Redline: First Class Support for Interactivity in Commodity Operating Systems

Ting Yang tingy@cs.umass.edu
Tongping Liu tonyliu@cs.umass.edu
Emery D. Berger emery@cs.umass.edu
Scott F. Kaplan[†] sfkaplan@cs.amherst.edu
J. Eliot B. Moss moss@cs.umass.edu

Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003-9264

[†]*Dept. of Mathematics and Computer Science*
Amherst College
Amherst, MA 01002-5000

Abstract

While modern workloads are increasingly interactive and resource-intensive (e.g., graphical user interfaces, browsers, and multimedia players), current operating systems have not kept up. These operating systems, which evolved from core designs that date to the 1970s and 1980s, provide good support for batch and command-line applications, but their *ad hoc* attempts to handle interactive workloads are poor. Their best-effort, priority-based schedulers provide no bounds on delays, and their resource managers (e.g., memory managers and disk I/O schedulers) are mostly oblivious to response time requirements. Pressure on any one of these resources can significantly degrade application responsiveness.

We present Redline, a system that brings first-class support for interactive applications to commodity operating systems. Redline works with unaltered applications and standard APIs. It uses lightweight specifications to orchestrate memory and disk I/O management so that they serve the needs of interactive applications. Unlike real-time systems that treat specifications as strict requirements and thus pessimistically limit system utilization, Redline dynamically adapts to recent load, maximizing responsiveness and system utilization. We show that Redline delivers responsiveness to interactive applications even in the face of extreme workloads including fork bombs, memory bombs and bursty, large disk I/O requests, reducing application pauses by up to two orders of magnitude.

1 Introduction

The enormous increases in processing power, memory size, storage capacity, and network bandwidth over the past two decades have led to a dramatic change in the richness of desktop environments. Users routinely run highly-graphical user interfaces with resource-intensive applications ranging from video players and photo editors to web browsers, complete with embedded Javascript and Flash applets.

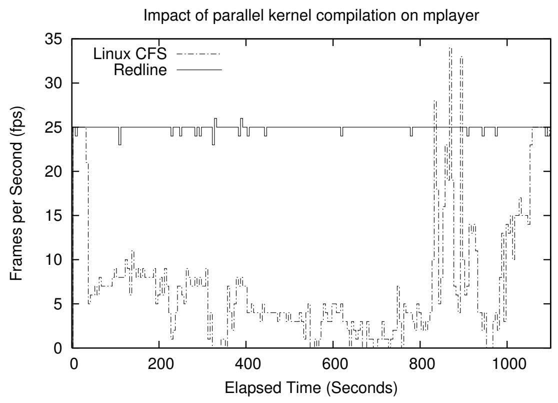

Figure 1: The frame rate of mplayer when performing a Linux kernel compiling using make -j32 on a standard Linux 2.6 kernel and on the Redline kernel.

Unfortunately, existing general-purpose operating systems do not provide adequate support for these modern applications. Current operating systems like Windows, Linux, and Solaris were designed using standard resource management policies and algorithms, which in turn were not developed with interactivity in mind. While their CPU schedulers attempt to enhance interactive behavior, their memory managers and I/O managers focus on increasing system throughput rather than reducing latency. This lack of coordination between subsystems, and the fact that they can work at cross purposes, means that pressure on any resource can significantly degrade application responsiveness.

For example, a memory-intensive application can cause the system to evict pages from the graphical user interface, making the system as a whole unresponsive. Similarly, disk-intensive applications can easily saturate I/O bandwidth, making applications like video players unusable. Furthermore, while best-effort, priority-based schedulers are a good match for batch-style applications, they provide limited support for ensuring responsiveness. Activities that strain the resources of a system—image or video processing, working with large data files or file systems, or switching frequently between a number of active applications—are likely to cause one of the resource managers to under-allocate resources to some interactive tasks, making those tasks respond poorly.

As an example, Figure 1 shows the frame rate of mplayer, a movie player, while a Linux kernel compilation, invoked using make -j32, is performed using both a standard Linux 2.6.x kernel and our Redline kernel. For the standard kernel, the compilation substantially degrades the interactivity of the movie player, rendering the video unwatchable. Worse, the entire GUI becomes unresponsive. Similar behavior occurs on Windows when watching a video using the Windows Media Player while running a virus scan in the background.

Contributions

We present Redline, a system that integrates resource management (memory management and disk I/O scheduling) with the CPU scheduler, orchestrating these resource managers to maximize the responsiveness of interactive applications.

Redline relies on *lightweight specifications* to provide enough information about the response time requirements of interactive applications. These specifications, which extend Rialto's *CPU specifications* [15], give an estimate of the resources required by an application over any period in which they are active. These specifications are concise, consisting of just a few parameters (Section 3), and are straightforward to generate: a graduate student was able to write specifications for a suite of about 100 applications—including Linux's latency-sensitive daemons—in just one day. In an actual deployment, we expect these specifications to be provided by application developers.

Each resource manager uses the specifications to inform its decisions. Redline extends a standard time-sharing CPU scheduler with an *earliest deadline first (EDF)*-based scheduler [18] that uses these specifications to schedule interactive applications (Section 6). Redline's memory manager protects the working sets of interactive applications, preferentially evicting pages from non-interactive applications and further reducing the risk of paging by maintaining a pool of free pages available only to interactive tasks—a *rate-controlled memory reserve* (Section 4). Redline's disk I/O manager avoids pauses in interactive applications by dynamically prioritizing these tasks according to their need to complete the I/O operation and resume execution on the CPU (Section 5).

Unlike real time systems that sacrifice system utilization in exchange for strict guarantees [15, 16], Redline provides strong isolation for interactive applications while ensuring high system utilization. Furthermore, Redline works with standard APIs and does not require any alterations to existing applications. This combination makes Redline practical for use in commodity operating system environments.

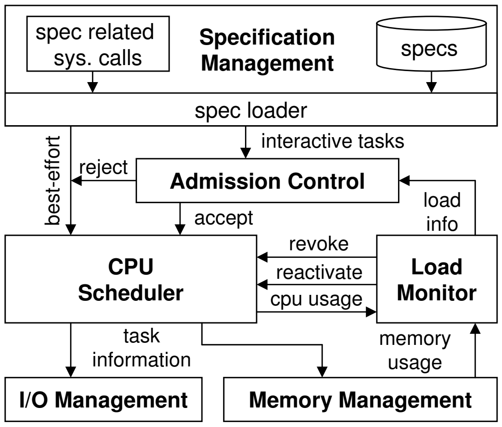

Figure 2: The Redline system.

We have implemented Redline as an extension to the Linux kernel (Section 7). We present the results of an extensive empirical evaluation comparing Redline with the standard Linux kernel (Section 8). These results demonstrate Redline's effectiveness in ensuring the responsiveness of interactive applications even in the face of extreme workloads, including bursty I/O-heavy background processes, fork bombs, and memory bombs.

2 Redline Overview

This paper focuses on Redline's support for interactive and best-effort tasks:

1. **Interactive (Iact)**: response-time sensitive tasks that provide services in response to external requests or events. These include not only tasks that interact with users, but also tasks that serve requests from other tasks, such as kernel daemons;

2. **Best-effort (BE)**: tasks whose performance is not critical, such as virus scanners.

Figure 2 presents an overview of the Redline system. At the top, the specification management allows a system administrator to provide specifications for a set of important applications. Redline loads each specification from a file whenever the corresponding application is launched. Redline treats any application without a specification as a best-effort task.

Whenever a new task is launched, Redline performs admission control to determine whether the system can accommodate it. Specifically, Redline dynamically tracks the load consumed by the active interactive tasks, and then uses the new task's specification to determine whether the resources are available to support it as an interactive task.

