Automatic User Interaction Detection and Scheduling with RSIO

Haoqiang Zheng[1,2] *and Jason Nieh*[1]
hzheng@vmware.com, nieh@cs.columbia.edu
[1]*Columbia University*
[2]*VMware, Inc.*

Abstract

Response time is one of the most important factors for the overall usability of a computer system. We present RSIO, a processor scheduling framework for improving the response time of latency-sensitive applications by monitoring accesses to I/O channels and inferring when user interactions occur. RSIO provides a general mechanism for all user interactions, including direct interactions via local HCI devices such as mouse and keyboard, indirect interactions through middleware, and remote interactions through networks. It automatically and dynamically identifies processes involved in a user interaction and boosts their priorities at the time the interaction occurs to improve system response time. RSIO detects processes that directly handle a user interaction as well as those indirectly involved in processing the interaction, automatically accounting for dependencies and boosting their priorities accordingly. RSIO works with existing schedulers, processes that may mix interactive and batch activities, and requires no application modifications to identify periods of latency-sensitive application activity. We have implemented RSIO in Linux and measured its effectiveness on microbenchmarks and real applications. Our results show that RSIO is easy to use and can provide substantial improvements in system performance for latency-sensitive applications.

1 Introduction

Rapid advances in hardware technology have enabled computers to accumulate an increasingly wide range of uses and applications, including web surfing, playing movies, software development, email, telephony, document processing, and financial bookkeeping, among others. Recent trends in virtualization and server consolidation have expanded the number of applications with different resource requirements and quality-of-service demands being run on the same system. Furthermore, virtual desktop infrastructure, terminal services, and web-based office applications are just a few examples of desktop computing requirements extending beyond the desktop to servers as well.

Users expect computers not only to run many different applications, but to be able to run them all at the same time. A key challenge is how to ensure that the system provides acceptable interactive responsiveness to users while multiplexing resources among a diverse collection of applications. It is particularly important that activities which are more latency-sensitive receive acceptable response time from the system while sharing system resources with other activities.

Since processor scheduling determines when a process can run, system designers have long recognized that good scheduling mechanisms are essential to support the requirements of latency-sensitive applications. To achieve quick response time for latency-sensitive processes in a system shared with non-latency-sensitive processes, a common practice is to delay the execution of non-latency-sensitive processes in favor of latency-sensitive ones. However, identifying latency-sensitive processes is difficult for several reasons.

First, latency-sensitive applications used in modern computers have a wide range of functions and often have very different execution behavior. Traditional desktop office productivity tools have very different resource demands than multimedia applications. Multimedia applications have very different resource demands from e-commerce applications. All of these applications have latency-sensitive requirements. As a result, commonly used approaches in commodity operating systems which detect interactive latency-sensitive processes based on processor resource usage and sleeping behavior are generally ineffective across this broad range of applications [10, 18, 5, 19].

Second, latency-sensitive applications often involve human-computer interactions that occur in many different ways. An interactive latency-sensitive process may interact with users directly through local human-computer interaction (HCI) devices such as mice, keyboards, and audio/video devices. It may interact with users indirectly via middleware such as X Windows. It may also interact with users remotely via the network. Existing approaches in commodity operating systems only detect interactions through the window system by tracking input focus [14, 6, 5]. As a result, they are ineffective at identifying latency-sensitive applications across the broad range of interaction types commonly found on modern computers.

Third, human-computer interactions on modern com-

puters are often handled not just by one process, but by a collection of processes. For example, processing a typed character in Emacs on a Linux system requires not just the Emacs application, but the window manager and X server as well. To deliver fast response time, it is crucial for a system to identify dynamic dependencies among processes that arise in handling a latency-sensitive request and account for those relationships in scheduling processes. However, commodity operating systems provide little if any support for identifying such dependencies, much less mechanisms for using that information for scheduling latency-sensitive processes.

Finally, the notion of a "latency-sensitive" process is actually misleading because a process may switch between executing latency-sensitive activities and non-latency-sensitive activities dynamically during its life cycle. For example, MATLAB users first create programs interactively during which they expect good system responsiveness, and then execute those programs to process large amounts of numerical data during which they typically expect to wait a while for the programs to complete. The first phase is latency-sensitive; the second phase is not. Given this dynamic behavior, users cannot be expected to specify whether a process or an application is latency-sensitive. Furthermore, any mechanism that depends on the average behavior of such a process, such as using the average sleep versus run ratio [10] of a process, will be ineffective and miss transitions between non-latency-sensitive and latency-sensitive activity.

To address these problems, we introduce RSIO (stands for Response time Sensitive I/O), a processor scheduling framework for improving the response time of applications during periods of latency-sensitive activity. RSIO is based on the observation that latency-sensitive activities typically need to respond quickly to I/O involving user interactions, such as user input or certain kinds of output. As a result, RSIO uses user I/O activity to guide processor scheduling of processes with latency-sensitive requirements, in contrast to other approaches that only use processor activity for processor scheduling. Using RSIO, operations on I/O channels involving user interactions can be specified as being latency-sensitive. RSIO then automatically and dynamically identifies the processes that perform those operations on those I/O channels as latency-sensitive when those operations occur. Unlike other approaches, RSIO does not specify processes themselves as latency-sensitive, recognizing that processes may execute latency-sensitive and non-latency-sensitive activities at different times.

RSIO operates by directly monitoring accesses to I/O channels that reflect interactions between users and applications. RSIO can monitor any I/O channel, including those for direct HCI devices such as keyboard and mice, via middleware such as the window system, and remote I/O channels such as the network sockets. When RSIO detects an operation on a given I/O channel that should be considered latency-sensitive, it identifies the process or group of processes performing that operation. RSIO then prioritizes those processes ahead of other processes that are not performing latency-sensitive activities. If the processes performing those operations depend on other processes, RSIO correctly accounts for those dependencies to ensure that all processes involved in processing a latency-sensitive operation are prioritized at the right time. RSIO prioritizes processes in a manner dependent on and compatible with existing schedulers in commodity operating systems. For example, when used with a priority scheduler, RSIO can simply boost the priority value of a process to improve its response time.

We have implemented RSIO in Linux and measured its performance on various benchmarks and real-world applications. We show that RSIO is easy to use with unmodified applications and describe the simple ways in which a complete desktop environment can be configured to take advantage of RSIO's framework for improving interactive performance. We measure RSIO performance overhead and show that it is modest. We also compare the performance of RSIO versus a vanilla Linux system and demonstrate that RSIO can provide substantial improvements in system response time for a wide range of applications with latency-sensitive activities.

This paper presents the design, implementation, and evaluation of RSIO. Section 2 discusses related work. Section 3 describes the RSIO usage model. Section 4 describes how RSIO dynamically detects which processes are performing latency-sensitive activities and when they occur. Section 5 explains how RSIO uses that information for processor scheduling. Section 6 presents experimental results demonstrating the effectiveness of RSIO. Finally, we present some concluding remarks and directions for future work.

2 Related Work

Many approaches to processor scheduling have been considered for improving the performance of applications with latency-sensitive activities. Schedulers may use processes or threads as the schedulable entity. For simplicity and without loss of generality, we loosely refer to the schedulable entity as a process in this paper.

Perhaps the most common approach used in commodity operating systems is to schedule interactive applications based on their processor usage and sleeping behavior. While different heuristics have been used, they are all based on raising the priority of a process which has slept for a longer period of time or has not used up its time quantum before sleeping. For example, FreeBSD [8] uses a multilevel feedback queue scheduler in which pro-

cesses that block waiting for I/O for one or more seconds are given a higher priority. Alternatively, the Linux 2.6 processor scheduler [10] attempts to identify interactive processes as those that sleep longer and run less and gives them higher priority. Several studies [18, 5, 19] have indicated that this approach does not work well. Variations of these heuristics have been adopted for scheduling other resources, such as the window system [14], with similar limitations. The fundamental problem is that processor usage behavior alone is a often a poor indicator of interactivity given the resource intensive nature of many modern interactive applications.

