

# RSIO: Automatic User Interaction Detection and Scheduling

Haoqiang Zheng  
VMware Inc  
3401 Hillview Ave  
Palo Alto, CA 94304  
hzheng@vmware.com

Jason Nieh  
Department of Computer Science  
Columbia University  
1214 Amsterdam Avenue MC0401  
New York, NY 10027  
nieh@cs.columbia.edu

## Abstract

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 automatically identifies processes involved in a user interaction and boosts their priorities at the time the interaction occurs to improve system response time. RSIO also detects processes indirectly involved in processing an interaction, automatically accounting for dependencies and boosting their priorities accordingly. RSIO works with existing schedulers 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.

**Categories and Subject Descriptors:** D.4.1 [Operating Systems]: Process Management-Scheduling

**General Terms:** Performance, Design, Experimentation

**Keywords:** Scheduling, Interactive applications, Dependencies

## 1. INTRODUCTION

Despite rapid advances in hardware technology, response time problems still plague modern computer systems. The rise of virtualization and cloud computing compound this problem by expanding the number of applications with different resource requirements and quality-of-service demands being run on the same system. 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. Since processor scheduling determines when a process can run, system designers have long recognized that good scheduling mechanisms are essential to provide quick response time for latency-sensitive

activities that handle user interactions. However, identifying processes involved in latency-sensitive activities is increasingly difficult for several reasons.

First, modern latency-sensitive applications often have very different execution behavior. Traditional desktop office productivity tools have different resource demands than multimedia applications. Multimedia applications have 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 [9, 3, 16].

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 [12, 3]. As a result, they are ineffective at identifying latency-sensitive applications across the broad range of interaction types common on modern computers.

Third, human-computer interactions on modern computers 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-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMETRICS'10, June 14–18, 2010, New York, New York, USA.  
Copyright 2010 ACM 978-1-4503-0038-4/10/06 ...\$10.00.

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 [9] of a process, will be ineffective and miss transitions between non-latency-sensitive and latency-sensitive activity.

To address these problems, we introduce RSIO (Response time Sensitive I/O), a new system for automatically identifying what activities are latency-sensitive and when they occur, enabling any processor scheduler to deliver better system response time. 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. Unlike traditional approaches that use process activity to guide scheduling, RSIO takes a radically different approach by using user I/O activity to guide processor scheduling of processes with latency-sensitive requirements. A simple RSIO configuration script can specify that operations on I/O channels involving user interactions be considered as 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. RSIO can also work with I/O channels that are used by both latency-sensitive and non-latency-sensitive activities by filtering out non-latency-sensitive interactions. When RSIO detects an operation on an I/O channel that is 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 Linux 2.6 schedulers and demonstrate that RSIO can provide substantial improvements in system response time for a wide range of applications with latency-sensitive activities.

## 2. RELATED WORK

Many approaches to processor scheduling have been con-

sidered 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.

The most common approach used in commodity operating systems is to schedule interactive applications based on their processor usage and sleeping behavior. For example, Windows [13] and FreeBSD [6] use a multi-level feedback queue scheduler in which processes that block waiting for I/O are given higher priorities and CPU-bound processes that use up their time quanta are given lower priorities. Similarly, the Linux  $O(1)$  scheduler [9] attempts to identify interactive processes as those that sleep longer and run less and gives them higher priority. Several studies [3, 16] have indicated that this approach does not work well. The fundamental problem is that processor usage behavior alone is often a poor indicator of interactivity given the resource intensive nature of many modern interactive applications.

Windows [13] provides some additional intelligence for processes waiting on I/O by adjusting how much its priority is raised depending on the device for which it was waiting. For example, a process waiting on keyboard I/O would get a larger boost than one waiting for disk I/O. RSIO differs from this form of using I/O information in several ways. First, RSIO does not tie its I/O mechanism to processor usage and sleeping behavior, avoiding the limitations of that approach. Second, RSIO is not limited to accounting for I/O usage only when processes block waiting on I/O. It accounts for any use of I/O, whether or not a process blocks, such as writing to an audio device, for example. Third, RSIO properly identifies processes that are indirectly involved in I/O interactions, not just those that are waiting on a device.

