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 do little to ensure responsiveness. Despite of their best-effort priority-based scehdulers that provide no bounds on delays, their resource managers (especially memory managers, disk I/O schedulers) are completely oblivious to response time requirements. Therefore, pressure on any 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 to 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 load to maximize responsiveness and system utilization. We show that Redline delivers responsiveness to interactive applications even in the face of extreme workloads including fork bombs, malloc bombs and bursty large disk I/O requests, reducing application pauses by up to two orders of magnitude.

1 Introduction

The enormous advances in processing power, memory size, storage capacity, and network bandwidth of computers 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.

Unfortunately, existing general-purpose operating systems do not provide adequate support for these modern applications. Current operating systems, like Windows, Linux, and Solaris were not designed with interactivity in mind. The memory manager and I/O subsystem of these systems all work independently from the CPU scheduler, maximizing the overall system throughput but ignoring application's response time requirements. Consequently, pressure on any one subsystem can significationly 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 their best-effort, priority-based schedulers are a good match for batch-style applications, they provide limited support for ensuring responsiveness.

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 a lightweight *specification*-based approach that provides enough information to allow it to meet the response time requirements of interactive applications. Redline's specifications, which extend Rialto's CPU specifications [15], give a rough estimate of the amount of resources required by an application over any period of time 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 latency-sensitive daemons in just one day. In an actual deployment, we expect these to be provided by application developers.

Each resource manager then uses these specifications to

inform its decisions. Redline's memory manager protects the working sets of interactive applications according to their specifications, preferentially evicts pages from non-interactive applications, and further reduces the risk of paging through a rate-controlled memory reserve (Section 4). Redline's disk I/O manager avoids pauses in interactive applications by dynamically prioritizing these tasks based on their specifications (Section 5). Finally, 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).

By contrast with 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.

We have implemented Redline as an extension to the Linux kernel (see Section 7 for a full discussion). 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 malloc bombs.

2 Redline Overview

Figure 1 presents an overview of the Redline system. The first component is specification management. It allows the system administrator to pick a set of important applications and give their specifications, which are stored in a special file. Redline loads specifications from that file whenever an application is launched. Redline also exposes a system call interface that allows users to change an application's specification during execution.

Redline divides tasks into four types based on their processing requirements:

1. **Real-time (RT)**: time critical tasks for which the consequence of missing a deadline is catastrophic to the system;

2. **Interactive (Iact)**: response-time sensitive tasks that provide services in response to external requests/events. These include not only typical interactive tasks, but also tasks that serve requests from other tasks, such as kernel threads and daemons;

3. **Throughput (Tput)**: tasks that require a certain amount of CPU bandwidth over the long-term, without the need for short-term responsiveness; and

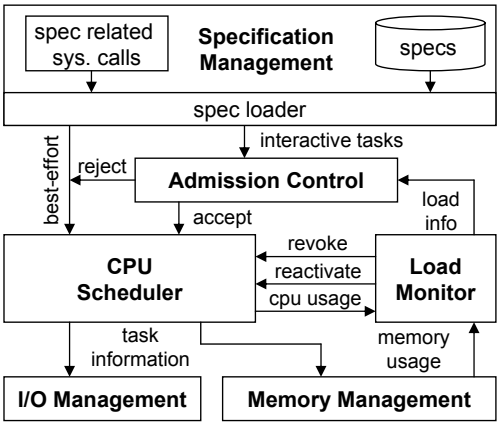

Figure 1: The Redline system.

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

This paper focuses on Redline's support for interactive (Iact) and best-effort (BE) tasks. Redline treats any application without a specification as a best-effort task.

Whenever a new task is launched, Redline performs admissions control to see whether the system can accommodate it. Instead of solely relying specifications, which would pessimistically reject most tasks, Redline uses an optimistic admission control mechanism in conjunction with dynamic load control. It monitors the CPU bandwidth consumed by interactive tasks, and accepts a new task as long as its requirements (together with current load) do not exceed a certain threshold. When Redline detects an overload, the load monitor chooses a victim to downgrade. This victim is the task that acts least like an interactive task (i.e., it is the most CPU-intensive). The CPU scheduler then downgrades the victim's specification, turning it into a best-effort task. This strategy allows other interactive tasks to continue to meet their response-time requirements. Whenever more resources become available, Redline will reactivate the specification of a previously-revoked task.

