], an open-source implementation of the X Window system. From the kernel's perspective, the X server is just a single process, although it may serve multiple applications. There is no communication between the X server and kernel regarding user behavior. Linux relies on estimation to identify interactive applications. It uses sleep time as the metric to estimate the interactivity of applications. However, as mentioned earlier, this assumption is not always valid. Many non-interactive applications sleep due to page faults and disk accesses, while many modern interactive applications are CPU-intensive. Table 1 summarizes interactivity estimation performed by Linux for commonly-used applications. Linux mistakes CPU-intensive interactive applications for non-interactive ones and considers most disk-intensive applications as non-interactive. Its estimation for gcc toggles between interactive and non-interactive from time to time. Fig. 2 demonstrates the limitations of the Linux interactivity estimator, in which $interactivity = 1$ refers to an interactive application and $interactivity = 0$ to a non-interactive one. The esti-



(a) A loaded system



(b) A power-managed system

Fig. 1. User-perceived latencies for *TuxRacer*

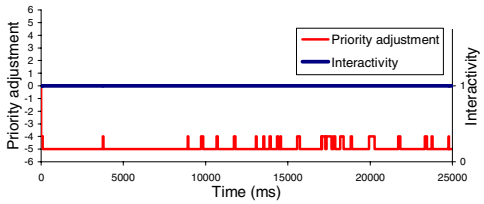
mator considers tar as an interactive application and gives it a priority bonus (-5) to boost its performance. On the other hand, $TuxRacer$, which is both interactive and CPU-intensive, is regarded as non-interactive and receives a priority penalty (+5) when the race starts. Based on the inaccurate estimation, Linux cannot provide adequate support for alleviating user experience of unresponsiveness.

Table 1. Linux interactivity estimation

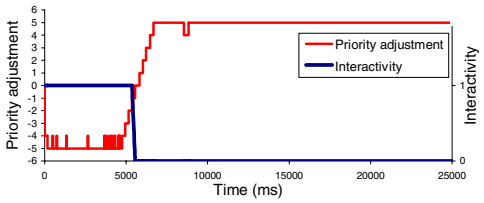
Application	Interactivity		Comments
	Actual	Linux	
$KEdit$	Yes	Yes	KDE text editor
$Mozilla$	Yes	Yes	Web browser
$Xpdf$	Yes	Yes	PDF viewer
$GpsDrive$	Yes	Yes	Navigation system
$TuxRacer$	Yes	No	Racing game
$FFmpeg$	No	Yes	Audio/video format converter
gcc	No	Yes/No	GNU C/C++ compiler
$MPlayer$	No	Yes	Media player
tar	No	Yes	Linux archiving utility

Besides its inaccuracy in identifying interactive applications, Linux offers no memory and disk management support for them. Many interactive applications are busy only sporadically, thus vulnerable to the page replacement algorithm and the disk I/O scheduling algorithm used by Linux. Fig. 3 shows the CDFs of page fault and disk latencies for $TuxRacer$ with a growing number (0 to 2) of stressors (tar) running in the background. Both distributions shift towards much longer latencies as the number of stressors increases. This indicates that for better responsiveness, memory and disk management strategies should favor the application under user focus.

The smart scheduler [10] of the X server schedules X clients at intervals of 20ms. The scheduler gives a client a unit priority penalty if it is still running after one interval. It gives the client a unit priority bonus if it has been idle for one interval or it has received a keyboard or mouse event. Since the client that receives keyboard or mouse events is under user focus, the smart scheduler is indeed focus-aware. However, it favors such clients conservatively. For example,

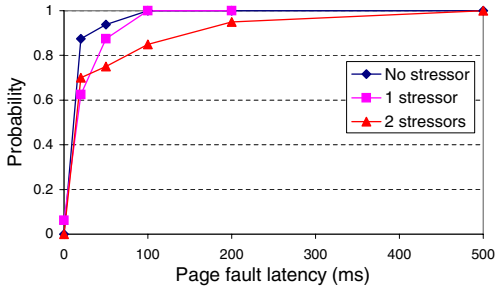


(a) Tar

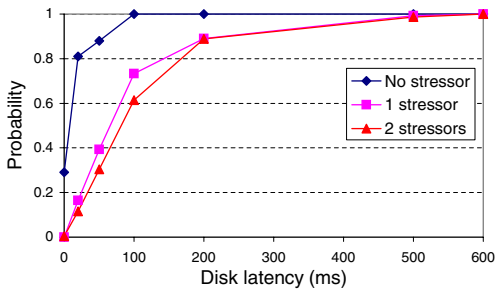


(b) $TuxRacer$

Fig. 2. Inaccuracy of Linux interactivity estimation



(a) Page fault latency



(b) Disk latency

Fig. 3. Executing *TuxRacer* with stressor *tar*

when multiple graphics-intensive clients are executed, the client under user focus may suffer.

4 Focus-aware Resource Management

In this section, we first discuss how to track the processes under user focus. We then detail how to favor them during process, memory, disk I/O, and power management through the OS and graphics management by the X Window system. Finally, we present our implementation of focus-aware resource management on the Linux/X system.