Another approach is to schedule using window system input focus. For example, schedulers used in Solaris [4] and Windows [13] raise the priority of processes associated with a window that has input focus. Input focus has also been used for scheduling other system resources [16]. Using input focus can work well, but 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 [4, 3].

HuC [3] schedules 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. For video applications, this results in all videos being displayed at the same frame rate regardless of window size. A fundamental 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. For example, a non-interactive kernel compilation generating lots of text will be mistakenly prioritized over other interactive applications.

Various approaches have focused on improving scheduling for latency-sensitive applications by adding additional pa-

rameters that can be specified. For example, Mac OS X [15] allows real-time processes to specify their periodic CPU resource requirements and timing constraints, and SMART [10] provides a bias on fair scheduling that uses the ability of batch processes to tolerate more latency to allow other latency-sensitive processes to run before them while preserving fair allocations. These approaches assume some way of knowing which activities are latency-sensitive and what processes are involved in those activities. RSIO is complementary to this work and focuses on automatically identifying latency-sensitive activities so that their scheduling parameters can be set correctly and dynamically. For example, RSIO can be used as an automatic mechanism for determining how to dynamically adjust the bias of a process in SMART.

Several approaches to real-time scheduling have recognized that applications may have different latency requirements during different periods of application execution. For example, SMART [10] 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. Priority inheritance [5] 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 [17] generalizes this work for dependencies due to other operating system resources and works for dynamic priorities.

While RSIO and SWAP both recognize that processes are dependent on each other, they differ in four fundamental ways. First, RSIO solves the problem of determining which processes are interactive and when they are interactive. SWAP assumes priorities are already known and has no idea how to identify interactive activities. Second, RSIO provides mechanisms to actively identify all processes involved in a user interaction to improve interactive performance. SWAP has no idea how to identify such processes, and only passively identifies a process that should run when a high priority process is blocked. Third, RSIO provides a new and novel scheduling policy for improving interactive performance based on access patterns to I/O devices. SWAP provides no scheduling policy and simply addresses the priority inversion problem with existing schedulers. Finally, RSIO introduces a new confidence-based mechanism based on time proximity. SWAP has a confidence-based mechanism, but it is simply based on whether a blocked high priority process is woken up by a given process.