Another approach widely used in commodity operating systems is to schedule using window system input focus. When using a GUI interface, users interact with the application window which has input focus. To improve system responsiveness, processor schedulers, such as those used in Solaris [15] and Windows [17], raise the priority of processes associated with a window that has input focus. Input focus has also been used for scheduling other system resources [19]. Using input focus can often work well, but it may also unintentionally raise the priority of non-interactive applications, for example, if the user leaves the mouse focus on a window running a compute-intensive batch application. More importantly, it does not work for applications that do not use the local GUI interface to interact with users, including console applications and applications that interact with remote users over a network. A key problem with using input focus is accurately tracking not just the process that receives input from the window system, but other processes involved in an interaction. This is not addressed in previous work [15, 6].

HuC [5, 6] introduces a novel approach to scheduling interactive and multimedia applications based on display output production. Processes are scheduled to equalize display output rates across windows, where the rate is based on the percentage of the window pixels that change per second. This can be useful for video applications, which results in all videos being displayed at the same frame rate regardless of window system. However, a key problem with this approach is that it results in undesirable behavior for mixes of interactive and non-interactive applications if the latter generate lots of display output.

Some approaches have focused on improving scheduling for interactive applications in the context of fair-share schedulers. SMART introduced the idea of a bias on fair scheduling to use the ability of batch processes to tolerate more latency to allow other latency-sensitive processes to run before them while preserving fair allocations [11]. However, the bias was set using processor usage behavior, which may not be a good indicator of the interactivity of a process. Borrowed-Virtual-Time scheduling uses the same idea [3], but requires users to specify appropriate bias values. RSIO is complementary to this work and can be used as a mechanism for determining how to dynamically adjust the bias of a process.

Several approaches to real-time scheduling have recognized that applications may have different latency requirements during different periods of application execution. For example, SMART [11] allows application developers to specify time constraints on sections of application code, which the scheduler then uses in ordering processes for execution. An application can have sections that are time-critical and sections that are not, enabling the scheduler to dynamically adjust the scheduling criteria for a process instead of treating the process with one set of static scheduling parameters. While this approach allows precise specification of time constraints on portions of code, it requires application modifications to do this. While RSIO does not focus on scheduling real-time applications, it also enables parts of an application execution to be treated as latency-sensitive. However, it does not require application modifications because it derives this behavior from I/O interactions.

While most related work has focused on the problem of scheduling once processes are assigned scheduling parameters, another key issue is how those parameters should be propagated correctly in the presence of process dependencies. Two areas in which this problem arises are priority inversion and coscheduling. For example, priority inheritance [7] is used to reduce priority inversion when a high priority process is blocked on a mutex resource by propagating the high priority value to other processes that need to run to unblock the process. SWAP [20] generalizes this work for dependencies due to other operating system resources and works for dynamic priorities. As another example, gang scheduling [13] is used for coscheduling by scheduling cooperating processes of a parallel application to run on different processors at the same time so they can communicate efficiently. Both priority inversion and coscheduling differ from the process dependency problem that RSIO addresses. Unlike the priority inversion problem, RSIO seeks to identify all processes involved in a user interaction, whether or not such processes are blocked or runnable. Unlike the coscheduling problem, RSIO supports cross-application dependencies and also deals with dependencies for uniprocessor scheduling. Furthermore, RSIO focuses on identifying cooperating processes, not addressing the complementary issue of which scheduling algorithm to use for running them.

3 RSIO Usage Model

RSIO is based on the observation that activities are often latency-sensitive because they are processing I/O due to human-computer interactions, and those activities are

more tied to the nature of the I/O than any particular process being executed. Furthermore, a process may engage in both latency-sensitive and non-latency-sensitive activities. In this context, specifying the priority or other scheduling parameters of a process may not be useful since how a process should be scheduled will change dynamically based on its I/O processing. Instead, RSIO provides a usage model based on allowing users and administrators to configure I/O channels. RSIO then automatically and dynamically derives the scheduling characteristics appropriate for processes based on their access and usage of those I/O channels.

RSIO provides a command to configure I/O channels as being latency-sensitive. The command, `rsio_config`, takes three types of parameters: channel, operation, and user. *Channel* specifies the I/O channel being configured. There are two types of channels, files and sockets. A file channel is a persistent entity in the file system that can be easily named. A socket is a dynamically created entity that is most easily named by referring to what the socket connects as opposed to the socket itself. If the channel is a file, it is identified by simply the filename. For example, if the I/O channel being identified is a mouse device, the channel name is the device name, `/dev/input/mice`. If the channel is a socket, it is identified both by the channel name and its creation operation. For example, if the channel refers to a network socket, the channel name is the destination hostname and port number and its creation operation can be `connect` or `accept`. It is easy to distinguish between file and socket channel types since only the latter includes a creation operation.

Operation specifies the I/O channel operation that a process performs to cause it to be flagged as a latency-sensitive process. The operation can be read, write, or both read and write. If the operation is a read, then any type of read operation performed by a process on the I/O channel will cause it to be flagged as a latency-sensitive process. If the operation is a write, then any type of write operation performed by a process on the I/O channel will cause it to be flagged as a latency-sensitive process. After performing a specified I/O channel operation, a process remains marked as latency-sensitive until RSIO determines that the relevant user interaction has completed.

User specifies for which user's processes the I/O channel should be considered latency-sensitive. By default, the channel is latency-sensitive for all users. If a specific user identifier is provided, a process will only be flagged as being latency-sensitive if it is owned by the given user and accesses the I/O channel.

To illustrate how easy it is to use RSIO, Figure 1 shows how a small number of RSIO configuration commands can be used to set up a default configuration of a system. For a Linux system, the startup script would go in `/etc/rc.local`. It sets up a system to use RSIO once the system is started using these commands.

```
---------------------------------
# tty devices
rsio_config READ /dev/tty0
rsio_config READ /dev/tty1
rsio_config READ /dev/tty2
rsio_config READ /dev/tty3
rsio_config READ /dev/tty4
rsio_config READ /dev/tty5
rsio_config READ /dev/tty6
rsio_config READ /dev/tty7

# mouse device
rsio_config READ /dev/input/mice

# audio device
rsio_config WRITE /dev/dsp

# network channels
rsio_config READ \
  CONNECT webproxy.columbia.edu:8080
rsio_config READ \
  ACCEPT mymachine.columbia.edu:22
---------------------------------
```

Figure 1: Default RSIO Configuration

The startup script configures four classes of I/O channels, TTY devices, the mouse device, the audio device, and network channels. TTY devices are terminal devices, including serial devices such as the original character-based terminals, and virtual terminals, which behave like character-based terminals from a programmer's perspective and are used by various applications such as the X window system for managing user input and display output. A system has a default set of TTY devices, which typically and in this case are represented by `/dev/tty0` to `/dev/tty7`. A successful read from a terminal device usually corresponds to user keyboard input. As a result, RSIO configures any read from a default TTY device as a latency-sensitive activity to improve system responsiveness to user keyboard input. Note that only terminal device reads are flagged as latency-sensitive, not writes. A write to a terminal device usually corresponds to application output to the display, but not all display output is the result of interactive activities. For example, a kernel compile generates lots of display output but is not latency-sensitive.

The mouse device `/dev/input/mice` handles all mouse events. A successful read from the mouse device corresponds to a process receiving mouse events. As a result, RSIO configures any read from a mouse device as a latency-sensitive activity to improve responsiveness to user mouse input. A system may have other user input

devices such as a joystick or gamepad. These devices can be treated in a similar way as the mouse device.

The audio device `/dev/dsp` is used for audio output. A successful write to the audio device results in audio output. Audio is latency-sensitive and delays in processing audio can result in audible clicks and degradation of audio quality. As a result, RSIO configures any write to the audio device as a latency-sensitive activity to ensure good audio quality. Note that a write operation typically occurs after a particular audio sample has been processed, so flagging a process as being latency-sensitive after the write occurs does not help with processing the just-written audio sample. However, audio processing is typically periodic and repetitive in nature, so all subsequent processing of audio samples will be handled in a latency-sensitive manner.