4.1 Tracking of Processes Under Focus

The X server knows exactly which X client is currently under user focus so that it can appropriately deliver keyboard and mouse events. However, it does not know the process id (*pid*) for any X client. Similar to the technique in [22], we utilize two pad fields in the data sent by an X client to the X server when the former attempts to open a connection with the latter. We modified the X11 library slightly, so that one pad field is set to the X client's *pid* and the other is set to some specific number to ascertain that the first pad field holds a valid *pid*. The second pad field is also used to indicate whether the connection is remote or local.

The X server records the *pid* for each X client. We added a global variable to the X server for the *pid* of the process under user focus (called *root-focus*), which is initialized to zero. When the X server sends a keyboard or mouse event to an X client, it checks whether the X client's *pid* is the same as the *root-focus*. If it is not, the X server conveys the *pid* of the new *root-focus* to the OS using a system call.

Since the modified X Window system can only identify the processes under user focus if the corresponding applications are X-based, the *root-focus* process thus identified sometimes is not the process that the user is indeed interested in, as illustrated in the following example.

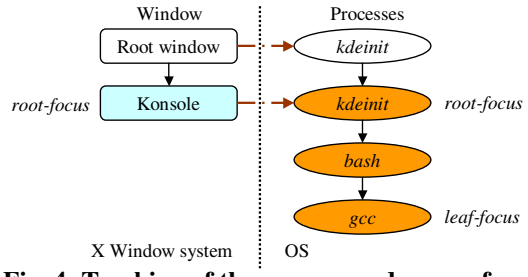


Fig. 4. Tracking of the process under user focus

Example 1: As shown in Fig. 4, in a $Konsole$ (KDE GUI console emulator) session, the X server identifies the $Konsole$ as the *root-focus* process, while the user is actually interested in gcc compilation running in this session. In terms of processes, gcc is a child process of $bash$ (command shell), which is in turn a child of $Konsole$. Each process maintains information for its children in Linux. We modified the kernel so that upon receiving the *pid* of the *root-focus* process, it marks this process $Konsole$ and its descendants, $bash$ and gcc, as under user focus. Process gcc, which the user is indeed interested in, is termed the *leaf-focus* process. In such cases, the X server has a different process under user focus than the OS. This is natural since the X server manages the graphics processing resource and the OS manages the CPU, memory, and disk resources. ∎

4.2 Resource Management

Given the processes under user focus, the OS and X server can allocate various resources in a focus-aware fashion and perform power management in a user-friendly way.

4.2.1 Process management

We introduced a new field, $focus\_flag$, to the task descriptor of each process, which indicates that the process is currently under user focus. The $focus\_flag$ of all the processes not under user focus is 0. For each process under user focus, the $focus\_flag$ is positive, either 1 or 2. The $focus\_flag$ of the *leaf-focus* process, which the user is actually interested in, is 1, while the $focus\_flag$ of all the other processes under user focus is 2. Let us revisit Example 1. The $focus\_flag$ of the *leaf-focus* process gcc, which is under user attention, is set to 1. The $focus\_flag$ of other processes under user focus, $Konsole$ and $bash$, is set to 2.

The scheduler determines which process should run when and for how long. Priority scheduling is often employed in many general-purpose OSs. Each process is given a priority and the runnable process with the highest priority is selected to run. Assigning a higher priority to the processes under user focus in order to reduce the user-perceived latency is straightforward, since it decreases the scheduling latency for these processes directly. In this work, instead of changing the scheduling policy of the Linux scheduler, we utilize its quantum and priority assignment mechanisms. The priorities of the processes under user focus are boosted, since each such process is involved in the current user-computer interaction. Once the user switches focus, the kernel resets the priorities of all the previous processes under user focus, and clears their corresponding $focus\_flag$. Then the kernel marks the $focus\_flag$ of current processes under user focus and gives a bonus (-15) to their priorities. The kernel checks whether there is any runnable process under user focus, whose priority is higher than the priority of the currently-running process. If there is, the execution of the currently-running process is suspended and the scheduler is invoked to select another process to run (usually the runnable process under user focus).

The X server, which is treated as a single process in the OS, interacts with multiple processes. To avoid resource competition among the X server and other processes that are not under user focus, the OS assigns a higher priority to the X server once it starts. In this way, the X server can preempt the processes not under user focus whenever user input or request for graphics processing arrives.

4.2.2 Memory management

Even if the processes under user focus are assigned a higher priority than other processes, its execution may be delayed due to memory competition. In [11], a certain amount of physical memory is reserved for interactive applications when the available memory is tight. This ensures that the memory is available to interactive applications during startup. However, pages in the address space of the processes under user focus are still vulnerable to the page replacement algorithm, which determines which pages to keep in memory and which pages to swap out when free memory becomes scarce. Most page-replacement algorithms, such as clock paging, not recently used (NRU) paging, and least recently used (LRU) paging, are unaware of current user focus. When some process not under user focus causes a page fault and there is no page frame available, the page replacement algorithm may choose the page owned by the process under user focus as a victim. When the process under user focus addresses data in the victim page, a page fault occurs and the OS traps, suspending this process, which increases the user-perceived latency.