Once the admission control mechanism accepts a task, Redline propagates its specification to the memory manager, the disk I/O manager, and the CPU scheduler. The memory manager uses the specification to protect the task's working set, preferentially evicting either non-working-set pages or pages from best-effort tasks. It also maintains a specially controlled pool of free pages for interactive tasks, thus isolating them from best-effort tasks that allocate aggressively. The disk I/O management assigns higher priorities to interactive tasks, ensuring that I/O requests from interactive tasks finish as quickly as possible. Finally, Redline's extended CPU scheduler provides the required CPU resources for the interactive tasks.

Unlike real-time systems, which pessimistically reject jobs if the combined specifications would exceed system capacity, Redline optimistically accepts new jobs based on the actual usage of the system, and adapts to overload if necessary. When Redline detects an overload—that the interactive tasks are trying to consume more resources than are available—then the load monitor selects a victim to downgrade, making it a best-effort task. This victim is the task that acts least like an interactive task (i.e., it is the most CPU-intensive). This strategy allows other interactive tasks to continue to meet their response-time requirements. Whenever more resources become available, Redline will promote the downgraded task, again making it interactive.

By integrating resource management with appropriate admission and load control, Redline effectively maintains interactive responsiveness even under heavy resource contention.

3 Redline Specifications

Under Redline, each interactive application (e.g., a text editor, a movie player, or a web browser) must have a specification to ensure its responsiveness. However, the application itself depends on other tasks that must also be responsive. Specifically, the graphical user interface (e.g., the X Windows server, the window/desktop manager) comprise a set of tasks that must be given specifications to ensure that the user interface remains responsive. Similarly, specifications for a range of kernel threads and daemons allow Redline to ensure that the system remains stable. Finally, specifications for a range of administrative tools (e.g., bash, top, ls, and kill) make it possible to manage the system even under extreme load.

A specification in Redline is an extension of *CPU reservations* [15] that allow an interactive task to reserve *C* milliseconds of computation time out of every *T* milliseconds. A specification consists of the following fields:

$$\langle \textit{pathname}:\textit{type}:C:T:\textit{flags}:\pi:\textit{io}\rangle$$

The first field is the pathname to the executable, and the second field is its type (usually Iact). Two important flags include I, which determines whether the specification can be inherited by a child process, and R, which indicates if

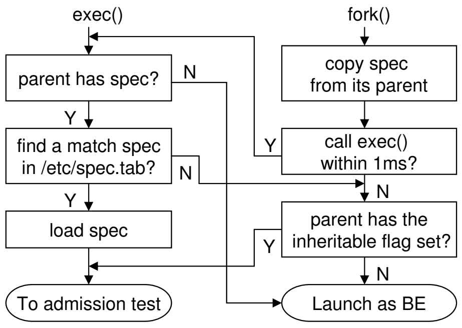

Figure 3: Loading specifications in Redline

the specification may be revoked when the system is overloaded. A specification can also contain two optional fields: π is the memory protection period in seconds (see Section 4) and *io* is the I/O priority (see Section 5).

For example, here is the specification for *mplayer*, an interactive movie player:

⟨/usr/bin/mplayer:Iact:5:30:IR:-:-⟩

This specification indicates that *mplayer* is an interactive task that reserves 5 ms out of each 30 ms period, whose specification is inheritable, that its interactive status can be revoked if necessary, and whose memory protection period and I/O priority are chosen by Redline automatically.

Setting specifications: We derived specifications for a range of applications by following several simple rules. Because most administrative tools are short-lived, reserving a small percentage of the CPU bandwidth over hundreds of milliseconds is sufficient to ensure responsiveness. While most kernel threads and daemons are not CPU intensive, they are response-time sensitive, so their reservation period should be in the tens of milliseconds. Finally, for interactive applications like a movie player, the reservation period should be around 30 ms to ensure 30 frames per second, which implies that the X server and window/desktop manager should also use the same reservation period.

We found that setting specifications was straightforward. It took one graduate student a single work day to manually generate a set of specifications for about 100 applications in a Linux system using the K Desktop Environment (KDE). Because this specification file is portable, specifications for a wide variety of applications could easily be shipped with an operating system distribution.

Loading specifications: Redline stores its specifications in a file (/etc/spec/spec.tab). An interactive task either uses the specification loaded from this file when exec() is invoked, or it adopts the one inherited from its parent. Figure 3 shows how Redline loads the specification for each task.

In practice, most tasks that invoke exec() do so shortly after being forked. Therefore, when a new task is forked, Redline gives it a 1 ms execution window to perform an exec(). If it does so, and if the parent task is itself an interactive task, then Redline searches the specification file for an entry with a matching path name. If a match is found, the specification provided is adopted for this new task.

If there is no entry for the application, or if the task does not invoke exec() during that initial 1 ms window, then Redline examines whether the task should inherit its parent task's specification. If that specification is marked as *inheritable*, then it is applied to the new task. Under all other circumstances, the new task is classified as best-effort.

If the new task's specification classify it as interactive, then Redline will submit the task to admission control. If Redline determines that the load is low enough to support the resource needs of this new interactive task, then it is admitted; otherwise, it is demoted to be a best-effort task.

Note that in the absence of any specifications, all tasks become best-effort tasks, and Redline acts like a standard system without special support for interactivity.

4 Redline Virtual Memory Management

The goal of existing virtual memory managers (VMM) in commodity operating systems is to maximize overall system throughput. Most VMM's employ "use-it-or-lose-it" policies under which memory referencing speed determines allocation: the more pages a task references per second, the larger its main memory allocation.

A task that blocks on I/O is more vulnerable to losing its allocation and then later being forced to page-swap when it awakens. Because typical VMM's do not isolate each task's allocation, a single memory-intensive task can "steal" the allocations of other tasks that are not actively using their pages, causing the other tasks to page-swap more often. Worse, page swapping itself is the kind of blocking I/O operation that can cause a task to lose more of its allocation, exacerbating the problem.

Because interactive tasks routinely block on user input, they are especially susceptible to allocation loss. If an interactive task has an insufficient allocation, its execution is likely to be interrupted by lengthy page-swap operations, leaving it unresponsive, and making it even more susceptible to further allocation loss.

To address this problem, the Redline VMM is designed to keep interactive tasks responsive even after blocking on input. Redline uses three mechanisms to achieve these goals: *protecting the working sets* of interactive tasks, using a *rate-controlled memory reserve*, and setting *speed-bump* pages.

Protecting working sets: The Linux VMM uses a page replacement algorithm that approximates a *global least recently used (gLRU)* policy. Pages are approximately ordered by recency of use, without consideration of the task to which each page belongs. The VMM cleans and selects for reclamation the least-recently used pages. The implicit goal of this policy is to minimize the total number of page swaps performed by the system, irrespective of how any one process performs.

For an interactive task to remain responsive, a VMM must keep its *working set*—those pages that are currently in active use—resident in memory. Under gLRU, a portion of a task's working set may be evicted from main memory if the system-wide demand for main memory is large enough, and if a task does not reference its pages rapidly enough. Under this memory pressure, interactive tasks can quickly become non-responsive.

Under the Redline VMM, each interactive task can specify a *memory protection period* π. The VMM will evict a page only if it has not been referenced for at least π seconds—that is, if the page has *expired*. By default, $\pi = 30 \times 60$, or 30 minutes if π is not supplied by the specification. This default period requires that a user ignore an interactive application for a substantial period of time before the VMM can evict its working set.

The Redline VMM handles the pages of interactive tasks and best-effort tasks differently. If page reclamation is caused by a best-effort task, then the VMM reclaims pages using the system's default VMM mechanism, but with a slight modification: only pages belonging to best-effort tasks and expired pages belonging to interactive tasks may be reclaimed. Among these two types of pages eligible for reclamation, the VMM selects the least recently used pages.