Once the admissions control mechanism accepts a task, the CPU scheduler activates the loaded specification. It also propagates the specification to the other resource managers, so that they can handle each type of tasks accordingly. The memory manager attempts to protect their working sets and allows them to evict pages from best-effort tasks if necessary. It also maintains a rate-controlled memory reserve that provides limited isolation among interactive tasks under servere memory pressure. 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. Redline's integrated resource management, combined with

appropriate admission and load control, effectively maintains interactive responsiveness under heavy resource contention.

3 Redline Specifications

While Redline executes applications without specifications in best-effort mode, it can use specifications to guarantee their responsiveness. We have defined specifications for a wide class of services. These specifications not only ensure the responsiveness of interactive applications like text editors, movie players, and web browsers. By specifying the requirements of applications that support the graphical user interface, including the X server, window/desktop manager, etc., Redline ensures that the GUI remains responsive. Its specifications for a range of kernel threads and daemons allows Redline to ensure that the system as a whole remains stable. Finally, we have generated specifications for a range of administrative tools (e.g., *bash*, *top*, *ls*, and *kill*) that allow users to manage their applications as well as the system even under extreme load.

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

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

The first field is the path name to the executable, and the second field is its type (usually lact, for interactive). Two important flags include I, whether the specification is inherited by a child process, and R, which indicates if 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:

$$\langle \text{/usr/bin/mplayer:lact:5:30:IR:-:-} \rangle$$

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

Setting Specifications: We derived specifications for a range of interactive applications by following several simple rules. Most administrative tools are very short-lived, so reserving a few percent of CPU bandwidth out of every several hundred milliseconds is sufficient to guarantee responsiveness. Most kernel threads and daemons are not CPU intensive, but very response time sensitive, so their reservation period should be in the tens of milliseconds. The reservation period for a movie player should be around 30ms to ensure 30 frames per second, which implies that the X server and window/desktop manager should also use the same reservation period (if not smaller). We found that setting specifications was straightforward. With little work, we were able to manually generate a set of specifications for about 100 applications in a Linux system with the K Desktop Environment (KDE). This specification file is portable and could easily be shipped with a Linux distribution.

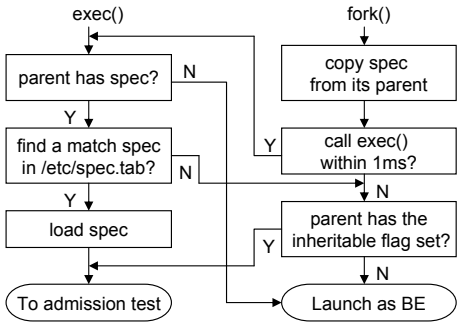

Figure 2: Loading Specifications in Redline

Loading Specifications: Redline stores its specifications in a file (/etc/spec/spec.tab). An interactive task uses either the specification loaded during exec() or one inherited from its parent. Figure 2 shows how Redline loads the specification for a task.

In practice, most tasks either invoke exec() soon after being forked, or never invoke it. Therefore, during fork(), Redline copies the parent's specification and gives the new task 1ms of execution time. If the new task does not invoke exec() within this 1ms window, Redline checks the specification. If the inheritable flag is set, Redline performs an admission test on the inherited specification. Otherwise, the new task is launched in the best-effort category.

If, however, the new task does invoke exec() within the 1ms window, Redline first checks its parent. If the parent is a best-effort task without any specification, then the new task is set to be best-effort. Otherwise, Redline searches the specification table for the new task using its full path name. If it finds a match, Redline loads the specification and performs an admission test. If there is no match and the parent is not marked for inheritance, the new task is set to be best-effort. Notice that, in the absence of specifications, Redline behaves exactly like a normal Linux system .