In this work, we slightly changed the LRU-based page replacement algorithm in Linux. Linux maintains two lists, *active list*, for pages that were recently accessed, and *inactive list*, for pages that have not been accessed for a long time. Pages in the *active list* are exempt from reclaiming. The age of a page describes how desirable it is to keep this page frame in the *active list*. When the kernel scans through the *active list* for pages to move to the *inactive list*, the age of a page increases (decreases) whenever the page was (not) referenced. The page becomes a candidate for being moved to the *inactive list* when its age reaches zero. We maximize the age of the pages in the code segment of the *leaf-focus* process. The possibility of being swapped out for such pages is decreased. The number of page faults caused by the *leaf-focus* process is thus reduced.

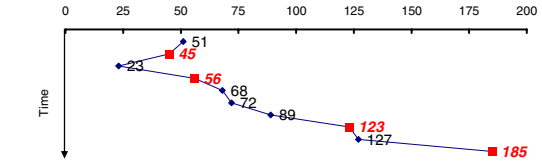
4.2.3 Disk I/O scheduling

Accessing the hard disk is time-consuming, since the hard disk controller must move the head to the exact position where the data are recorded. Current disk I/O scheduling algorithms, such as elevator, deadline, and anticipatory scheduling [23], aim at reducing the amount of time the disk spends seeking, thus increasing I/O throughput. However, the requests from the processes under user focus may suffer, aggravating computer unresponsiveness, as illustrated next.

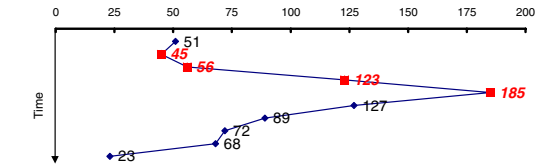
Example 2: Consider a disk with 200 cylinders (0-199). A request comes in to read a block on cylinder 51. While the seek to cylinder 51 is in progress, new requests come in to read blocks on cylinders 89, 185, 45, 127, 56, 123, 68, 72, 23, in that order. Requests for cylinders 185, 45, 56, and 123 are from the *leaf-focus* process. When the current request for cylinder 51 is taken care of, the disk I/O scheduling algorithm decides which request to handle next. Consider one of the disk I/O scheduling algorithms in Linux, the LOOK variation of the elevator algorithm [24]. It keeps moving in one direction until there are no more requests in that direction and then switches to the opposite direction. Fig. 5(a) shows the disk head movement for the LOOK strategy in Linux, assuming it was initially moving down. This order yields a total head movement of 190 cylinders. However, the request for cylinder 185 from the *leaf-focus* process is postponed to the end in the order, waiting until the head has moved 190 cylinders. The delay increases the disk latency, which may make the user perceive the time lags. ■

Table 2. No. of head movements for the disk I/O requests from the *leaf-focus* process

Requested cylinder	Original LOOK	Focus-aware LOOK
185	190	146
45	6	6
56	61	17
123	128	84



(a) Original LOOK



(b) Focus-aware LOOK

Fig. 5. Different strategies for scheduling disk I/O requests

In Linux, a request queue is maintained for the pending I/O operations for the disk. Requests are arranged in the request queue according to their location on the disk. The request at the head of the request queue is closest to the disk head, which reduces the seek time. The disk I/O scheduling algorithm attempts to merge requests to adjacent locations on the disk by combining two requests. If a request cannot be merged, the algorithm attempts to insert it in the request queue in the position that maintains the request queue's least seek time. In this work, we slightly modified the Linux LOOK disk I/O scheduling algorithm to make it focus-aware. We added a field $focus$ to the request descriptor of each request. If a request is from the *leaf-focus*, its $focus$ is set to 1. Otherwise, it is set to 0. Algorithm 1 gives the pseudo-code of the focus-aware LOOK algorithm. The requests with $focus = 1$ are always located ahead of others with $focus = 0$ in the request queue. When a new request arrives, the algorithm scans old requests from the tail of the request queue. If the new request is from the *leaf-focus* process, the algorithm skips the old requests with $focus = 0$. When the algorithm reaches an old request with $focus = 1$, it behaves in the same way as Linux LOOK afterwards. If the new request is not from the *leaf-focus* process, the algorithm also behaves in the same way as Linux LOOK. The focus-aware LOOK algorithm ensures that the requests from the *leaf-focus* process are always taken care of first. Fig. 5(b) shows the disk head movements for Example 2 using the focus-aware LOOK algorithm. Table 2 compares the number of head movements for the requests from the *leaf-focus* process using the above two algorithms. The average reduction of head movement for the focus-aware LOOK algorithm is 32.4% with respect to the Linux LOOK algorithm. However, the total number of head movements is increased from 190 to 308, indicating the overhead for the background processes.