### 3. RSIO USAGE MODEL

RSIO is based on the observation that activities are of

```
-----
# tty devices
rsio_config READ /dev/tty0
rsio_config READ /dev/tty1
rsio_config READ /dev/ttyN
# 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**

ten 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. Section 3.1 describes basic RSIO configuration. Section 3.2 discusses optional parameters for filtering.

### 3.1 RSIO Channel Configuration

RSIO provides a command, `rsio_config`, to configure I/O channels as being latency-sensitive. It takes two types of parameters: channel and operation. *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 is identified by the filename. For example, the I/O channel for a mouse device is `/dev/input/mice`. A socket is a dynamically created entity that 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 latency-sensitive. The operation can be read, write, or both read and write. 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.

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. While users may customize their own RSIO configurations, simple default settings such as those shown in Figure 1 work for the vast majority of programs so that users do not need to manage such configuration files unless they desire more customized control.

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 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/ttyN`. A successful read from a terminal device usually corresponds to user keyboard input. RSIO configures any read from a default TTY device as a latency-sensitive activity to improve system responsiveness to user keyboard input. 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. 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. 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. RSIO configures any write to the audio device as a latency-sensitive activity to ensure good audio quality. While a write operation occurs after a particular audio sample has been processed, audio processing is typically periodic and repetitive in nature, so all subsequent processing of audio samples will be handled in a latency-sensitive manner.

RSIO 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 can be 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. Monitoring keyboard or mouse input does not help since multimedia applications can generate long periods of display output without any user input. Our insight is that latency-sensitive applications that display output but do not require much user input usually also generate audio output. For example, movie players generate both video and sound, but non-latency-sensitive applications such as kernel compilation do not generate sound. As a result, this RSIO configuration automatically delineates between these two classes of applications by monitoring audio output instead of display output. While this approach works for most common cases, there are some scenarios in which RSIO may not correctly identify an activity as latency-sensitive. For example, playback of a silent movie will not be treated as latency-sensitive due to the absence of audio output. Conversely, a music player generating audio output will be treated as latency-sensitive even if no one is listening.

Network channels are used for handling 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 ma-

chine receiving data from a web server. 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. 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. RSIO configures any read on the network channel as latency-sensitive to improve system responsiveness when the user is remotely connected.

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 tracks input focus and treats processes receiving user input as being latency-sensitive only when such input is occurring. The configuration of the audio device enables audio and multimedia applications to be treated as latency-sensitive. The configuration of the network channels enables web and remote access applications to be treated as latency-sensitive during periods of user interaction. 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 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 in Section 7.

## 3.2 RSIO Access Filtering

Sometimes, non-latency-sensitive activities may also share access to the same I/O channels as latency-sensitive ones. For example, while port 22 is often used for latency-sensitive SSH connections, users can also use SCP commands to copy large files via this port. If port 22 is configured as a RSIO channel, then RSIO will also treat SCP commands as latency-sensitive even though they are not.

To address this issue, `rsio_config` can be used with additional optional parameters for each RSIO channel to filter out those non-latency-sensitive activities. Similar to spam filters that identify spam email based on various distinct characteristics, we observe that non-latency sensitive activities often have different I/O behavior from latency-sensitive activities. To illustrate how this works, we describe two types of filters that are supported in our RSIO prototype: inactive time and user identity.

The inactive time filter is designed based on the observation that users often need to pause to think between interactions. The filter allows users to specify the inactive time `I` and duration `D` for each RSIO channel. For example, if `I` and `D` are 1 second and 30 seconds, respectively, an activity must pause for at least one second every 30 seconds to be considered as latency-sensitive. By configuring a network

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

**Table 1: RSIO System State**

RSIO channel with this filter, SCP commands to copy large files can easily be filtered out since they will not have such inactive times while interactive SSH sessions will have such inactive times and be treated as latency-sensitive.