4 Redline Virtual Memory Management

The goal of existing virtual memory managers (VMM) in commodity operating systems is to maximize overall system throughput. There is no notion of fairness to tasks, let alone any consideration of the priorities and shares used in the CPU scheduler. Most VMM's employ "use-it-or-lose-it" policies, under which memory referencing speed determines allocation: the more pages referenced per second, the large a main memory allocation the VMM provides.

A task that blocks on I/O—and an interactive task will routinely block on input—is more vulnerable to losing its allocation and then later being forced to page swap when it awakens. Because typical VMM's do not provide isolation among tasks, a single memory-intensive task can "steal" the allocations of other tasks that are not actively using their pages. Worse, page swapping itself is the kind of blocking I/O operation that can cause a task to lose more of its allocation to other processes that quickly references their pages.

Failing to supply required memory pages on time can seriously hurt the responsiveness of interactive tasks, due to the enormous overhead of loading pages from disk. Therefore, the Redline VMM is designed to satisfy the memory requirements of interactive tasks, thus keeping them responsive even after blocking on input. Informed with necessary information from the specifications, the Redline VMM uses three mechanisms to achieve these goals: protecting the working sets of interactive tasks, *booking* pages used by best-effort tasks, and a *rate-controlled memory reserve*.

Protecting working sets: The Linux VMM uses a page replacement algorithm that approximates a global least recently used (LRU) policy. Pages are approximately ordered by recency of use and with no consideration of the task to which each page belongs. Pages used longest ago are cleaned and selected for reclamation. 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 as a consequence.

For an in interactive task to remain responsive, a VMM must keep its *working set*—those pages that are currently in active use—resident. In the Linux VMM, there is no identification of, nor any attempt to cache, a task's working set. If the system-wide demand for main memory is large enough, and if a task does not run and reference its pages rapidly enough, then parts of its working set will be replaced. Under this memory pressure, interactive tasks 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. Additionally, each page has a timestamp c that records the last time the page's reference bit was cleared. A page is *expired* if $(t-c) > \pi$, where t is the current time. By default, $\pi = 30 \times 60$, or 30 minutes if π is not supplied by the specification. This default requires that a user ignore an interactive application for a substantial period of time before some of its working set is evicted and it becomes temporarily unresponsive.

The Redline VMM handles the pages of interactive tasks and best-effort tasks differently. If the faulting process is in the best-effort category, 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; unexpired pages are not considered. Note also that this mechanism only considers pages that are not recently used. Pages belonging to any process that have been recently used are not examined by this reclamation mechanism. Should an insufficient number of pages be reclaimed by this process, it is repeated as many times as necessary.

If an interactive task faults, the VMM first tries to use the rate-controlled memory reserve (described in more detail below). If this reserve is insufficient, reclamation proceeds as above for best-effort tasks, reclaiming all not recently used pages except the unexpired pages from interactive tasks. If this attempt at reclamation is also insufficient, an additional pass aggressively reclaims more best-effort pages regardless of how recently they have been used. If even this aggressive approach is insufficient, then Redline is at risk for failing to meet its deadlines, and thus the VMM revokes the specification of some interactive task, thus demoting it to the best-effort class. Having done so, it starts over, examining the rate-controlled memory reserve and then reclaiming pages as described above.

Booking best-effort pages: Sometimes a best-effort task may *dirty* (modify) pages faster than the VMM can clean them, thus preventing the VMM from evicting its pages. Such best-effort tasks can hold large amount of memory, preventing interactive tasks from building up their working sets, but contribute no real benefits to improving user experience and system responsiveness. In order to reclaim memory from such tasks, the VMM can *book* pages used by best-effort tasks during the aggressive reclamation phase described above. That is, when the VMM finds a page belonging to a best-effort task, it unmaps that page and then marks it. If the best-effort task then references that page, the VMM notices the mark and, before re-mapping the page, it 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.

Rate-controlled reserve: The Linux VMM uses *watermarks* to trigger page reclamation. If the pool of free memory falls below the watermark, page reclamation is triggered. This approach is not desirable for interactive tasks because, should such a task fault, it would block during the potentially lengthy reclamation process, even if it needs only a small number of pages. The Redline VMM thus maintains a small memory reserve (about 8MB in our implementation) for interactive tasks, and controls their speed of consuming pages in this reserve. Although an interactive task still may block during reclamation, the reserve makes that situation significantly less likely.