Our strategy is also applicable to other disk I/O scheduling algorithms for improving computer responsiveness. Consider the deadline [25] and anticipatory [26] disk I/O scheduling algorithms in Linux. The former is a cyclic elevator with a deadline. It assigns a deadline to each request, 500ms to a read request and 1s to a write request. The latter is a cyclic elevator with a waiting policy. It waits for 6ms after completing a read request from a process, given that the process may issue an additional read request that does not require a seek operation. Both algorithms can be made focus-aware by modifying the cyclic elevator using the approach discussed above. Moreover, for the deadline disk I/O scheduling algorithm, assigning a shorter deadline, say, the human perceptual threshold (in the 50-100ms range), to the requests from the *leaf-focus* process can be effective in reducing disk latency for this process and thus improving computer responsiveness.

Algorithm 1 Focus-aware LOOK algorithm

```
1: Fetch next request req;
2: Request r ←{request queue tail};
3: while r! ={request queue head} do
4:    if req.focus = 1 then
5:       if r.focus = 0 then
6:          continue;
7:       else
8:          use the Linux LOOK strategy;
9:       end if
10:   else
11:      use the Linux LOOK strategy;
12:   end if
13: end while
14: if r.focus = 1 && r cannot be merged or inserted then
15:    insert r right before the first request with focus = 0 in the request
       queue;
16: end if
```

4.2.4 X client scheduling

The X server is in charge of providing window system services to many applications simultaneously. It distributes its limited resources among these applications using the X client scheduling algorithm. The X client under user focus may suffer when it competes for resources with multiple graphics-intensive clients. In this work, we modified the X server to give the X client under user focus the highest priority temporarily. Once the *root-focus* X client becomes ready, the X server processes its requests immediately. To make the focus switch smooth, the X server still maintains the smart scheduling priority [10] for each client, including *root-focus*. Upon user focus switch, the X server recovers the smart scheduling priority of the previous *root-focus* client. Giving the highest priority to the *root-focus* client may delay graphics processing requests from other clients that are not under user focus. However, since the user only

focuses on one application at any instant, the delay in the applications not under user focus may not be noticed.

4.2.5 Latency-driven DVS

Based on the work in [16], we utilize user-perceived latency to drive DVS. With the knowledge of the process under user focus, the DVS policy can trade off performance for power consumption judiciously, taking the quality of user experience into account. The X server computes an exponentially moving average of past user-perceived latencies given by Equation (1):

$$P_i = \frac{wP_{i-1} + M_{i-1}}{w+1} \quad (1)$$

where w is a decay factor. P_i and M_i denote the predicted and measured user-perceived latency for the i^{th} user input, respectively. If the predicted user-perceived latency for the next user input is larger than the maximum value of the human-perceptual threshold K_{max}, which means that the system responsiveness is unacceptable, the X server notifies the OS to increase the frequency/supply voltage level. On the other hand, if the predicted value is below the minimum value of the human-perceptual threshold K_{min}, the user may not perceive time lags, and hence the system can remain running at a lower performance level, thereby reducing power consumption. If the predicted value is between K_{max} and K_{min}, the OS does not change the current performance level to avoid unnecessary frequency/supply voltage transition overhead. The value of the decay factor w impacts the number of frequency/supply voltage transitions. Smaller values imply the OS may adjust the frequency/supply voltage very frequently. The transition time overhead increases the user-perceived latency, which makes the user feel that the computer is unresponsive. The transition energy overhead increases the total energy consumption. Larger values imply the number of transitions is decreased. However, the system may continuously run at the higher performance level before the OS is notified to decrease the frequency/supply voltage to reduce the power consumption. Thus, we choose $w = 3$ in this work. We use $K_{min} = 50$ms and $K_{max} = 100$ms to drive DVS. Note that both values are tunable in our implementation.

4.3 Implementation Issues

After having presented the design and implementation of the focus-aware resource management technique for each resource, we now present its overall architecture and address the new system calls required by the implementation.

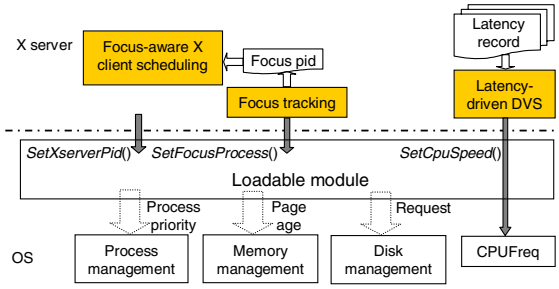


Fig. 6. Architecture for focus-aware resource management

Fig. 6 shows the overall architecture for our user focus-aware resource management technique. Each resource management block was described in Section 4.2. Focus-aware OS resource management, including all the new system calls, is implemented as a kernel loadable module available at [5]. The user can load or unload it easily without rebooting the system.