However, if page reclamation is caused by an interactive task, then the VMM first tries to use the *rate-controlled memory reserve* (described in more detail below). If this reserve provides insufficient space, reclamation proceeds as above for best-effort tasks, reclaiming not recently used pages from best-effort tasks and expired pages from interactive tasks. If this attempt at reclamation is also insufficient, the VMM then reclaims more recently used pages from best-effort tasks. If even this aggressive approach is insufficient, then there is not enough memory to cache the working sets of all of the interactive tasks. In this case, the VMM demotes some interactive tasks to best-effort tasks. After taking this step, the VMM repeats the above steps, attempting to reclaim sufficient main memory space.

Rate-controlled reserve: The Linux VMM forces the allocating task to reclaim pages when the amount of free memory falls below a threshold. This approach is not desirable for interactive tasks because, should such a task fault, it would block during the potentially lengthy reclamation process. Instead, the Redline VMM maintains a small free-memory reserve (about 8 MB in our implementation) for interactive tasks, and controls the rate at which this reserve is consumed. Although an interactive task still may block during reclamation, the reserve makes that situation significantly less likely.

The Redline VMM gives each interactive task a reserve budget b (the default is 256 pages) and records the time t_f

when the first page in its budget is consumed. For each page consumed from the reserve, the Redline VMM reduces the budget of the consuming task and then triggers a kernel thread to reclaim pages in background if necessary. We do not want any one task to quickly exhaust the reserve and affect other tasks, so the rate at which a task consumes reserved pages should not be faster than the system can reclaim them. Therefore, the Redline VMM charges each reserved page a cost c that is roughly the overhead of one disk access operation (5 ms in Redline). When a task expends its budget, the VMM evaluates the inequality $t_f + bc < t$, where t is the current time. If the inequality is true, then the VMM adds b to the task's budget. If the inequality is false, the task is consuming reserved pages too quickly. Thus, the VMM prevents the task from using the reserve until b pages are reclaimed or the reserve resumes its full capacity. We show the effect of this limited isolation between interactive tasks Section 8.

Setting speed-bump pages: A VMM can only reclaim a *clean* (unmodified) page; *dirty* (modified) pages must be copied to the backing store, thus cleaning them, before they can be reclaimed. Sometimes, a best-effort task may dirty pages faster than the VMM can clean them, thus preventing the VMM from reclaiming those pages. If enough such pages are unreclaimable, the VMM may be unable to cache the working sets of interactive tasks, thus allowing one best-effort task to degrade interactivity.

To prevent best-effort tasks from "locking" memory in this manner, whenever the Redline VMM is aggressively searching for pages to reclaim (see above) and finds a page belonging to a best-effort task, it removes access permissions to that page and marks it as a *speed-bump page*. If the task then references that page, execution traps into the kernel and the VMM notices that the referenced page is a speed-bump. Before restoring access permissions and resuming execution, Redline suspends the task briefly (e.g., 100 ms). The VMM therefore slows the task's memory reference rate, giving the VMM enough time to reclaim more of its pages.

5 Redline Disk I/O Management

Like the VMM, the I/O manager of a general-purpose OS does not distinguish between interactive and best-effort tasks. The policies that determine when and in what order pages are read from and written to disk are designed to optimize system throughput and are oblivious to CPU scheduler goals. This obliviousness can lead the I/O manager to schedule the requests for best-effort tasks before those of interactive tasks in a way that substantially harms response times.

To prevent this problem, the Redline I/O manager manages key I/O events in both the file system and block device layer so that responsive tasks can meet their deadlines.

Journaling: For Linux, *ext3* is a journaling file system that is the default for most distributions. Like any journaling file system, *ext3* commits its updates as atomic transactions, each of which writes a group of cached, dirty pages along with their new metadata. Its implementation is designed to maximize system-wide throughput, sometimes to the detriment of CPU scheduling goals. We describe here a particular problem with this file system's implementation that Redline fixes. Although this particular problem is specific to Linux's *ext3*, it is representative of the way in which any OS component that manages system resources can undermine interactivity.

Consider two tasks: an interactive task P_i, and a best-effort task P_{be}, which simultaneously use the write() system call to save data to some file on the same *ext3* file system. These system calls will not immediately initiate disk activity. Instead, the data written via this mechanism will be buffered as a set of dirty pages in the file system cache. Critically, these pages will also be added to a single, global, *compound* transaction by *ext3*. This transaction will thus contain dirty pages from any file written by any task, including pages written by both P_i and P_{be}.

Consider the case that P_{be} writes a large amount of data through write(), while P_i writes a small amount. Furthermore, after both tasks have performed these write() operations, suppose that P_i performs an fsync() system call to ensure that its updates are committed to disk. Because of the compound transactions used by *ext3*, P_i will block until both its own dirty pages and those of P_{be} are written to disk.

If the OS caches too many pages written by P_{be}, then the fsync() operation will force P_i to become noticeably unresponsive. This poor interaction between compound transactions and fsync() occurs not only for *ext3*, but also for *ReiserFS*[25]. Under Linux, the *dirty threshold* d is a system-wide parameter that determines what percentage of main memory may hold dirty pages—pages that may belong to any task—before a disk transfer is initiated to "clean" those pages. By default, $d = 10\%$, making it possible on a typical system for 100 MB to 200 MB of dirty pages to be cached and then written synchronously when fsync() is called.

Redline takes a number of steps to limit this effect. First, Redline assigns different dirty thresholds for each type of task (RT:10%, Iact:5%, BE:2MB). Redline further restricts the threshold for best-effort tasks to a constant limit of 2 MB, ensuring that no matter the size of main memory, best-effort tasks cannot fill the compound transactions of some journaling file system with a large number of dirty pages. Finally, Redline assigns the kernel task that manages write operations for each file system (in Linux, kjournald) to be an interactive task, ensuring that time-critical transaction operations are not delayed by other demands on the system.

Block device layer: Much like the journaling file systems described above, a block device layer may have unified data structures, thresholds, or policies that are applied irrespective of the tasks involved. These components are typically designed to maximize system-wide throughput. However, the unification performed by these components may harm the responsiveness of interactive tasks. Redline addresses these problems by handling these components of the block device layer on a per-task-type basis.

The Linux block device manager, dubbed *Completely Fair Queuing (CFQ)*, contains a single *request queue* that stores I/O requests from which actual disk operations are drawn. Although this request queue internally organizes I/O requests into classes (real-time, best-effort, and "idle"), it has a maximum capacity that is oblivious to these classes. When a new request is submitted, the *congestion control mechanism* examines only whether the request queue is full, with no consideration of the requesting task's class. If the queue is full, the submitting task blocks and is placed in a FIFO-ordered wait-queue. Thus, a best-effort task might rapidly submit a large number of requests, thus congesting the block device and arbitrarily delaying some interactive task that requests an I/O operation.

Redline addresses this problem by using multiple request queues, one per task class. If one of the queues fills, the congestion control mechanism will only block processes in the class associated with that queue. Therefore, no matter how many requests have been generated by best-effort tasks, those requests alone cannot cause an interactive task to block.

In addition, while the default request queue for CFQ is capable of differentiating between various request types, it does not provide sufficient isolation for interactive tasks. Specifically, once a request has been accepted into the request queue, it awaits selection by the I/O scheduler to be placed in the dispatch queue, where it is scheduled by a typical elevator algorithm to be performed by the disk itself. This default CFQ scheduler not only gives preference to requests based on their class, but also respects the priorities that tasks assign requests within each class. However, each buffered write request—the most common kind—is placed into a set of shared queues in best-effort class irrespective of the task that submitted the request. Therefore, best-effort tasks may still interfere with the buffered write requests of interactive tasks by submitting large numbers of buffered write requests.

Redline adds both an interactive Iact class to the request queue management, matching its CPU scheduling classes. All write requests are placed into the appropriate request queue class based on the type of the submitting task. The I/O scheduler prefers requests from the interactive class over those in the BE class, thus ensuring isolation of the requests of interactive tasks from BE tasks.

Additionally, the specification for a task (see Section 3) includes the ability to specify the priority of the task's I/O requests. If the specification does not explicitly provide this information, then Redline automatically assigns a higher priority to tasks with smaller T values. Redline organizes requests on per-task basis within the given class, allowing the I/O scheduler provide some isolation between interactive tasks.