The user identity filter simply filters based on user ID. By default, a RSIO channel is latency-sensitive for all users. If a user ID or a group ID is specified for a RSIO channel, only accesses from this user ID or group ID are considered as latency-sensitive. The user identity filter provides a convenient mechanism for system administrators to implement differentiated service for different group of users.

Many other filters can be implemented as well. For example, filtering can be done based on source or destination IP addresses for networking based RSIO channels, or based on a white or black list of applications that should or should not be considered as latency-sensitive.

RSIO filtering provides a powerful and general mechanism that can be used for any I/O channel shared by latency-sensitive and non-latency-sensitive activities. At the same time, our experience and the results presented in Section 9 show that for many users, a basic RSIO configuration without filtering can effectively detect most latency-sensitive activities. Furthermore, in cases such as the shared display output used by many activities, it can be more effective to implicitly differentiate between latency-sensitive and non-latency-sensitive activities by simply identifying another I/O channel used by latency-sensitive activities that is not used by non-latency-sensitive activities, such as the audio device.

## 4. RSIO I/O CHANNEL INSTANTIATION

Table 1 summarizes the system state RSIO maintains to identify RSIO I/O channels and processes involved in user interactions using those channels. `rsio_config` causes the instantiation of a RSIO I/O channel, which is represented 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, described in more detail below. 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 created when the socket is actually created and used, so it usually does not exist at the time the `rsio_config` command is executed. 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 creation matches with a deferred RSIO command. For example, if a RSIO command for accepting connections to the SSH local port 22 was deferred, 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. 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.

## 5. 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 `write`, `writv` and `sendmsg`. RSIO instruments these system calls. When a system call 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, then 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 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 8 describes how latency-sensitive processes are scheduled. 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. 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 associated with a process that is a primary handler. As mentioned in Section 4, 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 it 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. However, this requires users to specify the I/O channels to 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 I/O channels to monitor for user output are specified, it is 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. The additional complexity involved in monitoring output may not provide much benefit given the uncertainty in determining when the output is complete. 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 this handler deactivation problem 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 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*, which is used to indicate how confident RSIO is that this 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 [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 6.

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. There is no easy way to determine precisely when a user stops interacting with the computer. 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*. If a handler is activated at time  $t$ , it will expire at time  $t + \text{sys\_expire}$ . *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 commands [14, 11].

## 6. 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 no 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 well in practice. If a primary handler depends on another process for handling a user interaction, 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.

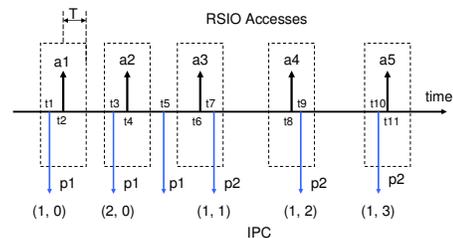
RSIO refers to a process that a primary handler depends upon as a *cohandler*. RSIO reuses the *handler* object discussed in Section 5 in a different way to maintain state associated with a cohandler. RSIO considers a process A as a possible cohandler for a process B if process A communicates with process 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; Section 7 discusses 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 cohancers. A RSIO confidence model decides if a potential cohandler is

an actual cohandler. When 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 cohancers. Suppose a handler for process A has been activated at time  $t_1$ . If process B communicates with process A during the time period  $t_1 - T$  to  $t_1 + T$ , the callback increments the confidence value of process B by one. For all other cohancers, 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  $co\_conf$ , 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  $t_1$  to  $t_{11}$ , process  $p$  was activated for 5 times, which happened at time  $t_2, t_4, t_6, t_8$  and  $t_{11}$ , respectively. Suppose process  $p$  has two potential cohancers  $p_1$  and  $p_2$  in its cohandler list with initial confidence values of zero. For interaction  $a_1$  at time  $t_2$ ,  $p_1$  has one interprocess communication with  $p$  at time  $t_1$  which is within  $T$  ms of  $t_2$ , so RSIO increases  $p_1$ 's confidence by 1. The confidence of  $p_2$  remains as 0. For interaction  $a_2$ , process  $p_1$  has another interprocess communication with  $p$  within the expected threshold while  $p_2$  has no such communications, so the confidence values of ( $p_1, p_2$ ) are adjusted to be (2, 0). For interaction  $a_3$ ,  $p_2$  has an interprocess communication within the expected threshold while  $p_1$  does not, so the confidence values of ( $p_1, p_2$ ) are adjusted to be (1, 1).  $p_2$  continues to communicate with  $p$  within the expected threshold for activation events  $a_4$  and  $a_5$  while  $p_1$  does not, so at the end of  $a_5$ , the confidence values of ( $p_1, p_2$ ) are adjusted to be (1, 3). At this point,  $p_2$  is treated as an activated cohandler while  $p_1$  is not. The example shows how RSIO automatically detects the cohandler transition from  $p_1$  to  $p_2$  based on its confidence model.