It is worth noting that this RSIO setup configures audio output to be latency-sensitive, but does not configure any display output to be latency-sensitive. Any approach that flags processes that generate display output as latency-sensitive is problematic because many common applications that are not latency-sensitive can generate lots of display output, a kernel compile being just one such example. On the other hand, many multimedia applications also generate lots of display output and should be considered latency-sensitive. Furthermore, multimedia applications can generate long periods of display output without any user input, so simply monitoring keyboard or mouse input does not help. RSIO addresses this issue by observing that applications that do not require much user input, generate display output, and are latency-sensitive usually also generate audio output. For example, movie players generate both video and sound. On the other hand, non-latency-sensitive applications such as kernel compilation do not generate sound. As a result, RSIO automatically delineates between these two classes of applications by monitoring audio output instead of display output.

Network channels are used for handling various kinds of remote interactions. This example setup shows two for illustrative purposes. The first network channel is an outgoing connection to a web proxy, `webproxy.columbia.edu`, at port number 8080. A read on this channel corresponds to the local machine receiving data from a web server. As a result RSIO configures any read on the network channel as a latency-sensitive activity to improve system responsiveness when processing a web page download so that web pages are displayed faster. Note that a web page download could instead be marked as latency-sensitive based on mouse or keyboard input, but this RSIO configuration enables faster web performance even in the absence of such inputs, such as for a scripted web page download. The second network channel is an incoming connection to the local machine's port 22 where the SSH daemon is listening for connections. A read on this channel corresponds to the local machine receiving data from a remote SSH client, which typically corresponds to user input. As a result RSIO configures any read on the network channel as latency-sensitive to improve system responsiveness when the user is remotely connected to the system.

The end result of this startup script is a set of RSIO I/O channels that can be used to capture many latency-sensitive activities in a standard desktop computer system. The configuration of TTY devices and the mouse device effectively provide a mechanism to track input focus and treat processes receiving user input as being latency-sensitive only when such input is occurring. The configuration of the audio device enables audio applications and multimedia applications to be treated as latency-sensitive. The configuration of the network channels enable web applications and remote access applications to be treated as latency-sensitive during periods of user interaction. Furthermore, the user is not required to identify any application processes, or set and tune any additional parameters such as shares or priorities. This simple yet powerful usage model provides flexibility and functionality not available with other process-centric approaches.

These examples illustrate how RSIO enables users to simply configure a small number of I/O channels to completely configure a system to use RSIO. Note that RSIO is intended to be used to configure I/O channels that are directly used by local or remote users. This is easy to do because the number of such I/O channels is limited, they are mostly created when the system is booted, and the latency-sensitivity of these channels is easy to determine. There are many other I/O channels that are indirectly used by users, such as IPC communication channels. Users are not expected to manipulate those channels. Instead, RSIO automatically handles those indirect channels in a manner described below.

4 Latency-sensitive Process Detection

RSIO maintains some additional system state to identify RSIO I/O channels and processes involved in user interactions using those channels. Table 1 summarizes the RSIO system state. These parameters and objects are discussed in further detail below.

4.1 RSIO I/O Channel Instantiation

`rsio_config` causes the instantiation of a RSIO I/O channel. RSIO represents an I/O channel using an *rchannel*. As shown in Table 1, an rchannel consists of three components: an access type, a user identifier, and a list of handlers, which will be described in more detail below.

State	Fields / Description
rchannel	access type, uid, handler list
handler	id, process, access type, access time, expiration time confidence, cohandler list
max_conf	maximum confidence value
co_conf	cohandler confidence value
sys_expire	system expiration time
co_delta	cohandler time window
reader	IPC reader process
writer	IPC writer process

Table 1: RSIO System State

In a Unix-style system, this additional state is associated with the in-memory `inode` structure, which is used to represent I/O channels. The state is created and deleted as part of `inode` creation and deletion. At creation, the access type is blank, the user identifier is zero, and the list of handlers is empty. RSIO state initialized in this way has no effect on the behavior of the system. The RSIO state only affects system behavior after the RSIO state is configured by a configuration command.

The configuration of RSIO state is somewhat different for the two types of I/O channels, files and sockets. For file I/O channels, such as TTY devices, the corresponding `inode` is created when the system boots and therefore exists by the time that a `rsio_config` command is executed. When the configuration command executes, RSIO simply finds the already created `inode` and configures its associated RSIO state. The access type is set to read, write, or read-write according to whether the read or write operations are used to activate this channel, and the user identifier is set based on the user field of the configuration command. This is implemented using the `ioctl` system call.

For socket I/O channels, a corresponding `inode` is also eventually created, but it is not created when the system boots and therefore usually does not exist by the time that a `rsio_config` command is executed. Instead, the `inode` is created at later time when the socket is actually created and used. To deal with this dynamic state, RSIO defers the execution of RSIO configuration commands on sockets and keeps a list of such commands. It then monitors socket creation system calls such as `connect` and `accept` and checks if any such creations matches with a deferred RSIO command. For example, if a RSIO command was deferred that is for accepting connections to the SSH local port 22, RSIO will monitor `accept` system calls and check if any such calls are for port 22. If such a system call is found, RSIO then identifies the corresponding `inode` created by the system call and updates its RSIO state in the same manner as discussed earlier for file I/O channels. Note that deferred RSIO commands remain in the deferred list since matching sockets may be created at any time and each such creation requires RSIO to update the respective `inode` state.

4.2 RSIO I/O Channel Activation

Given a set of RSIO-configured I/O channels, RSIO needs to identify user interactions on those channels and the processes involved in those interactions. For most types of I/O, an application cannot communicate with users directly, but instead does so through the operating system via system calls. RSIO therefore monitors relevant system calls that access RSIO configured I/O channels to detect such human-computer interactions.

RSIO monitors read and write operations that occur through system calls to detect the start of a human-computer interaction. Read operations include not only `read` system calls, but also system calls such as `readv` and `recvmsg`. Similarly, write operations include not only `write` system calls, but also system calls such as `writev` and `sendmsg`. RSIO instruments each of these system calls. When one of these system calls is performed, RSIO uses the system call arguments, specifically the file descriptor, to obtain the corresponding `inode` and check its RSIO state. If both the read and write flags are not set, the I/O channel has not been configured as latency-sensitive and no further action is taken.

If the access type is read and a read operation is performed, or the access type is write and a write operation is performed, RSIO checks if the user identifier of the calling process matches the RSIO state user identifier. If it matches, RSIO considers this system call as the start of a user interaction on a RSIO I/O channel. The calling process is referred to as a *primary handler* for this interaction and RSIO activates the process so it is considered as being latency-sensitive. Section 5 describes how latency-sensitive processes are scheduled. Note that RSIO performs its monitoring after the actual system call has successfully read or written I/O since there is no need to perform any action if the operation was not successful.

RSIO currently only handles I/O through read and write operations. It does not support user interactions through memory mapped I/O channels. However, in our experience, this is sufficient for most I/O channels of interest. For example, while memory mapped file systems are not uncommon, those forms of I/O are not typically user interactions. Perhaps the most common instance of memory mapped I/O that does involve user interactions is through the display device. However, as discussed in Section 3, RSIO does not typically treat that I/O channel as latency-sensitive since it is also commonly used by non-latency-sensitive activities.