We introduced three new system calls to implement the communication between the X server and Linux kernel, as shown in Fig. 6. System call *SetXserverPid* notifies the *pid* of the X server to the kernel as soon as the X server is started. Then the kernel gives a bonus (-15) to its priority to boost its performance. The time overhead of *SetXserverPid* is 0.63μs on an average. System call *SetFocusProcess* first resets priorities of all the previous processes under user focus. Then it tracks the process hierarchy using a field, p_cptr, in the task descriptor of the process, to find the processes currently under user focus and set their corresponding $focus\_flag$. It boosts the priorities of all the current processes under user focus and maximizes the page ages of the code segment of the current *leaf-focus* process. The time overhead of *SetFocusProcess* is 1.55μs. System call *SetCpuSpeed* adjusts the frequency/supply voltage dynamically, based on the information on the user-perceived latency. Initially, the system works at a lower performance level to reduce power consumption. When the average latency is above $K_{min} = 100$ms and the current performance level is low, the X server notifies the kernel to increase the frequency/supply voltage. When the average is lower than $K_{min} = 50$ms and the current performance level is high, the X server notifies the kernel to decrease the frequency/supply voltage. *SetCpuSpeed* calls the functions provided by CPUFreq [27] to dynamically adjust the CPU frequency/supply voltage. The time overhead of *SetCpuSpeed* is 251.95μs.

5 Evaluation

We chose an IBM Thinkpad R32 laptop described in Section 3.1 for our experiments. The OS is RedHat Linux 9.0 with the 2.4.20 kernel (with the new O(1) scheduler [6]), with XFree86 4.3 and KDE desktop environment. The kernel is patched with 2.4.20-ac2, which enables CPU frequency/voltage scaling.

We use two metrics for evaluation: user-perceived latency and overhead. These are described next.

User-perceived latency: A user input is typically handled in a Linux/X computer using the following steps, as shown in Fig. 7.

1. An interrupt is generated as soon as the user presses a key or mouse button.
2. The interrupt handler in the keyboard device driver notifies the X server about the interrupt.
3. The X server reacts to the interrupt by generating an X event and sending it to the corresponding X client.
4. The X client processes the X event and sends a request for graphics processing to the X server.
5. The X server generates and sends updated display data to the monitor device driver.
6. The X server sends a reply to the X client and the latter returns to the event-loop to fetch new events.

We performed measurements as follows. For keyboard and mouse events, the X server records the time, relative to the X server start time, when an event is generated (birth time). We modified the X server slightly so that the birth time is recorded relative to the start of the day instead. The X11 library is used as the application programming interface (API) for X-based graphical user interface (GUI) applications. Most applications call $XNextEvent()$ directly or indirectly, when a GUI toolkit is used, for fetching the new X events sent by the X server. If there is no new event, $XNextEvent()$ blocks. Its restart marks the end of processing for the previous event and the return to event-loop. We added bookkeeping code to $XNextEvent()$ so that

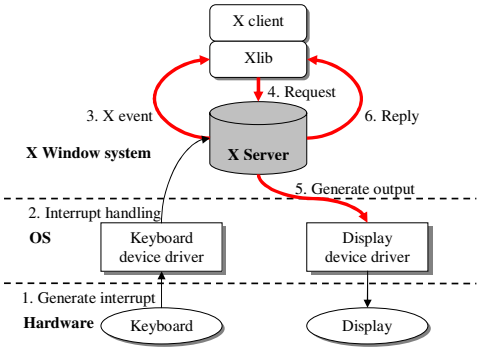


Fig. 7. User-perceived latency measurement via the X Window System

it records the difference between an X event's birth time and the time when the application reaches $XNextEvent()$ again. The difference is the user-perceived latency used in this work. It consists of steps (3)-(6). Based on our experience, the contribution of steps (1) and (2) is negligible compared to that of steps (3)-(6). The user-perceived latencies, and their average and distributions are used to evaluate computer responsiveness.

Overhead: Favoring the process under user focus may negatively impact the processes not under user focus when resource competition is tight. The completion time is used to evaluate compute-intensive applications, measured by the $time$ command provided by Linux. The scheduling latency is used to evaluate the impact on real-time applications. For a specific process, we record the time that it becomes runnable (when it calls $enqueue\_task()$). After each scheduling decision, the scheduler checks the process id, *pid*, being selected to run. If it is the target process, its corresponding scheduling latency is calculated.

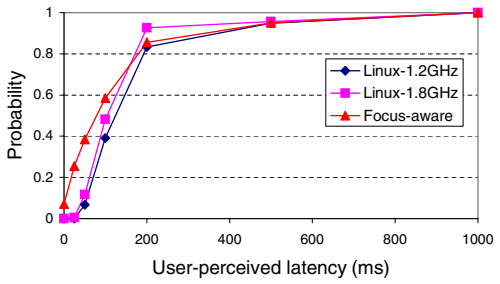
6 Experimental Results

We first present experimental results for real-world applications using artificial stressors to illustrate the effectiveness of focus-aware resource management in terms of computer responsiveness and energy efficiency. Next, we discuss how swiftly the user can switch focus using the focus-aware resource management technique. Finally, we discuss the overhead for the background applications. We use the term *focus-aware system* to refer to the Linux/X system augmented with focus-aware resource management, and $Linux$ to refer to the conventional Linux/X system.