The Redline VMM offers 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 reserved page consumed, 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 a memory demanding task to quickly exhaust the reserve and affect other tasks, so the speed of a task consuming 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 $t_f + b \times c < t$, 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. This limited isolation among interactive tasks is very important for system responsiveness as shown in Section 8.

5 Redline Disk I/O Management

Like the VMM, the I/O manager of a general purpose OS's 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. We describe the Redline I/O manager, which manages I/O events in a way that allows responsive tasks to meet their deadlines.

Journaling: For Linux, *ext3* is a journaling file system and the default for most distributions. Like any journaling file system, its updates are commited in *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 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: interactive task P_i and best-effort task P_{be}. Now consider that both tasks 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 with 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 contain dirty pages from any file written by any task, and thus will contain the pages written by both P_i and P_{be}, even though they were written by different tasks and to different files.

In this example, P_{be} writes a large amount of data through `write()`, while P_i writes some small amount. Assume that after both tasks have performed these `write()` operations, and 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 blocks until both its own dirty pages and those of P_{be} are written to disk.

If the OS caches too many of the pages written by P_{be}, then the `fsync()` operation will force P_i to be noticeably unresponsive. This poor interaction between compound transactions and `fsync()` occurs not only for *ext3*, but also for *ResierFS*[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.

Best-effort and interactive tasks should not have the same dirty threshold. Redline assigns different dirty thresholds for each type of task (RT:10%, Iact:5%, Tput:2%). Addtionally, Redline further restricts this 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 systems with large numbers of dirty pages. Finally, Redline places the kernel task assigned to manage write operations for each file system (in Linux, *kjournald*) 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, *Complete Fairness Queuing (CFQ)*, contains a single *request queue* of 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, the "idle"), it has a fixed, maximum capacity. When a task submits a request for a new I/O operation, the *congestion control mechanism* examines only whether the request queue is full—there is 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.

To address this problem, Redline uses 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 requesting I/O to block.

The default request queue for CFQ is well structured for differentiating between various request types, but it still 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, from which it is scheduled by a typical *elevator* algorithm on the disk itself. This I/O 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 typical kind—is placed into the best-effort class, no matter which task submits 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 *Iact* and a *Tput* class to the request queue management, thus matching its CPU scheduler's 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 *Iact* class over those in the *Tput* class, thus ensuring isolation of the requests of Iact tasks from Tput tasks.

Additionally, the specification for a task, as described in 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 shorter deadlines. Finally, Redline creates a per-task queue within the given class, allowing the I/O scheduler provide some isolation between Iact 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

Existing commodity operating systems handle interactive and best-effort tasks using the same time-sharing scheduler. They usually have no admission control to avoid overloading, and do not provide necessary isolation among tasks to prevent them from interfering with each other. To address overloading and interference, Redline extends the Linux kernel with admission control to prevent accepting too many interactive tasks, and with load control for quick recovery from overloads, with a new Earliest Deadline First (EDF) scheduling class to serve interactive tasks.

6.1 Admission and Load Control

An admission control, ensuring necessary resource is available to the requesting task, is required by any system that needs to provide response time guarantees. While general operating systems have no admission control at all, Real-time systems perform their admission tests based solely on specifications. For example, the total reserved bandwidth must be less then 1.0 when using EDF, and less than 0.69 for RM [18], so that none of the accepted tasks will miss any of their deadlines. This provides overly strong isolation for general purpose systems and prohibits many aperiodic interactive tasks from co-existing in the system and sharing CPUs effectively. In Redline admission and load control is based on a different observation: *As long as the CPU bandwidth consumed by reservations (i.e., by the EDF scheduling class) is not too high, the system will be responsive. Most of the time users may not even notice the difference.*