RSIO introduces a *handler* to maintain state associ-

ated with a process that is a primary handler. As mentioned in Section 4.1, RSIO maintains a handler list for each RSIO I/O channel. Whenever an interaction happens on a RSIO channel, RSIO checks the handler list of the channel to see if the calling process is already in the handler list. If the handler does not exist, a new handler object is created and inserted into the list. As shown in Table 1, a handler consists of seven components: an id, a reference to the associated process, an access type, an access time, an expiration time, a confidence value, and a cohandler list. For a primary handler, the id is the inode identifier of the corresponding I/O channel, the access type is the access type of the I/O channel, and the access time is the last time the process accessed the I/O channel. The access type is not strictly necessary, but is stored as part of the handler to avoid having to go back to the corresponding rchannel to look it up. A process is considered to have accessed a RSIO I/O channel if it performed a read or write operation and the channel's access type matches the operation. The cohandler list is initially empty. The expiration time, confidence value, and cohandler list are described in further detail below. Note that a process may access multiple RSIO I/O channels and hence may have multiple handlers associated with it. Handlers for a process will be deleted and removed from all handler lists when the process exits.

After a primary handler is activated, RSIO needs to determine when the handler should be deactivated and no longer considered latency-sensitive. An interaction usually finishes when an application outputs the resulting response to the user. RSIO could detect such output by monitoring I/O channels for this purpose. However, this would require users to specify which I/O channels should be considered for user output, as output to I/O channels such as disk should usually not be considered as the end of a user interaction. Even if the user specifies which I/O channels to monitor for user output, it is generally difficult to know which output is the last one. An application may generate a sequence of outputs in response to an interaction, and it would be desirable to maintain the processes involved in that interaction until the output to the user is complete. As a result, the additional complexity involved in monitoring output may not provide much benefit given the uncertainty in determining when the output is complete. Note that this problem does not occur for determining the start of a user interaction since it is easy to determine and desirable to use the first I/O for that purpose.

RSIO approaches the problem of determining when a handler should be deactivated from a different angle. We observe that an interaction between an application and a user typically continues for some period of time until one of two things happens. First, the user could switch from interacting with one application to interacting with another. Second, the user could simply stop interacting with the computer. RSIO uses a confidence model to address the first case, and a timeout model to address the second case.

RSIO detects when an interaction ends due to a user switching interaction to another process using a confidence model. As shown in Table 1, each handler includes a field called *confidence*. The confidence value is used to indicate how confident RSIO is that the given handler is still involved in a user interaction. If a new handler is created due to an interaction on a RSIO channel, its confidence value is initialized to one. If an interaction occurs and the handler already exists due to a previous interaction, its confidence value is incremented by one. For all other handlers in the handler list of the RSIO I/O channel, their confidence values are each decremented by one since they are not involved in the current interaction. Confidence values start at zero and can be incremented up to *max_conf*, the maximum allowable confidence value. max_conf is configurable and is five by default. If the confidence of a handler is decremented to zero, the handler will be deactivated and no longer considered latency-sensitive. Since a word is typically assumed to be five to six characters on average [2, 1], this default value of max_conf deactivates a process by the time a user has typed one word worth of user interactions into another process. A handler is deleted if its confidence is zero and its cohandler list is empty. Cohandlers will be discussed further in Section 4.3.

For example, if a user switches from interacting with process A to process B, process B will become latency-sensitive immediately. On each further interaction with process B, process A's confidence value will drop by one. And after a number of interactions no more than the maximum confidence value, process A will be deactivated because its confidence has dropped to zero. This mechanism enables RSIO to detect the end of an interaction due to a user switching to interact with another process.

RSIO determines when a user has stopped interacting with the computer system by using a simple timeout model. If a user stops interacting with the computer, there is no easy way to determine when that occurs. As a result, RSIO associates a timeout with each handler for this purpose, which is its expiration time, as shown in Table 1. When a handler is activated, RSIO assigns it an expiration time. A handler will be deactivated if that process does not access a RSIO I/O channel before its expiration time. RSIO assigns the expiration time by using the handler's access time and adding to it a system expiration time, *sys_expire*. In other words, if a handler is activated at time t, it will expire at time t + sys_expire. sys_expire is configurable and is 2 seconds by default. This default was selected based on previous research indicating a 2 second response time limit for simple com-

mands [16, 12].

4.3 Dependencies and Cohandlers

While the start of an interaction through a RSIO I/O channel is caused by one calling process and therefore one primary handler, multiple processes may be involved in the processing required for such an interaction. If only the primary handler is treated as latency-sensitive, it may block waiting for another process that is involved indirectly in the user interaction, resulting in a form of priority inversion. Even if the primary handler does not block, other processes involved in the user interaction may be in the critical path. If they are not treated as being latency-sensitive, they can be delayed in being scheduled, resulting in degraded system responsiveness. Unfortunately, while it is easy to determine the primary handler for a user interaction, there is no general way to precisely determine what other processes the primary handler may depend upon in processing the interaction.

For example, consider a user typing into a text editor such as Emacs on a Linux system running X Windows. Keyboard input occurs through a TTY device, which is read by the X server. The X server then communicates with Emacs to pass along the keyboard input. Since the X server process reads the I/O channel, it is the primary handler of the interaction. However, Emacs is the application actually doing the semantically interesting processing of the keyboard input. If only the primary handler is treated as latency-sensitive, system response time may suffer because Emacs is also indirectly processing the user input and is therefore latency-sensitive. In this case, the X server will not block on Emacs, but Emacs is in the critical path for generating a response. Furthermore, the operating system has direct knowledge that Emacs is involved in the user interaction. In this particular case, the X server may be able to obtain this information, but this is application-specific and does not generalize to other non-X interactions. For example, if the Emacs process then depends on another process to handle the user interaction, the X server will not be able to help with determining those dependencies.

RSIO automatically detects what processes a primary handler depends upon by using a simple heuristic that works quite well in practice. If a primary handler depends on another process for handling a user interaction, we observe that it is very likely for those processes to communicate within a short period of time of when the user I/O occurs. We refer to the user I/O as an activation event, since it activates a process as a primary handler. By monitoring the time proximity of activation events and interprocess communication events, RSIO can detect the processes indirectly involved in handling a user I/O interaction without needing to know any application-specific details.

RSIO refers to a process that a primary handler depends upon as a *cohandler*. RSIO reuses the *handler* object discussed in Section 4.2 in a different way to maintain state associated with a cohandler. RSIO considers a process A as a possible cohandler for a process B if process A communicates with process B after a handler has been created for process B, and using the same access type as process B's handler. Recall that a process will have a handler object if it has been previously activated as a primary handler. As shown in Table 1, RSIO maintains a cohandler list for each handler. Whenever a process A communicates with another process B that has an associated handler with a corresponding access type, RSIO checks the cohandler list of the handler to see if the process A is already in the cohandler list. If the cohandler does not exist, a new handler object is created and inserted into the list. For simplicity, we first assume that all communications between processes match the respective access type, and defer a discussion until Section 4.4 of how to determine whether a communication is the right access type.

The seven fields of the handler object are initialized in a different way in the case of a cohandler versus a primary handler. For a cohandler, the `id` is the process identifier of the corresponding primary handler, the `access type` is the access type of the primary handler, the `access time` is the last time the cohandler process communicated with the primary handler, the `expiration time` is set equal to the expiration time of the primary handler, the confidence value is initialized to zero, and the cohandler list is initially empty. Note that a process may serve as a cohandler for multiple other processes, and hence may have multiple cohandler handler objects associated with it. A process will be removed from all cohandler lists when it exits. Since a process generally does not communicate with many other processes, the cohandler lists are typically short in practice.

Processes in a cohandler list are just potential cohandlers. RSIO uses a confidence model to decide whether a potential cohandler is an actual cohandler or not. Whenever a handler is activated by accessing a RSIO I/O channel and its cohandler list is not empty, RSIO instantiates a callback to occur after a time period T to adjust the confidence of the cohandlers. Suppose a handler for process A has been activated at time `t1`. If process B communicates with process A during the time period `t1 - T` to `t1 + T`, the callback increments the confidence value of process B by one. For all other cohandlers, the callback decrements the confidence by one. A cohandler B is activated and treated as an actual cohandler if its confidence is larger than a confidence threshold *co_conf*, which is listed in Table 1. co_conf is configurable and is two by default to set the confidence threshold to be one

more than what is used for activating a primary handler. The range of confidence values is limited by max_conf just as for the primary handlers. The value of T is set by the parameter *co_delta* in Table 1. It is configurable and is 5 ms by default.