Finally, CFQ, by default, does not guard against starvation. A task that submits a low-priority I/O request into one of the lesser classes may never have that request serviced. Redline modifies the I/O scheduler to ensure that all requests are eventually served, preventing this starvation.

6 Redline CPU Scheduling

The time-sharing schedulers used by commodity operating systems to manage both interactive and best-effort tasks neither protect against overload nor provide consistent interactive performance. To address these limitations, Redline employs *admission control* to protect it against overload in most cases, and uses *load control* to recover quickly from sudden bursts of activity. Redline also uses an *Earliest Deadline First (EDF)*-based scheduler for interactive tasks to ensure that they receive CPU time as required by their specifications.

6.1 Admission and Load Control

Admission control determines whether a system has sufficient free resources to support a new task. It is required by any system that provides response time guarantees, such as real-time systems, though it is typically absent from commodity operating systems.

In real-time systems, admission control works by conservatively assuming that each task will always consume all of resources indicated in its specification. If the addition of a new task would cause the sum of the specified CPU bandwidths for all active tasks to exceed the available CPU bandwidth, then the new task will be rejected. This conservative approach overestimates the CPU bandwidth actually consumed by aperiodic tasks, which often use much less CPU time than their specifications would indicate. Thus, real-time admission control often rejects tasks that could actually be successfully supported, pessimistically reducing system utilization.

To increase system utilization, Redline uses a more permissive admission control policy. If the CPU bandwidth actually consumed by interactive tasks is not too high, then new interactive tasks may be admitted. Because this approach could lead to an overloaded system if interactive tasks begin to consume more CPU bandwidth, Redline employs load control that allows it to quickly recover from such overload. Consequently, Redline does not provide the inviolable guarantees of a real-time system, but in exchange for allowing short-lived system overloads, Redline ensures far higher utilization than real-time systems could provide.

To maximize utilization while controlling load, Redline strives to keep the CPU bandwidth consumed by interactive tasks within a fixed range. It tracks the actual CPU load that drives its admission and load control policies. We describe here how Redline tracks load and how it manages interactive tasks.

Load tracking: Redline maintains two values that reflect CPU load. The first, R_{load}, represents the actual, recent CPU use. Once per second, Redline measures the CPU bandwidth consumed during that interval by interactive tasks. R_{load} is the arithmetic mean of the four most recent CPU bandwidth samples. This four-second *observation window* is long enough to smooth short-lived bursts of CPU use, and is short enough that longer-lived overloads are quickly detected.

The second value, S_{load}, projects expected CPU load. When an interactive task i is launched, the task's specified CPU bandwidth, $B_i = \frac{C_i}{T_i}$, is added to S_{load}. Since specifications are conservative, S_{load} may overestimate the load. Therefore, over time, S_{load} is exponentially decayed toward R_{load}. Additionally, as interactive tasks terminate, their contribution to S_{load} is subtracted from that value, thus updating the projection.

Management policies: Redline uses these CPU load values to control interactive tasks through a set of three policies:

Admission: When a new interactive task i is submitted, admission control must determine whether accepting the task will force the CPU load above a threshold R_{hi}, thus placing too high a load on the system. If $max(R_{load}, S_{load}) + B_i < R_{hi}$, then i is admitted as an interactive task; otherwise it is placed into the best-effort class. Thus, Redline avoids overloading the system, but does so based on measured system load, thus allowing higher utilization than real-time systems.

Revocation: Because of Redline's permissive admission control, it is possible for some interactive tasks to increase their CPU consumption and overload the system. Under these circumstances, Redline revokes the interactive classification of some tasks, demoting them to the best-effort class, thus allowing the remaining interactive tasks to remain responsive.

Specifically, if $R_{load} > R_{max}$, where $R_{max} > R_{hi}$, then Redline revokes tasks until R_{load} falls below R_{hi}. Redline prefers to revoke a task that exhausted its reservation during the observation window, indicating that the task may be more CPU-bound and less interactive. However, if there are no such tasks, Redline revokes the task with the highest CPU bandwidth consumption. Certain tasks are set to be invulnerable to revocation to preserve overall system responsiveness, such as kernel threads and the graphical user interface tasks.

Reactivation: If $R_{load} < R_{lo}$, Redline will reactivate previously revoked tasks, promoting them from the best-effort class to the interactive class. A task is eligible for reactivation if *both* (a) it passes the usual admission test, *and* (b) its virtual memory size minus its resident size is less than free memory currently available (i.e., it will not immediately induce excessive swapping). We further constrained Redline to reactivate only one task per period of an observation window to avoid reactivating tasks too aggressively.

6.2 The EDF Scheduling Algorithm

Redline extends the Linux fair queueing proportional share scheduler, *CFS* [20]. Redline's EDF scheduler has its own set of per-CPU run queues just as the CFS scheduler does. Redline inserts interactive tasks into the run queues of *both* the CFS and EDF schedulers so that each interactive task can receive any unused CPU bandwidth after consuming its reserved bandwidth. The EDF scheduler has precedence over the CFS scheduler. During a context switch, Redline first invokes the EDF scheduler. If the EDF scheduler has no runnable tasks to schedule, then Redline invokes the CFS scheduler.

Redline maintains the following information for each interactive task: the *starttime* and *deadline* of the current reservation period, and the remaining entitled computation time (*budget*). As a task executes, the EDF scheduler keeps track of its CPU usage and deducts the amount consumed from *budget*. The EDF scheduler checks whether to assign a new reservation period to a task at the following places: when a new task is initialized, after a task's budget is updated, and when a task wakes up from sleep. If the task has consumed its budget or passes its deadline, the EDF scheduler assigns a new reservation period to the task using the algorithm in Listing 1.

Listing 1 Assign a new reservation period to task p

```
1:  /* has budget, deadline not reached */
2:  if ($budget > 0$) && ($now < deadline$) then
3:      return
4:  end if
5:  /* has budget, deadline is reached */
6:  if ($budget > 0$) && ($now \geq deadline$) then
7:      if has no interruptible sleep then
8:          return
9:      end if
10: end if
11:
12: /* no budget left: assign a new period */
13: dequeue($p$)
14: $starttime \leftarrow max(now, deadline)$
15: $deadline \leftarrow starttime + T$
16: $budget \leftarrow max(budget + C, C)$
17: enqueue($p$)
```

A task may reach its deadline before expending the budget (see line 6) for the following reasons: it did not actually have enough computation work to exhaust the budget in the past period; it ran into a CPU overload; or it experienced non-discretionary delays, such as page faults or disk I/O.

The EDF scheduler differentiates these cases by checking whether the task voluntarily gave up the CPU during the past period (i.e., had at least one interruptible sleep, see line 7). If so, the EDF scheduler assigns a new period to the task. Otherwise, it considers that the task missed a deadline and pushes its work through as soon as possible.

If a task consumes its budget before reaching the deadline, it will receive a new reservation period. But the start time of this new period is later than the current time (see line 14). The EDF scheduler considers a task *eligible* for using reserved CPU time only if $starttime \leq now$. Therefore, it will not pick this task for execution until its new reservation period starts. This mechanism prevents a interactive task from consuming more than its entitlement and thereby interfering with other interactive tasks.

At any context switch, the EDF scheduler always picks for execution the eligible task that has the earliest deadline. We implemented its run queue using a tagged red-black tree similar to the binary tree structure proposed in EEVDF [28]. The red-black tree is sorted by the start time of each task. Each node in the tree has a tag recording the earliest deadline in its subtree. The complexity of its enqueue, dequeue, and select operations are all $O(\log n)$, where n is the number of runnable tasks.