Therefore, Redline incorporates the actual CPU consumption into its admission control. It attempts to keep the CPU load consumed by reservations within a controlled range. It does so using three policy controls: 1) admission of new EDF-class tasks; 2) revocation of EDF tasks when there is excessively high load; and 3) reactivating tasks for EDF service when the load is low. Each control has an associated load threshold: R_{hi} for admission, R_{max} for revocation, and R_{lo} for reactivation, where $R_{max} > R_{hi} > R_{lo}$. Roughly speaking, if the load exceeds R_{max}, Redline revokes EDF tasks; if the load exceeds R_{hi}, it stops admitting new EDF tasks; and if the load falls below R_{lo}, it tries to reactivate tasks. In addition to using the exponentially smoothed load average, *Rload*, Redline also keeps a history of recent *Rload* values, to provide information about variation of the load. In particular, it samples *Rload* once a second and keeps the four most recent samples. We chose a four second *observation window* as being long enough to smooth out short bursts, while limiting the period of time during which Redline's responsiveness will degrade without the system's taking aggressive action. We now consider the details of the three policy controls.

Admission Test: While *Rload* is a good measure of the actual load presented by *previously admitted* tasks, when admitting a *new* task, Redline starts with that task's bandwidth specification $B_i = C_i/T_i$. It maintains a best estimate of anticipated load, which we call *Sload*. When it admits task i, it increments *Sload* by B_i. However, as newly admitted tasks run, we wish to use their *actual* load in making future decisions. Therefore, Redline decays *Sload* towards *Rload* (by a factor of 0.75 each second). Before it admits task i it compares $max(Rload, Sload)+B_i$ against R_{hi}, and admits the task only if that new estimated load is less than R_{hi}. Redline also decrements *Sload* appropriately when tasks exit. *Sload* is helpful in preventing Redline from ad-

mitting too many new tasks before it sees the actual load they present (reducing true overcommitment), while *Rload* allows Redline to overcommit (compared to the B_i specifications) effectively.

Once Redline accepts a task, the scheduler activates its specification, and also selects the memory protection period and I/O priority for it (these are discussed in their respective sections of the paper). It propagates necessary information to other resource managers, so that they can handle interactive tasks in a proper manner.

Revocation: Due to Redline's semi-optimistic admission control, a CPU may become overloaded when some interactive tasks running on it suddenly change their behavior. Therefore, Redline dynamically revokes tasks if necessary to keep the CPU bandwidth consumed by reservations under control. In revoking, Redline uses the threshold R_{max}>R_{hi}. Specifically, if the sampled load exceeds R_{max} for *every* sample in the observation window for a CPU, Redline revokes tasks on that CPU until *Rload* falls below R_{hi}. In choosing tasks to revoke, Redline prefers to revoke a task that exhausted its reservation during the observation window (indicating that the task has become more CPU-bound and less interactive). However, if there are no such tasks, Redline revokes the task with the highest reserved bandwidth. Certain tasks are set to be invulnerable to revocation to preserve overall system responsiveness, namely kernel threads, daemons, *Xorg*, and the window/desktop manager.

Reactivation: If all slots in the observation window fall below R_{lo} on a CPU, Redline begins reactivating tasks for EDF scheduling. A task is eligble 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 every period of an observation window to avoid reactivating tasks too aggressively.

6.2 The EDF Scheduling Algorithm

Recently, Linux adopted a new fair queueing proportional share scheduler called CFS [20], which Redline extends. The new EDF scheduler component 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 inteactive tasks can receive extra CPU allocations after using up their entitled reservation. The EDF scheduling class has precedence over the CFS scheduling class. During a context switch, Redline first invokes the EDF scheduler to pick a task from its run queue. If the EDF scheduler does not select a task, then Redline invokes the CFS scheduler to pick a task to run.

Redline maintains the following information for each interactive task: the *starttime* and *deadline* of the current reservation period, and entitled computation time left (*budget*). As a task executes, the EDF scheduler keeps track of its CPU usage and deducts the amount consumed from *budget* (at every timer interrupt or context switch). 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 ≥ 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 ← max(now,deadline)
15: deadline ← starttime + T
16: budget ← 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 need of a 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 *Rload* 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.

Specification: We believe that using CPU reservations is a good choice because it does not require precise, a priori application information. Because 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. Specifically, a 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 interactive tasks. Once admitted, an interactive task is managed according to its real usage.