Figure 2 presents an example to illustrate the cohandler confidence model for a process `p`. From time `t1` to `t11`, process `p` was activated for 5 times, which happened at time `t2`, `t4`, `t6`, `t8` and `t11`, respectively. Suppose process `p` has two potential cohandlers `p1` and `p2` in its cohandler list with initial confidence values of zero. For interaction `a1` at time `t2`, `p1` has one interprocess communication with `p` at time `t1` which is within `T` ms of `t2`, so RSIO increases `p1`'s confidence by 1. The confidence of `p2` remains as 0. For interaction `a2`, process `p1` has another interprocess communication with `p` within the expected threshold while `p2` has no such communications, so the confidence values of `(p1, p2)` are adjusted to be `(2, 0)`. For interaction `a3`, `p2` has an interprocess communication within the expected threshold while `p1` does not, so the confidence values of `(p1, p2)` are adjusted to be `(1, 1)`. `p2` continues to communicate with `p` within the expected threshold for activation events `a4` and `a5` while `p1` does not, so at the end of `a5`, the confidence values of `(p1, p2)` are adjusted to be `(1, 3)`. At this point, `p2` is treated as an activate cohandler while `p1` is not. The example shows how RSIO automatically detects the cohandler transition from `p1` to `p2` based on its confidence model.

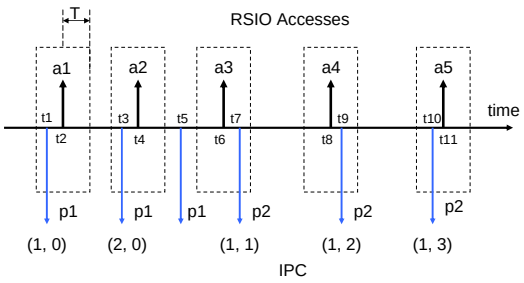

Figure 2: Cohandler Detection

A cohandler may communicate with other processes that should also be considered as latency-sensitive. Each cohandler has its own cohandler list. Since each cohandler process has an associated handler object, RSIO can recursively identify potential cohandlers of cohandlers in the same manner it identifies cohandlers of primary handlers.

When a primary handler or cohandler process forks a new process, the process creation is treated by RSIO as a form of communication between the child and parent processes. As a result, RSIO identifies the child process has a potential cohandler and is added as a new cohandler to the cohandler list of the parent process. RSIO's confidence model is again used to activate or deactivate the child process as an actual cohandler based on resulting interprocess communications while the parent is processing a user interaction.

4.4 Interprocess Communication Detection

To detect cohandlers, RSIO needs to monitor interprocess communications and determine a notion of access type for them. In a UNIX style system, processes can communicate or synchronize with each other using various mechanisms, including pipes, sockets, pseudo terminals, signals, futexes, IPC semaphores, file locks, etc. RSIO monitors interprocess communications that are commonly used for data communications to capture communications that are used for passing data related to user interactions. For this purpose, RSIO monitors three types of interprocess communication mechanisms: pipes, sockets, and pseudo terminals, the latter being widely used by X window applications. Other mechanisms are ignored because they are mostly used for synchronization, instead of communication.

For pipes, sockets and pseudo terminals, RSIO needs to determine the processes involved in an interprocess communication using these mechanisms. It is easy to determine one of the processes involved using these mechanisms by monitoring the system calls that use these mechanisms and determining the calling process of the system call. However, operating systems typically do not provide a way to track the process involved in the other end of such a communication at the time of the system call. Since pipes, sockets, and pseudo terminals are all represented as `inodes` in the kernel, RSIO associates two additional fields with each `inode` to track processes at both ends of an interprocess communications.

As listed in Table 1, these two fields are a `reader` field and a `writer` field. Both fields are initially NULL when the `inode` is created. Whenever a process successfully accesses an `inode` using a read system call, the `reader` field is updated to reference the calling process. Similarly, whenever a process successfully accesses an `inode` using a write system call, the `writer` field is updated to reference the calling process. The latest reader and writer is thus stored for each `inode`. The `reader` and `writer` fields are reset to NULL when the respective process closes the file descriptor corresponding to this inode.

Whenever an interprocess communication of interest occurs, RSIO identifies the calling process of the system call and uses the reader and writer fields to determine the other process involved. If one of the processes has an associated handler because it is a primary handler or cohandler, RSIO checks the access type of the handler

to see if it matches the interprocess communication. For example, if the calling process performs a read system call to communicate with a process that has an associated handler with a read access type, RSIO considers this a match. Similarly, if the calling process performs a write system call to communicate with a process that has an associated handler with a write access type, RSIO considers this a match. If a match occurs, RSIO proceeds with the cohandler creation and update mechanism discussed in Section 4.3.

5 RSIO Scheduling

RSIO is a general mechanism that dynamically detects whether processes are latency-sensitive by identifying all activated primary handlers and cohandlers. This information can be used by any processor scheduler to improve the responsiveness of a system. For example, a priority scheduler could use this information to boost the priority of processes that have been marked latency-sensitive. As another example, a fair-share scheduler could use this information to increase the shares of processes that have been marked latency-sensitive. Alternatively, a multi-level feedback queue scheduler could use a separate queue for processes that are marked latency-sensitive and schedule processes from this queue ahead of other queues. To illustrate further how RSIO can be used in commodity operating systems for scheduling, we describe one way in which RSIO can be used with the Linux processor scheduler to improve the responsiveness of a Linux system.

Linux uses a priority-based processor scheduler which dynamically adjusts the priorities of processes based on process usage and sleeping behavior. In Linux 2.6, a process's dynamic priority is decided by two components, its nice value and a dynamically computed priority bonus. The nice value is specified by the user and has a range of [-20, 19]. A smaller nice value is translated into a higher priority. The priority bonus has a range of [-5, 5]. A process's priority bonus is decided by its `sleep_avg`, which represents the sleep versus run ratio of this process. Based on assumption that an interactive process often spends much of its time sleeping, the Linux 2.6 scheduler gives more priority bonus to processes that have a larger `sleep_avg`. Processes with more priority bonus will have higher priority and should have better response time. However, previous work has shown that predicting a process's interactiveness based on its `sleep_avg` has various limitations [18, 5, 19].

Using RSIO, we change the way Linux computes a process's dynamic priority to take advantage of RSIO's ability to more accurately determine when processes are performing latency-sensitive activities and need better response time. We still use the same algorithm for computing a process's dynamic priority based on its nice value and a priority bonus. However, the priority bonus is instead determined simply based on whether RSIO has indicated that the given process is latency-sensitive. By default, a process is simply assigned a priority bonus of 0. If a process becomes an activated handler or cohandler for a RSIO I/O channel, it is assigned a priority bonus of 10 until it is deactivated. This maintains the same dynamic range of priority values as used by the default Linux scheduler, but adjusts priority values within that range in a manner that more accurately reflects when processes are latency-sensitive. This change in behavior is very simple and requires changing only a few lines of code in the Linux processor scheduler.

6 Experimental Results

We have implemented a RSIO prototype in the Linux 2.6.19 kernel and modified the Linux scheduler to use RSIO in the manner described in Section 5. To demonstrate its effectiveness, we compare the performance of RSIO versus vanilla Linux on several micro-benchmarks and real-world interactive applications. We used application workloads that represent a wide-range of different usage scenarios, including (1) running a mix of interactive and non-interactive applications from a local console, (2) using a technical computing tool similar to MATLAB which has periods of interactive use and background number crunching calculations, (3) web browsing on a loaded machine, (4) multimedia video playback on a loaded machine, and (5) supporting multiple remote users engaged in periods of interactivity and long-running computations. We also measure the performance overhead of RSIO versus vanilla Linux. For RSIO, we set up the system with default parameters and configured I/O channels using the simple system configuration script shown in Figure 1 and discussed in Section 3.