6.3 SMP Load Balancing

Load balancing in Redline is quite simple, because the basic CPU bandwidth needs of an interactive task will be satisfied once it is accepted on a CPU. It is not necessary to move an accepted task unless that CPU is overloaded. The only thing Redline has to do is select a suitable CPU for each new interactive task during calls to `exec()`. Redline always puts a new task on the CPU that has the lowest R_{load} at the time. Once the task passes the admission test, it stays on the same CPU as long as it remains accepted. If the CPU becomes overloaded, Redline will revoke at least one interactive task tied to that CPU. Once turned into best-effort tasks, revoked tasks can be moved to other CPUs by the load balancer. When a revoked task wakes up on a new CPU, Redline will attempt to reactivate its specification if there are adequate resources there.

7 Discussion

In this section, we discuss several design choices for Redline, and we address alternatives that may merit further investigation.

Specifications: Setting specifications in Redline does not require precise, *a priori* application information. Since Redline's admission control takes the current CPU load into account, a user or system administrator can modestly over-specify a task's resource needs. The only danger of over-specification is that a newly launched task may be rejected by the admission control if the system is sufficiently loaded with other interactive tasks. Once admitted, an interactive task is managed according to its real usage, negating the impact of the over-specification. If an interactive task is under-specified, it will at least make steady progress with the reserved CPU bandwidth allocated to it. Furthermore, if the system is not heavily loaded, an under-specified interactive task will also allocated some of the remaining CPU bandwidth along with the best-effort tasks. Thus, even under-specified tasks become poorly responsive only if the system load is high.

We therefore believe that specification management should not be an obstacle. A simple tool could allow a user to experimentally adjust the reservations (both C and T), finding minimal requirements for acceptable performance. Redline could also be made to track and report actual CPU bandwidth usage over time, allowing fine-tuning of the specifications.

Memory Management: For any task to be responsive, the operating system must cache its working set in main memory. Many real-time systems conservatively "pin" all pages of a task, preventing their removal from main memory to ensure that the working set must be cached. In contrast, Redline protects any page used by an interactive task within the last π seconds, thus protecting the task's working set while allowing its inactive pages to be evicted. In choosing a value for π, it is important not to make it too small, causing the system to be behave like a standard, commodity OS. It is safer to overestimate π, although doing so makes Redline behave more like a conservative real-time system. This method of identifying the working set works well under many circumstances with reasonable choices of π.

However, there are other mechanisms that estimate the working set more accurately. By using such mechanisms, Redline could avoid the dangers caused by setting π poorly. One such alternative is the working set measurement used in CRAMM [32]. CRAMM maintains reference distribution histograms to track each task's working set on-line with low overhead and high accuracy. While we believe that this approach is likely to work well, it does not guarantee that the working set is always properly identified. Specifically, if a task performs a *phase change*, altering its reference behavior suddenly and significantly, this mechanism will require a moderate period of time to recognize the change. During this period, CRAMM could substantially under- or over-estimate application working set sizes. However, we believe such behavior is likely to be brief and tolerable in the vast majority of cases. We intend to integrate the CRAMM VMM into Redline and evaluate its performance.

8 Experimental Evaluation

Figure 1 shows that the execution of a single compilation command (`make -j32`) significantly degrades responsiveness on a standard system, while Redline is able to keep the system responsive and play the video smoothly. In this section, we further evaluate the performance of Redline implementation by stressing the system with a variety of extreme workloads.

Platform: We perform all measurements on a system with a 3 GHz Pentium 4 CPU, 1 GB of RAM, a 40GB FUJITSU 5400RPM ATA disk, and an Intel 82865G integrated graphics card. The processor employs *symmetric multithreading (SMT)* (i.e., Intel's HyperThreading), thus appearing to the system as two processors. We use a Linux kernel (version 2.6.22.5) patched with the CFS scheduler (version 20.3) as our control. Redline is implemented as a patch to this same Linux version. For all experiments, the screen resolution was set to 1600 x 1200 pixels.

All experiments used both of the SMT-based virtual CPUs except when measuring the context switch overhead. Furthermore, we ran each experiment 30 times, taking both the arithmetic mean and the standard deviation of all timing measurements.

Application Settings and Inputs: Table 1 shows a subset of the task specifications used for the experiments under Redline. It includes the *init* process, *kjournald*, the X11 server *Xorg*, KDE's desktop/window manager, the *bash* shell, and several typical interactive applications. We left the memory protection period (π) and I/O priority empty in all the specifications, letting Redline choose them automatically.

	C:T (ms)		C:T (ms)
init	2:50	kjournald	10:100
Xorg	15:30	kdeinit	2:30
kwin	3:30	kdesktop	3:30
bash	5:100	vim	5:100
mplayer	5:30	firefox	6:30

Table 1: A subset of the specifications used in the Redline experiments.

The movie player, *mplayer*, plays a 924.3 Kb/s AVI-format video at 25 frames per second (f/s) with a resolution of 520 x 274. To give the standard Linux system the greatest opportunity to support these interactive tasks, we set *mplayer*, *firefox*, and *vim* to have a CPU scheduler priority of -20—the highest priority possible. Also note that a pessimistic admission test, like that of a real-time system, would not accept all of the applications in Table 1, because they would over-commit the system. Redline, however, accepts these as well as many other interactive tasks.

8.1 CPU Scheduling

Scheduler Overhead: We compare the performance of the Redline EDF scheduler with the Linux CFS scheduler. Figure 4(a) presents the context switch overhead for each scheduler as reported by *lmbench*. From 2 to 96 processes, the context switch time for both schedulers is exceedingly comparable.

However, when lmbench measures context switching time, it creates a workload in which exactly one task is unblocked and ready to run at any given moment. This characteristic of lmbench may be unrepresentative of some workloads, so we further compare these schedulers by running multiple busy-looping tasks, all of which would be ready to run at any moment. We tested workloads of 1, 20, 200, and 2,000 tasks. We perform this test twice for Redline, launching best-effort tasks to test its CFS scheduler, and launching interactive tasks to test its EDF scheduler. In the latter case, we assigned specification values for C:T as 1:3, 1:30, 1:300 and 1:3,000 respectively, thus forcing the Redline EDF scheduler to perform a context switch almost every millisecond. Note that with these specification values, the Redline EDF scheduler is invoked more frequently than a CFS scheduler would be, running roughly once per millisecond (whereas the usual Linux CFS quanta is 3 ms).

The number of loops are chosen so that each run takes approximately 1,400 seconds. Figure 4(b) shows the mean total execution time of running these groups of tasks with Linux CFS, Redline CFS, and Redline EDF. In the worst case, the Redline EDF scheduler adds 0.54% to the running time, even when context switching far more frequently than the CFS schedulers. Note that the total execution time of these experiments was bimodal and, as shown by the error bars, made the variance in running times larger than the difference between results. The overhead of using the Redline schedulers is negligible.

Fork Bombs: We now evaluate the ability of Redline to maintain the responsiveness of interactive tasks. First, we launch mplayer as an interactive task, letting it run for a few seconds. Then, we simultaneously launch many CPU-intensive tasks, thus performing a fork bomb. Specifically, a task forks a fixed number of child tasks, each of which executes an infinite loop, and then kills them after 30 seconds. We performed two tests for Redline: the first runs the fork bomb tasks as best-effort, and the second runs them as interactive. In the latter case, the fork bomb tasks were given CPU bandwidth specifications of 10:100. For the Linux test, the interactive task had the highest possible priority (-20), while the fork bomb tasks were assigned the default priority (0).