**Figure 2: Cohandler Detection**

A cohandler may communicate with other processes that should also be considered latency-sensitive. Each cohandler has an associated handler object and its own cohandler list. RSIO can thereby recursively identify potential cohancers of cohancers in the same way it identifies cohancers 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.

## 7. 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 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, the `writer` field is updated on a successful write system call. 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`.

When 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. If a match occurs, RSIO proceeds with the cohandler creation and update mechanism discussed in Section 6.

## 8. 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. A priority scheduler could use this information to boost the priority of processes that have been marked latency-sensitive. A fair-share scheduler could use this information to increase the shares of processes that have been marked latency-sensitive. 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 implementation of how RSIO can be used with a priority scheduler to improve system responsiveness, specifically the Linux O(1) processor scheduler. This priority-based scheduler dynamically adjusts the priorities of processes based on processor usage and sleeping behavior to improve the performance of interactive processes. A process's dynamic priority is decided by 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`, the sleep versus run ratio of the process. Processes with larger `sleep_avg` get more priority bonus and thus higher priority, so they should get better response time. However, predicting a process's interactivity based on its `sleep_avg` has various limitations [3, 16].

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 keep the same algorithm for computing a process's dynamic priority based on its nice value and a priority bonus. However, the priority bonus is determined by whether RSIO has indicated that the given process is latency-sensitive. By default, a process is 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 scheduler.

## 9. EXPERIMENTAL RESULTS

We have implemented a RSIO prototype in the Linux 2.6.19 kernel and modified the Linux scheduler to use RSIO as described in Section 8. To demonstrate its effectiveness, we ran several micro-benchmarks and real-world applications to compare the performance of RSIO versus the vanilla Linux O(1) scheduler. Since our RSIO prototype and the O(1) scheduler adjust priorities over the same dynamic range, this conservative comparison demonstrates how much RSIO can improve system responsiveness while maintaining the same limits on priority adjustments. Since the O(1) scheduler has been replaced by the CFS scheduler [8] in more recent versions of Linux, we also compare the performance of our RSIO prototype versus the CFS scheduler in the Linux 2.6.31.12 kernel.

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 configuration script previously shown in Figure 1.

For most of our workloads, we measure response time to quantify system performance. Users often care about when the system is responding poorly as opposed to just average response time. 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 and worst case response time performance.

Measurements were done using an HP xw9300 PC with a 2.6 GHz AMD Opteron processor and 2 GB RAM. For measurements involving a remote client, the client was a Dell Precision 370 with a 2.8 GHz Intel Pentium 4 processor and 2 GB RAM. All machines ran Ubuntu 6.06 and were connected via a dedicated 1 Gb Ethernet switch. All machines ran the Linux 2.6.19 kernel except when the HP PC used the CFS scheduler and required the 2.6.31.12 kernel.

## 9.1 Active Console Benchmark

The first scenario represents a local user using a window system with multiple consoles 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 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 as non-latency-sensitive activities.

For this scenario, we used 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.

Figure 3 compares the response time of RSIO versus O(1) and CFS using the active console benchmark under different system loads, from no load to running 64 concurrent kernel compilation processes. When running without any background workload, RSIO, O(1), and CFS provide the same response time of 10 ms for the interactive directory listing command. As the load on the system increases, the response time of O(1) and CFS increases dramatically. When 64 kernel compilation processes were running, the worst response time for O(1) and CFS was 2.5 s and 0.84 s, respectively. This is a significant and noticeable delay for interactive activities and makes typing and executing interactive commands very unpleasant. The 90th percentile response time for both cases was over 0.5 s. This is five times longer than the 100 ms response time threshold [11] 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. 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 well over an order of magnitude better than both O(1) and CFS. 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 threshold at which users can detect any response time delays.