For most of our workloads, we measure response time to quantify system performance. In particular, users often care about when the system is responding poorly as opposed to just average response time. In fact, users are typically unhappy with the responsiveness of a system if it has good average response time but unexpectedly long delays in system responsiveness some of the time. To capture this notion, we report our results in terms of both the 90th percentile response time performance and the worst case response time performance.

The machine used for all our measurements is an HP xw9300 PC with a 2.6 GHZ AMD Opteron processor and 2 GB RAM. The server was running Ubuntu 6.06, and the kernel used was Linux 2.6.19.

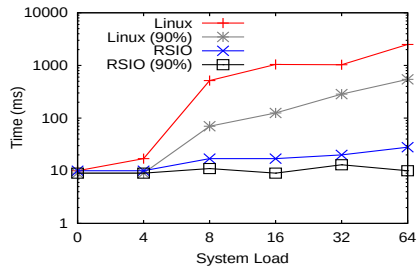

Figure 3: Active Console Benchmark

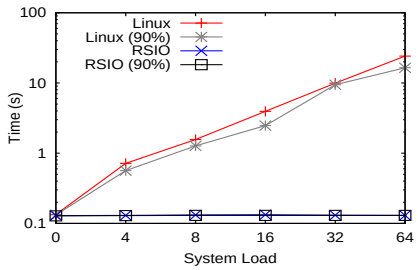

Figure 4: Octave Benchmark

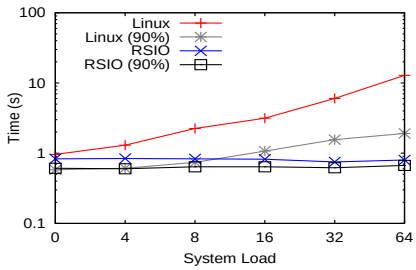

Figure 5: Web Browsing Benchmark

6.1 Active Console Benchmark

The first scenario represents an active console in which a local user is using a window system with multiple console windows open, one of which the user is actively using by typing and executing simple commands. Other windows are being used to run non-latency-sensitive batch jobs. For the user to receive good system response time, the console window that the user is actively using, along with its associated commands being executed, should be detected as performing latency-sensitive activities. Other batch jobs should be treated by the system as non-latency-sensitive activities.

For this scenario, we constructed an active console benchmark consisting of two GNOME terminal windows. GNOME terminal is a pseudo terminal application that allows users of the GNOME Linux desktop environment to execute commands using a UNIX style shell environment. In one terminal, a Linux kernel compilation is executed, which is a long running batch job. In the other terminal, a user types at the command prompt "time ls" to execute the command to list the contents of a directory and time its execution. Once the command completes, the user repeats the same typing and command execution. The user repeated the command fifty times, and we measured the elapsed time for executing the command each time to quantify the response time of the system. We also varied the load on the system due to the Linux kernel compilation by allow the compilation to be done in parallel with different numbers of processes. This was done using the `-j` option to specify the number of kernel compile processes to be generated. For example, we use the command `make -j 4` to start the Linux kernel compile with 4 concurrent processes.

Figure 3 compares the response time of Linux versus RSIO using the active console benchmark under different system loads. We varied the system load imposed by the kernel compilation from no load when no kernel compilation processes were run, to allowing 64 concurrent processes to run to perform the kernel compilation. When running without any background kernel compilation workload, Linux and RSIO provide the same response time of 10 ms for the interactive directory listing command. However, as the load on the system increases, the response time of vanilla Linux increases dramatically. When 64 kernel compilation processes were running, the worst response time for Linux was 2.5 s, which is a significant and noticeable delay for interactive activities and makes typing and executing interactive commands very unpleasant. Similarly, the 90th percentile response time for Linux was over .5 s. The response time is four times longer than the 100 ms response time threshold [12] for having users feel that the system is reacting instantaneously.

In contrast, RSIO correctly identifies the active console since keyboard input through the TTY device is configured as a RSIO I/O channel, and any processes involved in reading that keyboard input are marked as latency-sensitive. As a result, the worst case response time of RSIO is 28 ms even with 64 kernel compilation processes running. This is a bit worse than the 10 ms response time in the low load case, but almost an order of magnitude better than Linux. The small performance degradation is mostly caused by I/O contention since both the background load and the interactive commands exercise the file system. Furthermore, the 90th percentile response time of RSIO is almost independent of background load. The worst case and 90th percentile response times of RSIO are well below the response time threshold at which users can detect any response time delays.

6.2 Octave Benchmark

The second scenario represents a remote user running an application similar to MATLAB that has both latency-sensitive and non-latency-sensitive phases while other remote users are running other batch jobs. The latency-sensitive phase corresponds to frequent user interactions. The non-latency-sensitive phase corresponds to long running batch processing. For the user to receive good system response time, the latency-sensitive phases of the application should be detected when they occur. Other non-latency-sensitive phases of the application and other batch jobs should be treated by the system as non-

latency-sensitive activities.

For this scenario, we constructed an octave benchmark consisting of two SSH sessions representing two different users connected to a server over the network. One user is running a kernel compilation in the same manner as discussed for the active console benchmark, but over an SSH session instead of using a local GNOME terminal. This is used to represent batch processing activity. The other user is running Octave [4], a MATLAB-like application that involves phases of interactive use and long-running computations. In particular, the user runs Octave by typing two sets of commands in the following order:

```
tic; load A.dat; toc; (1)
for i=1:1000; X=A\A; end; (2)
```

The first set of commands consists of some timing commands and obtaining input data. The commands "tic" and "toc" are used to report the elapsed time of the command executed between these two commands. The command "load A.dat" loads data from a local file "A.dat" to create a 200x200 two dimensional array. The second set of commands is an iterative loop that performs a set of long running computations on the two-dimensional array. The first set of commands represent a user interacting with the application to set up a computation to run and should be considered as latency-sensitive. The second set of commands represent the long running computation itself and should not be considered as latency-sensitive. Once the two sets of commands complete, the user repeats the same typing and command execution. The user repeated the commands ten times, and we measured the elapsed time for executing the interactive set of commands each time to quantify the response time of the system.

Figure 4 compares the response time of Linux versus RSIO using the Octave benchmark under different system loads. We varied the system load imposed by the kernel compilation from no load when no kernel compilation processes were run, to allowing 64 concurrent processes to run to perform the kernel compilation. When running without any background kernel compilation workload, Linux and RSIO provide the same response time of 129 ms for the interactive phase of the Octave benchmark. However, as the load on the system increases, the response time of vanilla Linux increases dramatically. When 64 kernel compilation processes were running, the worst response time for Linux ballooned to 24.15 s, resulting in a completely unacceptable delay of almost half a minute during the interactive phase of the benchmark. Even the 90th percentile response time is XXX, which is also an unacceptable delay. The performance is horrible because the Octave benchmark does not sleep much, and thus the Linux scheduler mistakenly always considers the benchmark as a non-interactive process since it only uses processor usage and sleep behavior to determine interactivity.

In contrast, RSIO correctly identifies the SSH session running the Octave benchmark as latency-sensitive when it is receiving user input since the network channel for SSH is configured as a RSIO I/O channel. As a result, the worst case response time of RSIO is 130 ms even with 64 kernel compilation processes running. This is essentially identical to the response time for the benchmark in the low load case, and is more than two orders of magnitude better than Linux. The worst case response time of RSIO is independent of background load across the range of system load considered. Furthermore, the response time of RSIO is not much more than the response time threshold at which users can detect any response time delays. RSIO achieves this response time performance even though users are connected to the machine remotely, demonstrating that RSIO can automatically detect user interactions via SSH connections to identify latency-sensitive phases of an application.

6.3 Web Browsing Benchmark

The third scenario represents a local user running a web browsing application and downloading various web pages while other batch jobs are running. For the local user to receive good system response time, the web browser that the user is using should be detected as performing latency-sensitive activities. Other batch jobs should be treated by the system as non-latency-sensitive activities.