Figure 5 shows the number of frames rate of achieved by mplayer during these tests. In Figure 5(a), the fork bomb comprises 50 tasks, while Figure 5(b) shows a 2,000-task fork bomb. Both of these figures show, for Redline, best-effort and interactive fork bombs. Under Linux with 50 tasks, mplayer begins normally. After approximately 10 seconds, the fork bomb begins and mplayer receives so little CPU bandwidth that its frame rate drops nearly to zero. Amusingly, after the fork bomb terminates at the 40 second mark, mplayer "catches up" by playing frames at more than triple the normal rate. For Linux, the 2,000-task fork bomb has the identical effect on mplayer. The load is so high that

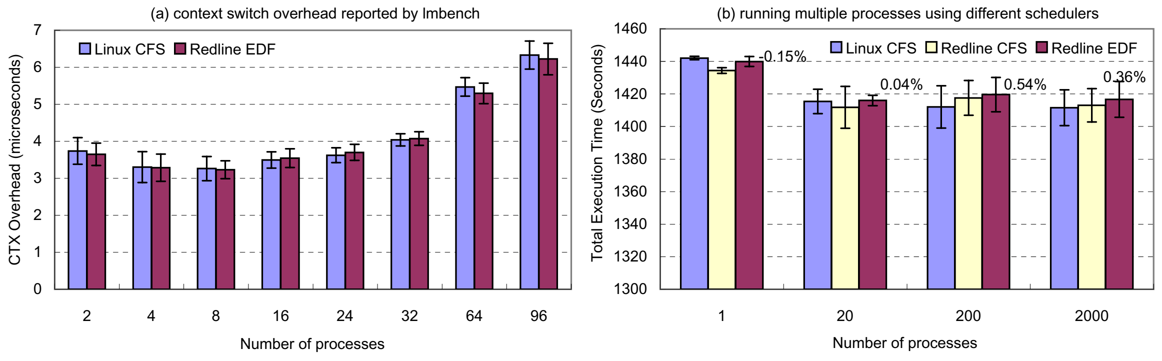

Figure 4: An evaluation of CPU scheduling overhead. Figure (a) shows the context switching times as evaluated by *lmbench*. Figure (b) shows the total running time of varying numbers of CPU intensive tasks. Note the y-axis starting at 1,300 to make the minor variation visible.

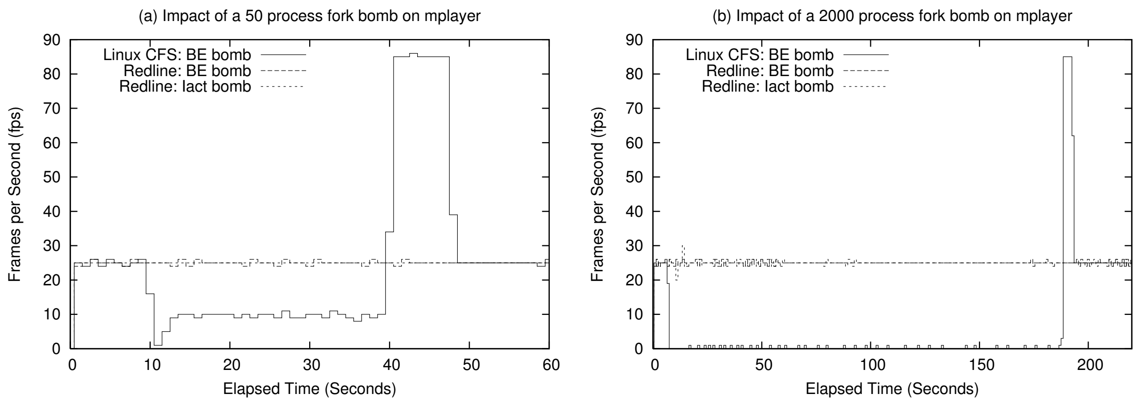

Figure 5: Playing a video and launching fork bombs of (a) 50 or (b) 2,000 (b) tasks.

even the fork bomb's parent task is unable to kill all of the children after 30 seconds, delaying the "catch-up" phase of mplayer until approximately the 190 second mark. In fact, the whole Linux system becomes unresponsive, with simple tasks like moving the mouse and switching between windows becoming so slow that human intervention is impossible.

In Redline, these fork bombs, whether run as best-effort or interactive tasks, have a negligible impact on mplayer. Only the 2,000 task interactive fork bomb briefly degrades the frame rate to 20 f/s, which is a barely perceptible effect. This brief degradation is caused by the 1 ms period that Redline gives to each newly forked task before performing an admission test, leaving the system temporarily overloaded.

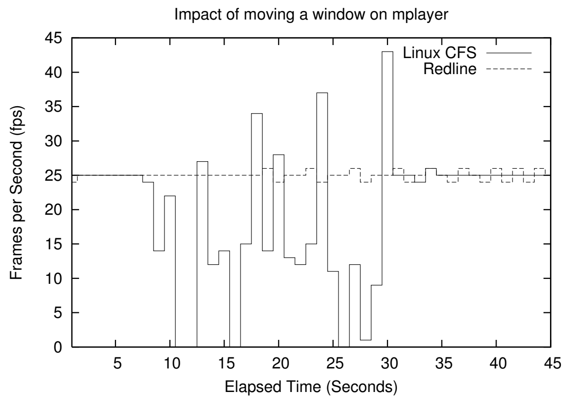

Figure 6: Playing video while dragging around a window.

Competing Interactive Tasks: In order to see how interactive tasks may affect each other, we launch mplayer, and then we have a user drag a window in a circle for 20 seconds. Figure 6 shows that, under Linux, moving a window has substantial impact on mplayer. Because we use a high screen resolution and a weakly powered graphics card, Xorg requires a good deal of CPU bandwidth to update the screen. However, the CFS scheduler gives the same CPU share to all runnable tasks, allowing the window manager to submit screen update requests faster than Xorg can process them. When mplayer is awakened, it has to wait until

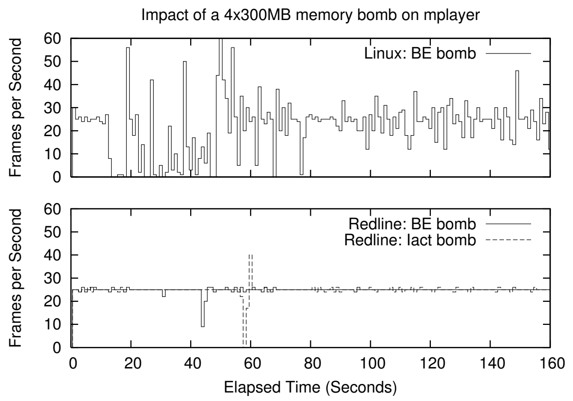

Figure 7: Playing video with 4 x 300 MB memory bomb tasks. The frame rate is severely erratic under Linux, but is steady under Redline.

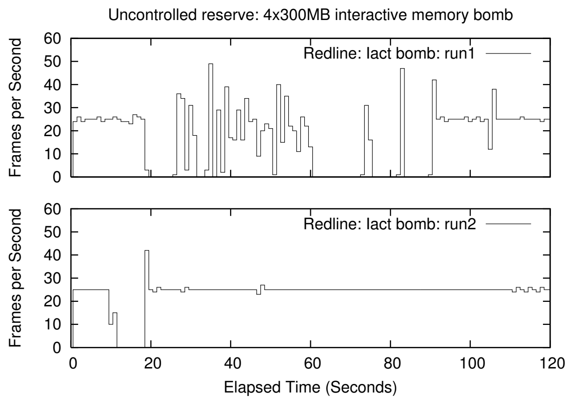

Figure 8: Playing a movie with 4 x 300 MB interactive memory bomb tasks on a Redline system without the rate controlled memory reserve.

all other runnable tasks make sufficient progress before it is scheduled for execution. Moreover, its requests are inserted at the end of Xorg's backlogged service queue. Consequently, the frame rate of mplayer becomes quite erratic as it falls behind and then tries to catch up by submitting a group of frame updates in rapid succession.

In Redline, mplayer plays the movie smoothly no matter how quickly we move the window, even though Xorg and all of the tasks comprising the GUI are themselves interactive tasks. We believe that because Xorg effectively gets more bandwidth (50% reserved plus proportional sharing with other tasks), and the EDF scheduler causes mplayer add its requests into Xorg's service queue earlier.