Because over-specification is reasonably safe, and because the specifications are simple and lightweight, specification management should not be an obstacle. A simple tool could allow a user to experimentally adjust the reservations (both C and T), finding the 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 reservations.

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 achieves this goal by protecting any page used by an interactive task within the last π seconds. The choice of π is important. If it is too small, then pages still in active use may be evicted, leaving an interactive task unable to meet its deadlines. If π is too large, then the VMM caches some pages after they have fallen into disuse, thus overestimating the working set size, reducing the number of interactive tasks that Redline will admit, and underutilizing the system. It is safer to overestimate π, but doing so makes Redline behave somewhat 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 a mechanism, Redline could avoid both dangers outlined above caused by setting π poorly.

One such alternative is the working set measurement used in CRAMM [32]. This system maintains reference distribution histograms to track each task's working set online 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, when a task performs a *phase change*, altering its reference behavior suddenly and significantly, this mechanism will require a modest period of time to recognize the change. During this period, both of the problems described above caused by under- or over-estimating the working set size would be possible. However, we believe such behavior is likely to be tolerably transitory in the vast majority of cases. We intend to integrate the CRAMM VMM into Redline and evaluate its performance.

8 Experimental Evaluation

In this section, we evaluate the performance of Redline implementation by examining its ability to handle various extreme workloads.

Platform: We perform all measurements on a system with a 3.00 GHz Pentium 4 CPU, 1 GB of RAM, a 40GB FUJITSU 5400RPM ATA notebook disk, and an Intel 82865G integrated graphic card. The processor employs *symmetric multithreading (SMT)* (i.e., Intel's Hyper-Threading), thus appearing to the system as two processors. For L1 caches, the processor has a 12 KB instruction cache and an 8 KB data cache. The L2 cache is unified and 512 KB. 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 all both of the SMT-based virtual CPU except when measuring the context switch overhead. Furthermore, we ran each experiment 30 times, taking both the arithmetic mean and the standard deviation for all timing measurements.

Application Settings and Inputs: Table 1 shows a subset of the specifications used in Redline. It includes the *init* process, *kjournald*, the X11 server *Xorg*, KDE's desktop/window manager, the *bash* shell, and several typical in-

teractive applications. We left the memory protection period (π) and I/O priority empty in all the specifications, letting Redline choose them automatically.

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. Note that a pessimistic admission test would not accept all of the applications in Table 1 because they would overcommit the system. Redline, however, accepts these and many other of interactive tasks for these experiments.

	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.

8.1 CPU Scheduling

Scheduler Overhead: We compare the performance of the Redline EDF scheduler with the Linux CFS scheduler. Figure 3(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 unblock 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 testi its EDF scheduler. In the latter case, we assigned specification values for C:T as 1:3, 1:30, 1:300 and 1:3000 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 3(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, thus making the variance in running times is larger than the difference between the results.

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 simulataneously 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 4 shows the number of frames rate of achieved by *mplayer* during the test. In Figure 4(a), the fork bomb comprises 50 tasks, while Figure 4(b) shows a 2,000 task fork bomb. In Redline, a fork bomb can be best-effort or interactive. 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 even the fork bomb's parent task is unable to kill all of the children after 30 seconds. In fact, the whole Linux system becomes unresponsive, with simple 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 perceptable 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.

Competing Interactive Tasks: In order to see how interactive tasks may affect each other, we launch *mplayer*, and then we have a user to drag a window in a circle for 20 seconds. Figure 5 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