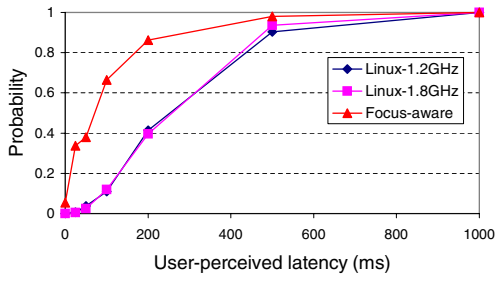
Real applications typically demand more than one hardware resource, resulting in very intricate resource competition. Therefore, to evaluate our focus-aware resource management technique for CPU, memory, and disk resource allocation, respectively, we wrote three artificial stressors to consume the CPU, memory, and disk, respectively. The CPU stressor runs a busy loop of double multiplication, consuming 99.8% of the CPU cycles on an average if running alone, as reported by top. Its memory consumption is negligible. The memory stressor allocates 50MB of memory and then runs a busy loop to set the whole allocated memory to a specific value using $memset()$. The disk stressor opens a synchronous file and repeatedly writes 10MB of data into it.

6.1 Computer Responsiveness and Energy Efficiency

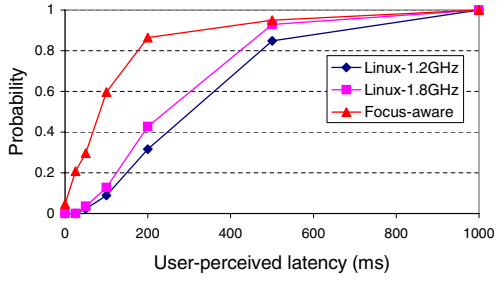
We consider an interactive application, $TuxRacer$, as being under user focus. Figs. 8(a)-(d) show the CDFs of user-perceived latency for $TuxRacer$ without a stressor, and under CPU, memory, and disk stressors, respectively. In each case, the CDF for the focus-aware system shifts towards shorter latencies, indicating an improvement in responsiveness. The trend towards shorter latencies in the distributions for the focus-aware system is even more dramatic under different stressors than without any stressor. In each case, Table 3 shows the reduction in user-perceived latency for $TuxRacer$ in the focus-aware system with respect to Linux running at 1.8GHz and 1.2GHz, respectively. It is obvious that focus-aware resource management improves responsiveness for interactive applications significantly during resource competition. The focus-aware system employed DVS so that the performance level could be lowered during the experiment. This implies an additional power advantage of the focus-aware system. We address this issue in detail later.



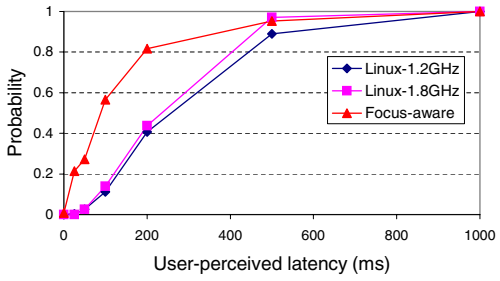
(a) No stressor



(b) CPU stressor



(c) Memory stressor



(d) Disk stressor

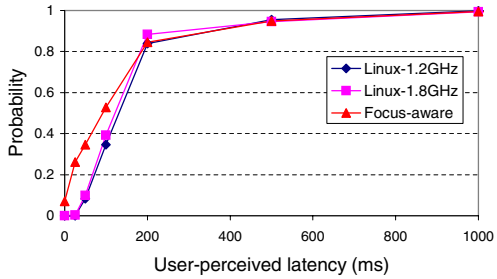
Fig. 8. *TuxRacer* under artificial stressors

Next, we used tar (a batch application), $MPlayer$ (a real-time application), and $GpsDrive$ (an interactive application) as stressors. tar is used to decompress the Linux

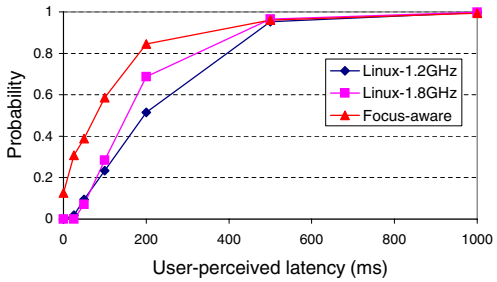
Table 3. User-perceived latency reduction for *TuxRacer* under artificial stressors.

Stressor	None (%)	CPU (%)	Memory (%)	Disk (%)
Linux-1.8GHz vs. Focus-aware	5.3	59.9	51.5	45.9
Linux-1.2GHz vs. Focus-aware	22.5	62.2	60.4	55.1