8.2 Memory Management

Memory Bombs: We simulate a workload that has a high memory demand by using memory bombs. This experiment creates four tasks, each of which allocates 300 MB of heap space and then repeatedly writes to each page in an infinite loop. For Redline, we perform two experiments: the first launches the memory bombs as best-effort tasks, and the second launches them as interactive ones using specification 10:100.

The upper part of Figure 7 shows, for Linux, the frame rate for mplayer over time with the memory bomb tasks running. The frame rate is so erratic that movie is unwatchable. Both video and audio pause periodically. The memory bomb forces the VMM to swap out many pages used by GUI applications, making the system as a whole unresponsive. As shown by the lower part of Figure 7, under Redline, mplayer successfully survives both the best-effort and interactive memory bombs. Each of them only leads to one brief disruption of the frame rate (less than 3 seconds), which a user will notice but is likely to tolerate. The system remains responsive, allowing the user to carry out GUI operations and interact as usual.

The Rate Controlled Reserve: In order to demonstrate the importance of the rate controlled reserve, we remove it from Redline and repeat the interactive memory bomb experiment. Figure 8 shows how mplayer behaves in two different runs. In the first, memory-demanding interactive tasks quickly exhaust the free memory, forcing others tasks to reclaim pages when allocating memory. Therefore, mplayer is unable to maintain its frame rate. At approximately the 90 second mark, the Redline VMM finally demotes an interactive task to the best-effort class, and then the frame rate of mplayer stabilizes. Depending on when and which tasks the Redline VMM chooses to revoke, the interactive memory bomb can prevent the system from being responsive for a long period of time. Here, more than one minute passes before responsiveness is restored. However, during the second run, mplayer was unresponsive for only about 10 seconds, thanks to a different selection of tasks to demote. Ultimately, the limited isolation among interactive tasks provided by this small rate controlled reserve is crucial to the maintaining a consistently responsive system.

Speed-bump Pages: To examine the effectiveness of Redline's speed-bump page mechanism, we first start one 500 MB memory bomb task. After a few seconds, we launch a second 500 MB memory bomb. Under Linux, we set this second task's priority to be -20. Under Redline, we launch it as an interactive task whose specification is set to 10:100. Figure 9 presents the *resident set sizes (RSS)*—the actual number of cached pages—for each task over time. Under Linux, the second task, in spite of its high priority, is never allocated its complete working set of 500 MB. Here, the first task dirties pages too fast, preventing the Linux VMM from ever reallocating page frames to the higher priority task. However, under Redline, the second task is quickly allocated space for its full working set, stealing pages from the first, best-effort task.

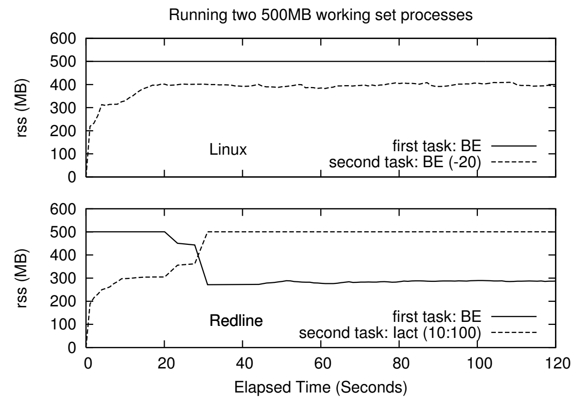

Figure 9: Competing memory bomb tasks. Under Linux, the lower-priority background task prevents the higher-priority foreground task from caching its working set.

As an alternative test of this mechanism, we launched four 300 MB memory bombs, and then launched the firefox web browser. Under Redline, firefox was ready and responsive after 30 seconds; under a standard Linux kernel, more than 4 minutes passed before the browser could be used.

8.3 Disk I/O Management

We examine the effectiveness of Redline's disk I/O management by running tasks that perform intensive disk I/O.

Writing: To test disk writing operations, we launch two background tasks meant to interfere with responsiveness. Specifically, each task repeatedly writes to an existing, 200 MB file using buffered writes. We then use vim, modified to report the time required to execute its write command, to perform sporadic write requests of a 30 KB file. It is set to the highest priority (-20) under Linux, while it is launched as an interactive task with a specification of 5:100 under Redline.

For Linux, each transaction in the journaling file system is heavily loaded with dirty pages from the background tasks. Thus, the calls to fsync() performed by vim causes it to block for a mean of 28 seconds. Under Redline, the reduced dirty threshold for best-effort tasks forces the system to flush the dirtied pages of the background task more frequently. When vim calls fsync(), the transaction committed by the journaling file system takes much less time because it is much smaller, requiring a mean of only 2.5 seconds.

Reading: We play a movie using mplayer in the foreground while nine background tasks consume all of the disk bandwidth. Each background task reads 100 MB from disk in 20 MB chunks using direct I/O (bypassing the file system cache to achieve steady I/O streams). Figure 10 shows the number frame rate of mplayer over time for both Linux and Redline. Under Linux, mplayer blocks frequently because of the pending I/O requests of the background tasks, thus making the frame rate severely degraded and erratic. Redline automatically assigns higher I/O priorities to the interactive task, allowing it to maintain its framerate throughout.

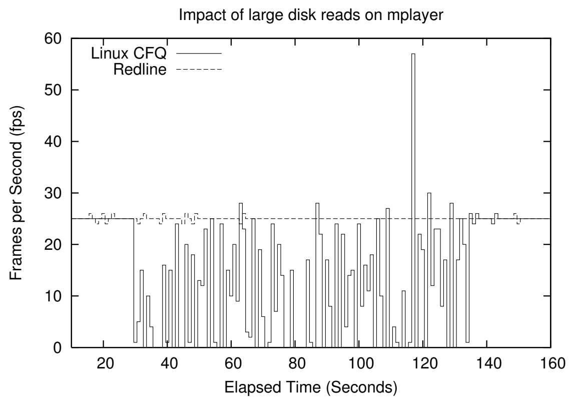

Figure 10: The impact of massive reads on mplayer.

9 Related Work

We now briefly discuss related work in the areas of scheduling, virtual memory management, and I/O management. Table 2 summarizes the characteristics of several representative CPU schedulers and operating systems, and compares them to Redline.

CPU Scheduling: Neither time-sharing schedulers (e.g., used by Linux, FreeBSD, Solaris, Windows) nor proportional-share schedulers [10, 31, 27, 28, 22] that simulate the ideal GPS model [24] provide adequate support to address response time requirements. These systems typically employ heuristics to improve the responsiveness of interactive tasks. For example, FreeBSD boosts the priorities for I/O-bound tasks, and Windows Vista does the same for tasks running in its multimedia class.

Several extended general-purpose schedulers provide performance isolation across scheduling classes. A-SFQ [26] dynamically adjusts weights to maintain stable CPU bandwidth for the class serving soft real-time tasks. BVT [8] and BERT [3] achieve the same goal by adjusting virtual time or deadlines, while BEST [1] and SMART [21] incorporate EDF into proportional share schedulers. However, without appropriate admission and load control, none of these systems can provide CPU bandwidth guarantees under heavy workloads.

Unlike these general-purpose systems, real-time systems often impose draconian admissions control and enforcement mechanisms in order to provide strict performance guarantees. Examples include Nemesis [16], CPU service class [6], Linux/RK [23], Rialto [15], Lin et al.'s scheduler [11], and work by Deng et al. [7] and PSheED [17]. While specification-based admission tests pessimistically reject tasks in exchange for strict guarantees, Redline adapts to the actual system workload, maximizing resource

		Admission control	**Performance isolation**		**Intg. Mgmt.**		**Without app. mod.**
			interclass	**intraclass**	**mem**	**I/O**	
CPU Scheduler	Stride [31],EEVDF [28], VTRR [22],PD [27]	×	×	×	n/a	n/a	√
	SFQ [10], A-SFQ [26]	×	strong	×	n/a	n/a	√
	BVT [8], BERT [3]	×	strong	weak	n/a	n/a	×
	BEST [1]	×	weak	weak	n/a	n/a	√
	SMART [21]	×	strong	strong	n/a	n/a	×
	PSheED [17], Deng et al. [7]	pessimistic	strict	strict	n/a	n/a	×
Operating Systems	Linux, FreeBSD, Solaris, Windows	×	×	×	×	×	√
	Solaris Container [19], Eclipse [4], SPU [30]	×	strong	×	√	√	√
	QLinux [29]	×	strong	×	×	√	√
	Linux-SRT [5]	pessimistic	strong	×	×	√	×
	Rialto [15]	pessimistic	strict	strict	×	√	×
	Nemesis [16]	pessimistic	strict	strict	×	√	×
	Redline	load based	strong	dynamic	√	√	√

Table 2: A comparison of Redline to other CPU schedulers and operating systems

utilization while accommodating as many interactive tasks as possible.