## 9.2 Octave Benchmark

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

For this scenario, we used an octave benchmark with 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 [2], a MATLAB-like application that involves phases of interactive use and long-running computations. The user runs Octave and types 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 represent a user interacting with the application to set up a computation to run and should be considered as latency-sensitive. It consists of 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 represent the long running computation and should not be considered as latency-sensitive. It consists of an iterative loop that performs a set of long running computations on the two-dimensional array. Once the 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.

Figure 4 compares the response time of Linux schedulers versus RSIO using the Octave benchmark under different loads with different numbers of kernel compilation processes. The response time of O(1) is 133 ms when running without any background load, but increases dramatically as the load increases. When 64 kernel compilation processes were running, the worst response time for O(1) ballooned to 24.15 s, a completely unacceptable delay of almost half a minute

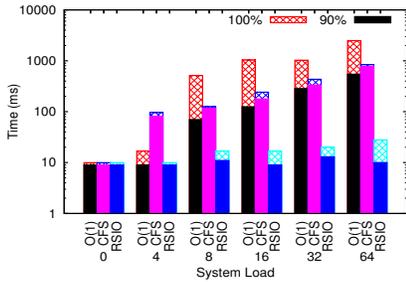


Figure 3: Active Console Benchmark

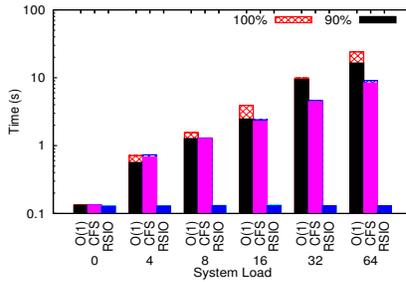


Figure 4: Octave Benchmark

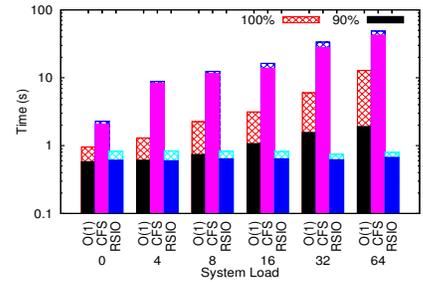


Figure 5: Web Browsing Benchmark

during the interactive phase of the benchmark. Even the 90th percentile response time is 16.47 s. The performance is horrible because the Octave benchmark does not sleep much, and thus the O(1) scheduler mistakenly always considers the benchmark as a non-interactive process since it only uses processor usage and sleep behavior to determine interactivity. The response time of CFS was similar to, but slightly better than, that of O(1). When no background workload was running, the worst response time was 112 ms. When 64 kernel compilation processes were running, the worst response time increased to 9.13 s, which is still unacceptable.

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. RSIO response time is 129 ms with no background load, and only 130 ms in the worst case even with 64 kernel compilation processes running. RSIO performs 185 times better than O(1) and 70 times better than CFS. The worst case response time of RSIO is essentially 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.

### 9.3 Web Browsing Benchmark

The third scenario represents a local user running a web browser to download 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 used 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 causes the other frame to reload “http://news.google.com” repeatedly for five minutes. This web page provides current news and is frequently updated. 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 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 a JavaScript program controls 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 schedulers versus RSIO using the web browsing benchmark under different system loads. The response time of the O(1) scheduler without background load is 0.96 s, but grows dramatically as the load on the system increases. When 64 kernel compilation processes were running, the worst case response time for Linux was 12.85 s, a completely unacceptable delay during the interactive phase of the benchmark. The 90th percentile response time was 1.91 s, twice as slow as running the benchmark on an unloaded system. The performance of CFS is much worse for this benchmark. The worst case response time was 2.29 s when running without background load. When 64 kernel compilation processes were running, the worst case response time increased to 48.9 s.

In contrast, RSIO correctly identifies the web browser 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 performed well under all different system loads. The worst web page reloading time for even a loaded system was only 0.84 s, slightly better than Linux’s web response time in the low load case. The slight improvement can be explained by the fact that even without the background load, the system still runs many other processes which 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. 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 [11]. 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.

### 9.4 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 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 used 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 is occurring, another user is remotely connected to the same machine and running a kernel compilation in the

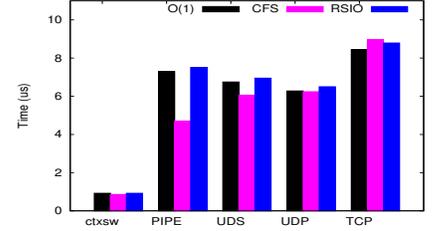
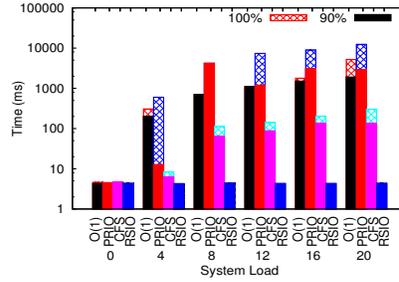
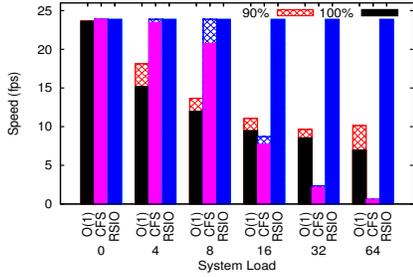


Figure 6: Media Player Benchmark

Figure 7: Multi-User Benchmark

Figure 8: LMBench Benchmark

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 schedulers versus RSIO using the media player benchmark under different loads with different numbers of kernel compilation processes. Without any background load, Linux schedulers and RSIO provide perfect video playback at 24 frames/second (fps). As the load on the system increases, the frame rate using Linux schedulers gets lower. When 64 kernel compilation processes were running, the worst case frame rate for O(1) was less than 7 fps, less than 30% of the desired frame rate. The 90th percentile frame rate was also poor at less than 10 fps. The CFS scheduler was doing fine when less than 8 kernel compilation processes were running. But as the number of background processes increased above 8, the performance of CFS decreases even more dramatically. When 64 kernel compilation processes were running, the worst case frame rate dropped to 0.6 fps.