kernel source code. $MPlayer$ is used to play MP3 music. Fig. 9 shows the CDFs of the user-perceived latency for $TuxRacer$ in the tar-loaded and $GpsDrive$-loaded systems. Both distributions of the user-perceived latency in the focus-aware system shift towards shorter latencies compared to the Linux system running at 1.8GHz and 1.2GHz. In the tar-loaded focus-aware system, 47% of the user-perceived latency is above $K_{max} = 100$ms, as opposed to 61% in Linux running at 1.8GHz and 65% in Linux running at 1.2GHz. We obtain similar results in the $GpsDrive$-loaded focus-aware system. In this case, the user-perceived latency above $K_{max} = 100$ms in the focus-aware system is reduced by 42.0% with respect to Linux. The above two cases indicate that focus-aware resource management provides better system responsiveness compared to the Linux system. The X server performs DVS according to the user-perceived latency. It notifies the OS to change the processor performance level dynamically. We recorded the processor performance level after each user input was received. Fig. 10 shows these recorded traces for the experiments depicted in Fig. 9. We assume, in general, that each user input takes similar time and the processor is always busy, which is mostly true for $TuxRacer$. In the tar-loaded focus-aware system, for 21% of all user inputs, the processor was at the lower performance level. The average power consumption for focus-aware resource management can be estimated to be 28.1W. It yields an energy reduction of 6.3% compared to Linux. Similarly, we can estimate the energy reduction for the $GpsDrive$-loaded focus-aware system to be about 7.5% with respect to Linux. It is obvious that the focus-aware system can ensure better responsiveness even with less energy consumption.



(a) tar as stressor

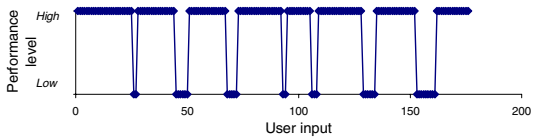


(b) $GpsDrive$ as stressor

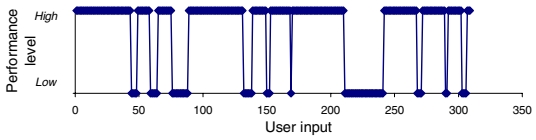
Fig. 9. *TuxRacer* under different stressors

6.2 User Focus Switch

One of the user concerns is how swiftly the focus can be switched. Fig. 11 shows the traces of user-perceived latencies for $KEdit$ and $TuxRacer$ for switching focus in both Linux and focus-aware systems. For $KEdit$ in Fig. 11(a), there are five focus switches from another $KEdit$. Immediately after a focus switch, user perceived-latencies increase in both systems. The average user-perceived latency after a focus switch is reduced from 12.4ms in Linux running at 1.8GHz to 5.9ms in the focus-aware system.

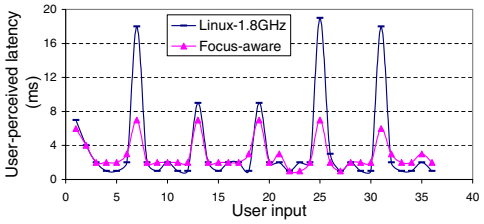


(a) tar as stressor

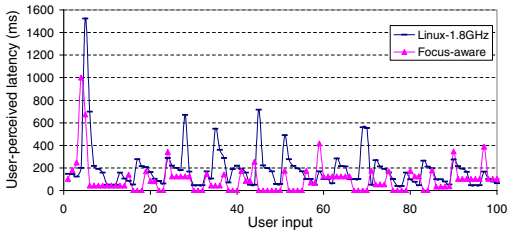


(b) $GpsDrive$ as stressor

Fig. 10. Performance-setting decisions for *TuxRacer*



(a) $KEdit$



(b) $TuxRacer$

Fig. 11. Effect of focus switch

In Fig. 11(b), the focus is switched from $GpsDrive$ to $TuxRacer$. The user-perceived latency for $TuxRacer$ after a focus switch is reduced from 147ms in Linux running at 1.8GHz to 104ms in the focus-aware system. The periodic behavior of the traces is due to $GpsDrive$ running in the background. It updates the map periodically, making $TuxRacer$ suffer. The focus-aware system performs better than Linux after the focus is switched.

6.3 Overhead

After observing the advantages of focus-aware resource management, we consider its impact on background applications. Table 4 shows the completion time of tar running in the background. It also shows the completion time for $FFmpeg$ and the user-perceived latency for $KEdit$ and $TuxRacer$. Note that when $FFmpeg$ finishes before tar, it restarts. The completion time is just that for the first run. The completion time of tar in the focus-aware system is longer than that in Linux when $FFmpeg$ or $TuxRacer$, which are compute-intensive, are under user focus. When $KEdit$ is under user focus, the overhead is small because there is a lot of idle time during its usage so that tar can still run very frequently. Although when the application under user focus is compute-intensive, the focus-aware system is not that sympathetic to the applications running in

the background, the system caters to user attention with all its resources, as is obvious from the reduction in the user-perceived latency/completion time (Table 4) for the applications under user focus. Moreover, by bringing an idle window under user focus, a user can make all background applications run in almost the same way as they do under Linux. We also experimented with the case when the user plays MP3 music in the background. Table 5 shows the scheduling latency of $MPlayer$, playing music in the background. The value of scheduling latency is still in the acceptable range in the focus-aware system.