Memory Management: A number of memory managers employ strategies that improve on the traditional global LRU model still used by Linux. The Windows virtual memory manager adopts a per-process working set model, with a kernel thread that attempts to move pages from each process's working set onto a *standby* reclamation list, prioritized based on task priorities. In Zhou et al. [33], the virtual memory manager maintains a miss ratio curve for each process and evicts pages from the process that incurs the least penalty. Token-ordered LRU [14] allows *one* task in the system to hold a token for a period of time to build up its working set. CRAMM [32] and Bookmarking GC [12] use operating system support that allows garbage collected applications avoid page swapping. Unlike Redline, none of these memory managers can prevent the system from evicting too many pages from interactive applications.

Disk I/O Management: Traditional disk I/O subsystems are designed to maximize the overall throughput, rather than minimizing response time. The widely used SCAN (Elevator) algorithm sorts I/O requests by sector number to avoid unnecessary seeks. Anticipatory I/O [13] improves throughput even further by delaying I/O service so it can batch a number of I/O requests. Some operating systems (e.g. Linux and Windows) can prioritize I/O requests according to task priorities. This mechanism has little effect in practice, since most applications are launched with the same default priority. While Redline's I/O scheduler is effective, I/O schedulers developed for soft real time systems, such as R-SCAN in Nemesis [16], Cello in QLinux [29] and DS-SCAN in IRS [9], could be incorporated into Redline to enable more precise control over I/O bandwidth.

Integrated Resource Management: Like Redline, Nemesis [16], Rialto [15] and Linux-SRT [5] use *CPU reservations* to provide response time guarantees and include specialized I/O schedulers. However, both systems employ pessimistic admission policies that reduce their ability to handle a large number of aperiodic tasks with unpredictable workloads. More importantly, these systems largely punt on the thorny issue of memory management: Nemesis allocates a certain amount of physical memory to each task according to its contract and lets each task manage its own memory via *self-paging*, while Rialto simply locks all memory pages used by real time tasks. Similarly, QLinux [29] divides tasks into classes and uses a hierarchical SFQ scheduler to manage CPU and network bandwidth together with Cello as its I/O scheduler, but does not address the problem of memory management.

Solaris Containers [19] combine system resource management with a boundary separation provided by *zones*. Each zone can be configured to have a dedicated amount of resources and then acts as a completely isolated virtual server. Eclipse [4] and SPU [30] follow a similar approach, partitioning resources *statically* and ignoring the issue of responsiveness. Moreover, it is highly impractical to create one zone/container for every possible interactive application due to complex interactions among them and the tremendous configuration effort that would be required. By contrast, Redline's adaptive approach is simpler and better matches the needs of supporting interactive applications in commodity operating systems.

IRS [9] and Resource Containers [2] also provide integrated management of multiple resources with a focus on providing real time service to multiple clients, although IRS does not address memory management. However, unlike Redline, these systems require that all applications in a system explicitly state their resource needs, requiring complex modifications to existing applications.

10 Conclusion

We present Redline, a system designed to support highly interactive applications in a commodity operating system environment. Redline combines lightweight specifications with an integrated management of memory, disk I/O, and CPU resources that delivers responsiveness to interactive applications even in the face of extreme workloads.

The Redline system (built on Linux) is open-source software and may be downloaded at `http://redline.cs.umass.edu`.

11 Acknowledgements

This material is based upon work supported by the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. A. Banachowski and S. A. Brandt. Better real-time response for time-share scheduling. In *Proc. of the* 11th *WPDRTS*, page 124.2, 2003.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the* 3rd *OSDI*, pages 45–58, 1999.

[3] A. Bavier, L. Peterson, and D. Mosberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, 1999.

[4] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proc. of the 1998 USENIX*, pages 235–246, 1998.

[5] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proc. of the* 7th *RTAS*, pages 135–140, 2001.

[6] H.-H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proc. of the* 6th *ICMCS, Vol. 1*, pages 296–301, 1999.

[7] Z. Deng, J. Liu, L. Y. Zhang, M. Seri, and A. Frei. An open environment for real-time applications. *Real-Time Systems*, 16(2-3):155–185, 1999.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose schedular. In *Proc. of the* 17th *SOSP*, pages 261–276, 1999.

[9] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduing for periodic soft real-time applications. In *Proc. of the* 9th *MMCN*, pages 34–45, Berkeley, CA, 2002.

[10] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the* 2nd *OSDI*, pages 107–121, Seattle, WA, 1996.

[11] C. han Lin, H. hua Chu, and K. Nahrstedt. A soft real-time scheduling server on the Windows NT. In *Proc. of the* 2nd *USENIX Windows NT Symposium*, pages 149–156, 1998.

[12] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proc. of the 2005 PLDI*, pages 143–153, 2005.

[13] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc. of the* 18th *SOSP*, pages 117–130, 2001.

[14] S. Jiang and X. Zhang. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Perform. Eval.*, 60(1-4):5–29, 2005.

[15] M. B. Jones, D. L. McCulley, A. Forin, P. J. Leach, D. Rosu, and D. L. Roberts. An overview of the Rialto real-time architecture. In *Proc. of the* 7th *ACM SIGOPS European Workshop*, pages 249–256, 1996.

[16] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[17] G. Lipari, J. Carpenter, and S. K. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proc. of the* 21st *RTSS*, pages 217–226, 2000.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[19] J. Mauro and R. McDougall. *Solaris Internal: Core Kernel Components*. Sum Microsystems Press, A Prentice Hall Title, 2000.

[20] I. Molnar. http://people.redhat.com/mingo/cfs-scheduler/.

[21] J. Nieh and M. S. Lam. SMART: A processor scheduler for multimedia applications. In *Proc. of the* 15th *SOSP*, page 233, 1995.

[22] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) proportional share scheduler. In *Proc. of the 2001 USENIX*, pages 245–259, 2001.

[23] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proc. of the* 5th *RTAS*, pages 111–120, 1999.

[24] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. In *Proc. of IEEE INFOCOM*, 1992.

[25] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proc. of the 2005USENIX*, pages 105–120, 2005.

[26] M. A. Rau and E. Smirni. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *Proc. of the* 7th *MASCOTS*, page 252, Washington, DC, 1999.

[27] A. Srinivasan and J. H. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of System Software*, 77(1):67–80, 2005.

[28] I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first : A flexible and accurate mechanism for proportional share resource allocation. Technical Report TR-95-22, Old Dominion University, 1995.

[29] V. Sundaram, A. Chandra, P. Goyal, P. J. Shenoy, J. Sahni, and H. M. Vin. Application performance in the QLinux multimedia operating system. In *Proc. of the* 8th *ACM Multimedia*, pages 127–136, 2000.

[30] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proc. of the* 8th *ASPLOS*, pages 181–192, 1998.

[31] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TR-528, MIT Laboratory of CS, 1995.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. of the* 7th *OSDI*, pages 103–116, 2006.

[33] P. Zhou, V. Pandy, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curves for memory management. In *Proc. of the* 11th *ASPLOS*, pages 177–188, Boston, MA, Oct. 2004.