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. RSIO performed well under all different system loads and was able to maintain the full 24 fps frame rate in all cases. Only worst case results are shown for RSIO since 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-of-service requirements to improve performance without requiring users to specify more complex scheduling parameters such as reservations or time constraints.

## 9.5 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. The latency-sensitive activities correspond to users executing short and simple commands. The non-latency-sensitive ones 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 while periods of batch processing should be treated as non-latency-sensitive.

Since it is difficult to get multiple users to do repeatable activities that can be measured to capture this scenario, we used 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 for three hours. 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 then respond to the user. The benchmark automatically starts another round of measurements 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 two to four minutes to complete.

Figure 7 compares the response time of Linux schedulers versus RSIO using the multi-user benchmark for different numbers of emulated users. For comparison, we also report the response time for Linux 2.6.19 when boosting the priority of all users using `nice -10`, which we denote as PRIO. In an unloaded system, the short command takes 4.35 ms to complete for all four approaches. As the number of users increases, the response time of O(1) increases dramatically. When 20 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. The 90th percentile response time was 1.9 s, still an unacceptable delay for interactive activities. The performance is poor because the O(1) scheduler 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.

The response time is even worse for PRIO when all of

the processes for all users were boosted to higher priority. When 20 users were running, the worst response time was over 10 s, twice as bad as the response time of O(1) 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. 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.

CFS performs better than O(1) and PRIO. It acts like a round-robin scheduler when all the processes are CPU bound and run at the same priority. CFS performs better than O(1) and PRIO because it used a much smaller time quantum than O(1) and PRIO. However, its worst response time is still poor, over 300 ms when 20 users were running.

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 response time of RSIO is below 4.4 ms across all system loads.

## 9.6 Overhead

We also compared the performance of RSIO versus Linux schedulers running LMBench [7], a popular tool for kernel overhead measurements. The performance difference between RSIO and O(1) shows the overhead of RSIO since both are implemented based on Linux 2.6.19. Figure 8 shows results for various LMBench measurements which exercise interprocess communication and system calls. The overhead added by RSIO was less than  $0.35 \mu\text{s}$  in all cases, representing less than 5% overhead. 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 8 also shows the overhead of CFS is close to that of O(1), except for the PIPE latency measurement in which the overhead is 36% smaller. The performance difference is likely due to CFS using the 2.6.31.12 kernel version instead of the 2.6.19 kernel version used for O(1) and RSIO.

While LMBench captures the overhead imposed by RSIO for typical performance benchmarks, it is not designed as a latency-sensitive application and 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.

## 10. CONCLUSIONS

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. 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.

## 11. ACKNOWLEDGMENTS

Michael Swift provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-0717544, CNS-0914845, and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

## 12. REFERENCES

- [1] Typing. In <http://en.wikipedia.org/wiki/Typing>.
- [2] J. W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.
- [3] Y. Etsion, D. Tsafir, and D. G. Feitelson. Process Prioritization Using Output Production: Scheduling for Multimedia. *ACM Transactions on Multimedia Computing, Communications and Applications*, 2(4), Nov. 2006.
- [4] S. Evans, K. Clarke, D. Singleton, and B. Smaalders. Optimizing UNIX Resource Scheduling for User Interaction. In *Proceedings of the Summer 1993 USENIX Conference*, June 1993.
- [5] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2), Feb. 1980.
- [6] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.
- [7] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, Jan 1996.
- [8] I. Molnar. Linux CFS Scheduler. <http://kerneltrap.org/node/11737>.
- [9] I. Molnar and C. Kolivas. Interactivity in Linux 2.6 Scheduler. <http://www.kerneltrap.org/node/780>.
- [10] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Oct. 1997.
- [11] J. Nielsen. *Designing Web Usability*. New Riders Publishing, 2000.
- [12] K. Packard. Efficiently Scheduling X Clients. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, June 2000.
- [13] M. Russinovich and D. Solomon. *Windows Internals (5th Edition)*. Microsoft Press, 2009.
- [14] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3), Sept. 1984.
- [15] A. Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006.
- [16] L. Yan, L. Zhong, and N. K. Jha. Towards a Responsive, Yet Power-efficient, Operating System: A Holistic Approach. In *Proceedings of the 13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2005.
- [17] H. Zheng and J. Nieh. SWAP: A Scheduler with Automatic Process Dependency Detection. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.