Table 4. Overhead of focus-aware resource management

User focus application	Avg. user-perceived latency/completion time			
	$Focus$		Stressor (tar)	
	Linux (1.8GHz)	Focus-aware	Linux (1.8GHz)	Focus-aware
$FFmpeg$	36.1s	32.3s	116.6s	337.9s
$KEdit$	3.7ms	2.9ms	62.0s	63.0s
$TuxRacer$	141.5ms	118.9ms	59.2s	140.4s

Table 5. Scheduling latency of *MPlayer*

User focus application	Scheduling latency (ms) of $MPlayer$	
	Linux	Focus-aware
$FFmpeg$	1.2	2.7
$KEdit$	0.6	1.3

7 Discussions and Conclusions

The focus-aware resource management has a totally different design philosophy from Linux. The former values the user more than the computer while the latter inherits the opposite trait from its ancestors. However, our implementation on the Linux/X system requires only minimal changes and is fully based on existing process, memory, and disk I/O mechanisms in Linux. When there is no GUI application or the process under focus is idle, the focus-aware system behaves exactly in the same way as the Linux/X system.

In this work, we showed how computer responsiveness can be improved in the Linux/X system. By analyzing the characteristics of resource usage for interactive applications, we first discussed the factors that contribute to computer unresponsiveness, $e.g.$, resource competition for CPU, memory, disk, and graphics processing. We established the inadequacy of Linux in tackling computer unresponsiveness on loaded and power-managed systems. Based on the information on user focus, which indicates the interests and wishes of the users directly, we proposed and implemented a user focus-aware resource management technique based on the Linux/X system. According to the user focus information obtained from the X Window system, various resources are allocated to multiple applications by the X Window system and OS in a focus-aware fashion. By favoring the application currently under focus in different ways, we achieved a significant improvement in computer responsiveness, which in turn provided opportunities for reducing system power consumption through DVS without hurting user experience or productivity. The DVS policy was directly driven by the user-perceived latency. Experiments demonstrated that the focus-aware system can reduce the average user-perceived latency significantly on an IBM Thinkpad R32 laptop, with some reduction in energy consumption.

References

[1] T. W. Butler, "Computer response time and user performance," in *Proc. Conf. Human Factors in Computing Systems*, Dec. 1983, pp. 58–62.

[2] B. Shneiderman, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 265–285, Sept. 1984.

[3] D. P. Siewiorek, "New frontiers of application design," *Commun. ACM*, vol. 45, no. 12, pp. 79–82, Dec. 2002.

[4] Software bloat, http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?bloat.

[5] Focus-aware resource management kernel module, http://www.princeton.edu/~lyan/project.htm.

[6] I. Molnar, "Goals, design and implementation of the new ultra-scalable O(1) scheduler," Jan. 2002, http://kernel.kernelnotes.de/linux-2.6.2/Documentation/sched-design.txt.

[7] Y. Etsion, D. Tsafrir, and D. Feitelson, "Human-centered scheduling of interactive and multimedia applications on a loaded desktop," Tech. Rep., Hebrew University, Mar. 2003.

[8] TuxRacer, http://www.tuxracer.com.

[9] X Window system, http://www.x.org.

[10] K. Packard, "Efficiently scheduling X clients," in *Proc. Usenix Technical Conf.*, June 2000, pp. 175–186.

[11] S. Evans, K. Clarke, D. Singleton, and B. Smaalders, "Optimizing Unix resource scheduling for user interaction," in *Proc. USENIX Summer Tech. Conf.*, June 1993, pp. 205–218.

[12] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 1998, pp. 76–81.

[13] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic performance setting for dynamic voltage scaling," in *Proc. Int. Conf. Mobile Computing & Networking*, July 2001, pp. 260–271.

[14] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with PACE," in *Proc. ACM Int. Conf. Measurement & Modeling of Computer Systems*, June 2001, pp. 50–61.

[15] L. Zhong and N. K. Jha, "Dynamic power optimization of interactive systems," in *Proc. Int. Conf. VLSI Design*, Jan. 2004, pp. 1041–1047.

[16] L. Yan, L. Zhong, and N. K. Jha, "User-perceived latency driven voltage scaling for interactive applications," in *Proc. Design Automation Conf.*, June 2005, pp. 624–627.

[17] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd ed. Reading, Mass: Addison Wesley Longman, 1998.

[18] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-computer Interaction*. Lawrence Erlbaum Associate, 1983.

[19] E. Grochowski and R. Halem, "Technological impact of magnetic hard disk drives on storage systems," *IBM Systems Journal*, vol. 42, no. 2, pp. 338–346, 2003.

[20] Intel SpeedStep Technology, http://www.intel.com.

[21] XFree86, http://www.xfree86.org.

[22] Y. Endo and M. Seltzer, "Improving interactive performance using TIPME," in *Proc. ACM Int. Conf. Measurement & Modeling of Computer Systems*, June 2000, pp. 240–251.

[23] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *Proc. Symp. Operating Systems Principles*, Oct. 2001, pp. 117–130.

[24] Linux kernel source code, version 2.4.20, http://lxr.linux.no/source/drivers/block/elevator.c.

[25] Linux kernel source code, version 2.6.0, http://lxr.linux.no/source/drivers/block/deadline-iosched.c?v=2.6.0.

[26] Linux kernel source code, version 2.6.0, http://lxr.linux.no/source/drivers/block/as-iosched.c?v=2.6.0.

[27] CPUFreq, http://www.brodo.de/cpufreq/.