For this scenario, we constructed a web browsing benchmark consisting of the Mozilla Firefox web browser visiting a locally stored web page with two frames. One frame runs a JavaScript program that controls the reloading of web pages in the other frame. The JavaScript program causes the other frame to reload "http://news.google.com" repeatedly for five minutes. This web page provides current news articles and is frequently updated with different content. Each page reload is done five seconds after the previous reload completes, providing the user some time to view the contents of the web page before reloading a new version. The JavaScript program also reports the elapsed time from sending the HTTP request until the web page is completely reloaded. While the web browsing activity is occurring, another user is remotely connected to the same machine and running a kernel compilation in the same manner as discussed for the Octave benchmark, representing batch processing activity. Because the web browsing benchmark uses a JavaScript program to control the web page reloading, there is no actual user input when running this benchmark. However, users typically still expect good

responsiveness for such web page viewing activities, as scripted web page downloads are not uncommon in practice.

Figure 5 compares the response time of Linux versus RSIO using the web browsing benchmark under different system loads. We again varied the system load imposed by the kernel compilation from no load when no kernel compilation processes were run, to allowing 64 concurrent processes to run to perform the kernel compilation. When running without any background kernel compilation workload, Linux provides good response time, the worst case being only .96 s to download the web page. Usability studies have shown that web pages should take less than one second to download for the user to experience an uninterrupted web browsing experience [**?**]. Thus, the .96 s response time for an unloaded system is fast enough for an uninterrupted web browsing experience. However, as the load on the system increases, the response time of vanilla Linux increases dramatically. When 64 kernel compilation processes were running, the worst case response time for Linux was 12.85 s, resulting in a completely unacceptable delay of almost half a during the interactive phase of the benchmark. The 90th percentile response time was 1.91 s, which is still twice as slow as running the benchmark on an unloaded system.

In contrast, RSIO correctly identifies the web browser running the web browsing benchmark as latency-sensitive when it is receiving web data from the Internet since the network channel to the web proxy is configured as a RSIO I/O channel. RSIO correctly identifies the web browser activity as latency-sensitive even though there is no actual user input when running the benchmark. As a result, RSIO performed well under all different system loads. The worst web page reloading time for even a loaded system was only 0.84 s. This was even slightly better than Linux's web response time in the low load case. While some variation in performance is possible due to external factors from obtaining web data from a live web site across the Internet, the slight improvement can be also explained by the fact that even without the background load, the system still runs many other processes and any of those processes can affect the performance of the web browsing benchmark if they have similar priority as the web browser, as would be the case for the vanilla Linux system. Overall, only RSIO was able to consistently provide subsecond web page download times that were fast enough for an uninterrupted web browsing experience even when the system was loaded.

6.4 Media Player Benchmark

The fourth scenario represents a local user playing a movie while other batch jobs are running. For the local user to receive good playback quality, the media player application that the user is using to play the movie should be detected as performing latency-sensitive activities. Other batch jobs should be treated by the system as non-latency-sensitive activities.

For this scenario, we constructed a media player benchmark consisting of the MPlayer application playing a locally stored 5.36 MB MPEG-1 video clip with 834 352x240 frames. The video was scaled to 800x600 during playback, and the movie clip was played in a loop for five minutes. While the movie playback activity is occurring, another user is remotely connected to the same machine and running a kernel compilation in the same manner as discussed for the Octave benchmark, representing batch processing activity. Although media players often receive no user input while playing a movie, users clearly expect good system responsiveness to deliver all video frames and audio samples on time at the desired playback rate. Slowing down the playback rate would be undesirable and result in poor quality video and audio. For this benchmark, we used frame rate as the measure of performance. We logged the frame rate reported by the application, and used the worst case frame rate and the 90th percentile frame rate to quantify performance.

Figure 6 compares the video performance of Linux versus RSIO using the media player benchmark under different system loads. We again varied the system load imposed by the kernel compilation from no load when no kernel compilation processes were run, to allowing 64 concurrent processes to run to perform the kernel compilation. When running without any background kernel compilation workload, both Linux and RSIO are able to provide perfect video playback at 24 frames/second (fps). However, as the load on the system increases, the frame rate using vanilla Linux gets progressively worse. When 64 kernel compilation processes were running, the worst case frame rate for Linux was less than 7 fps, less than 30% of the desired frame rate. The 90th percentile frame rate was less than 10 fps, which also provided qualitatively poor video performance.

In contrast, RSIO correctly identifies the media player playing the movie clip as latency-sensitive because it is playing both audio and video and output to the audio device is configured as a RSIO I/O channel. As a result, RSIO performed well under all different system loads and was able to maintain the full 24 fps frame rate in all cases. In fact, only results for the worst case are shown for RSIO since if the worst case is already perfect, the 90th percentile results are the same. These results would not be achievable by simply using input focus since it is not uncommon for a user to watch a movie while input focus may be somewhere else. Furthermore, these results show that by identifying latency-sensitive activities, RSIO can also be used for applications with quality-

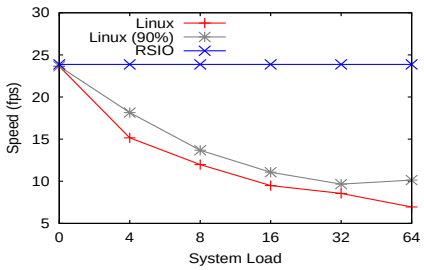

Figure 6: Media Player Benchmark

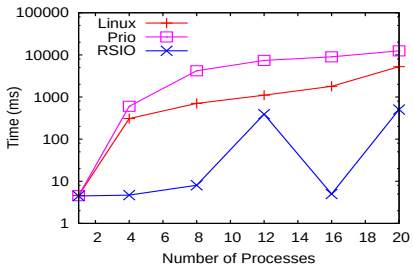

Figure 7: Multi-User Benchmark

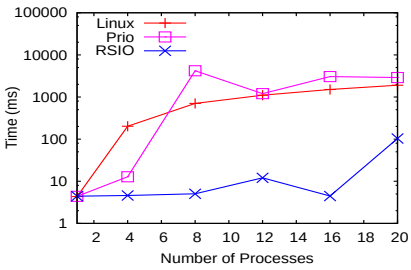

Figure 8: Multi-User Benchmark (90%)

of-service requirements to improve performance without requiring users to specify more complex scheduling parameters such as reservations or time constraints.

6.5 Multi-user Benchmark

The fifth scenario represents a system supporting multiple users, each of which is engaged in both latency-sensitive and non-latency-sensitive activities. In contrast to the previous scenarios, this scenario involves multiple interactive sessions competing for processor resources. The latency-sensitive activities correspond to users executing short and simple commands. The non-latency-sensitive activities correspond to users running batch jobs that require longer computations. For example, consider a group of students sharing the same server and using MATLAB to do their homeworks. Their MATLAB usage consists of two phases: a command typing phase and an execution phase. The first phase is interactive while the second phase is not. For the users to receive good system response time, the latency-sensitive activities should be detected when they occur, and periods of batch processing should be treated by the system as non-latency-sensitive activities.

Since it is difficult to get multiple users to do repeatable activities that can be measured to capture this scenario, we created a multi-user benchmark to emulate a set of students on a set of client machines that are remotely connected to a server and engaged in "typing then execute" behavior. The benchmark runs remotely on the client and creates a SSH connection to the server when started. Once started, it alternates between a "typing" phase and a "execution" phase in a loop. We emulate the "typing" phase by writing a short running command and a long running command to the SSH connection. After receiving the command, the shell process running on the server will execute the command and respond to the user after the command finishes. The benchmark automatically starts another round of measurement after receiving the response for both commands. Since the server operating system cannot distinguish between whether an SSH connection is generated by a real user or an application, we simply run multiple instances of the benchmark on a separate client machine to emulate the multi-user scenario. For example, we run five instances of the benchmark to emulate five users. To measure system response time, we report the elapsed time from sending the short command to the server until receiving the response for the command from the server. On an unloaded system, the short command takes only a few milliseconds to complete while the long command takes roughly ten minutes to complete.

Figures 7 and 8 compares the response time of Linux versus RSIO using the multi-user benchmark for different numbers of emulated users. For comparison, we also report the response time for Linux when boosting the priority of all users using `nice` -10, which we denote as PRIO. In an unloaded system, the short command takes 4.4 ms to complete for Linux, PRIO, and RSIO. However, as the number of emulated users increases, the response time of vanilla Linux increases dramatically. When 20 emulated users were running, the worst response time was over 5 s, more than three orders of magnitude worse than when running on an unloaded system. Similarly, the 90th percentile response time was over 2 s, which is still an unacceptable delay for interactive activities. The performance is poor because Linux cannot identify the latency-sensitive "typing" phase based just on its `sleep_avg` mechanism because it is short relative to the "execution" phase, resulting in any average-based measures being unable to identify such transitions between latency-sensitive and non-latency-sensitive activities by the same process.

Figures 7 and 8 show that the response time is even worse when all of the processes for all emulated users were boosted to higher priority. When 20 emulated users were running, the worst response time was over 10 s, twice as bad as the response time of Linux when users do not attempt to raise the priorities of any processes. This is because higher priority processes in Linux receive a large time quantum for execution, and processes at the same priority are run in round-robin order. As a result, all of the high priority processes have to wait their turn to run, and each turn takes a longer period of time, resulting

in longer delays and worse response times.

In contrast, RSIO correctly identifies the SSH sessions as latency-sensitive when and only when they are receiving input since the network channel for SSH is configured as a RSIO I/O channel. Although each SSH session emulates a user with both latency-sensitive and non-latency-sensitive activities, RSIO only boosts the priority of each process while it is performing the latency-sensitive activity. As a result, the worst case response time of RSIO is less than 500 ms, more than an order of magnitude less than the response time for either of the two configurations of vanilla Linux. Furthermore, the worst case response time for 8 emulated users or less was less than 8 ms. While the response time in all cases was significantly better than Linux, there was some noticeable variability in worst case response time across different numbers of users. This is largely due to multiple processes being considered as interactive by RSIO at the same time when 12 and 20 such emulated users were running concurrently. In this case, all of them are competing for processor resources and all of them receive degraded performance. Figure 8 shows that such contention does not occur in most cases. The 90th percentile response time was less than 100 ms in all cases, and less than 10 ms for 16 emulated users or less.

6.6 Overhead

To quantify the overhead of RSIO, we also compared the performance of RSIO versus vanilla Linux running LMbench [9], a popular tool for kernel overhead measurements. Figure 9 shows the results for various LMbench measurements, which exercise various forms of interprocess communication and system calls. The overhead added by RSIO in each measurement was less than .35 μs, representing less than 5% overhead in all cases. These results show that RSIO incurs modest overhead for this benchmark, which translates to negligible overhead for real applications in practice that do not focus just on exercising interprocess communication and system call usage.

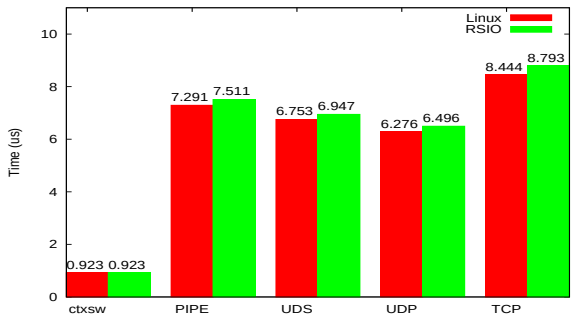

Figure 9: LMBench Benchmark

While LMbench captures the overhead imposed by RSIO for typical performance benchmarks, it is important to note that, like other performance benchmarks, LMbench is not designed as a latency-sensitive application. It does not involve user interaction during its execution. As a result, LMbench by design does not access any RSIO I/O channels and does not measure the overhead of that aspect of RSIO. However, RSIO I/O channels are by design accessed by applications that involve user interactions. Such user interactions typically operate at user time scales which are much slower than the time scales measured by typical kernel performance benchmarks, so the latter are not good indicators of RSIO I/O channel performance. As shown in the five application scenarios, the performance overhead of RSIO versus Linux was negligible as quantified by application performance on an otherwise unloaded system.

7 Conclusions and Future Work

RSIO introduces a new approach to processor scheduling for latency-sensitive activities that handle user interactions. RSIO monitors I/O channel usage instead of processor usage for detecting and prioritizing processes when they are handling latency-sensitive activities. It automatically tracks processes access I/O channels that handle user interactions, and detects communications among processes to determine processes involved in a user interaction. This is accomplished in part by using a confidence model with parameters based on human response time characteristics. RSIO's mechanism works with both local and remote I/O channels, and is compatible with existing processor schedulers. Our experimental results show that RSIO can provide substantial improvements in system responsiveness for a wide-range of applications, including console applications, applications that mix interactive and batch activities, common web browsing and multimedia applications, remote applications, and multi-user scenarios.

RSIO focuses on detecting what and when processes are latency-sensitive. When a system has many latency-sensitive activities running at the same time, scheduling them in a manner that is fair and provides good responsiveness is important. RSIO can serve as a basis for future work in developing better schedulers for supporting mixes of latency-sensitive activities.

References

[1] Size comparisons. In *http://en.wikipedia.org/wiki/Size_comparisons*, Wikipedia.

[2] Typing. In *http://en.wikipedia.org/wiki/Typing*, Wikipedia.

[3] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in

a general-purpose scheduler. In *Proceedings of the 17th ACM symposium on Operating systems principles* (1999), ACM Press, pp. 261–276.

[4] EATON, J. W. *GNU Octave Manual*. Network Theory Limited, 2002.

[5] ETSION, Y., TSAFRIR, D., AND FEITELSON, D. G. 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* (2004).

[6] ETSION, Y., TSAFRIR, D., AND FEITELSON, D. G. Process prioritization using output production: Scheduling for multimedia. *ACM Trans. Multimedia Comput. Commun. Appl. 2*, 4 (2006).

[7] LAMPSON, B. W., AND REDELL, D. D. Experience with Processes and Monitors in Mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[8] MCKUSICK, M. K., AND NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley.

[9] MCVOY, L. W., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference* (1996), pp. 279–294.

[10] MOLNOR, I., AND KOLIVAS, C. Interactivity in linux 2.6 scheduler. *http://www.kerneltrap.org/node/780*.

[11] NIEH, J., AND LAM, M. S. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)* (New York, Oct. 5–8 1997), vol. 31,5 of *Operating Systems Review*, ACM Press, pp. 184–197.

[12] NIELSEN, J. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, 2000.

[13] OUSTERHOUT, J. K. Scheduling Techniques for Concurrent Systems. *International Conference on Distributed Computing Systems* (1982), 22–30.

[14] PACKARD, K. Efficiently scheduling X clients.

[15] S. EVANS, K. CLARKE, D. S., AND SMAALDERS, B. Optimizing unix resource scheduling for user interaction. In *USENIX Summer Technical Conference* (June 1993), USENIX.

[16] SHNEIDERMAN, B. Response time and display rate in human performance with computers. *ACM Comput. Surv. 16*, 3 (1984).

[17] SOLOMON, D., AND RUSSINOVICH, M., Eds. *Inside Microsoft Windows 2000*. Microsoft Press Redmond, WA, USA, 2000.

[18] TORREY, L. A., COLEMAN, J., AND MILLER, B. P. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Softw. Pract. Exper. 37*, 4 (2007), 347–364.

[19] YAN, L., ZHONG, L., AND JHA, N. K. Towards a responsive, yet power-efficient, operating system: A holistic approach. *Mascots 00* (2005), 249–257.

[20] ZHENG, H., AND NIEH, J. SWAP: A Scheduler With Automatic Process Dependency Detection. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI-2004)* (2004).