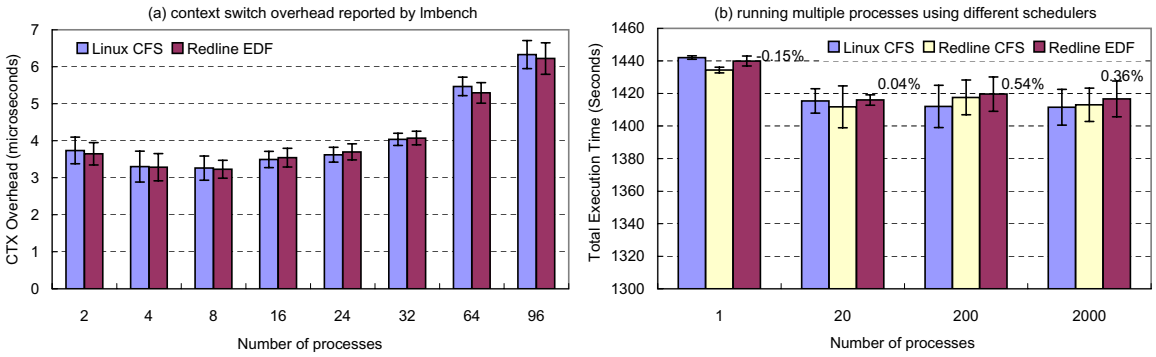

Figure 3: An evaluation of CPU scheduling overhead. Figure (a) shows the context switching times as evaluated by *lmbench*. Figure (b) shows the total running time of varying numbers of CPU intensive tasks.

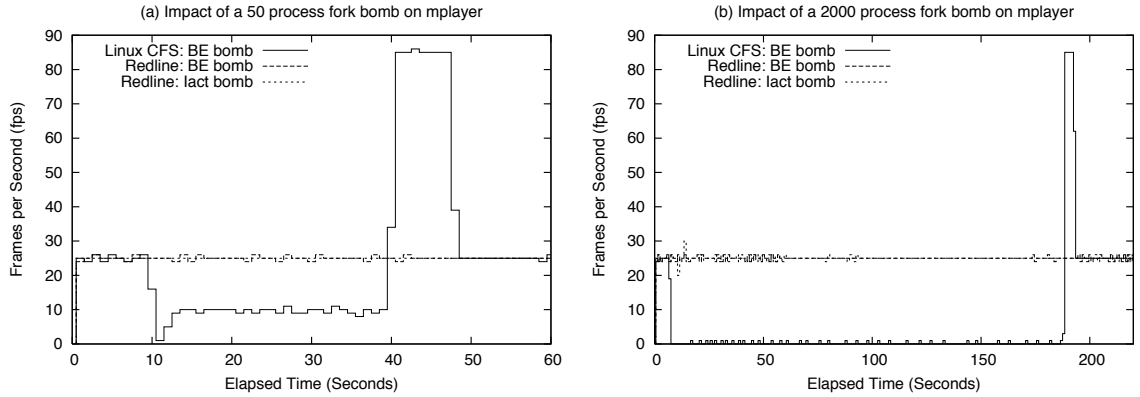

Figure 4: Playing a video and launching fork bombs of (a) 50 or (b) 2,000 (b) tasks.

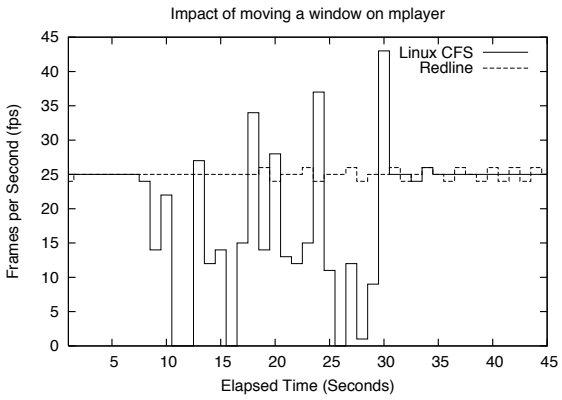

Figure 5: Playing video while dragging around a window.

all other runnable tasks to make enough 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 makes *mplayer* add its requests into *Xorg*'s service queue earlier.

8.2 Memory Management

Memory Bomb: We simulate a workload that has a high memory demand (*memory pressure*) by using a *memory bomb*. This experiment forks four child 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 where the first launches the memory bomb as best-effort tasks, and the second launches them as interactive ones. In the latter case, we use a specification of 10:100 for the memory bomb tasks.

The upper part of Figure 6(a) 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 to pause periodically. The memory bomb forces the VMM to swap out many pages

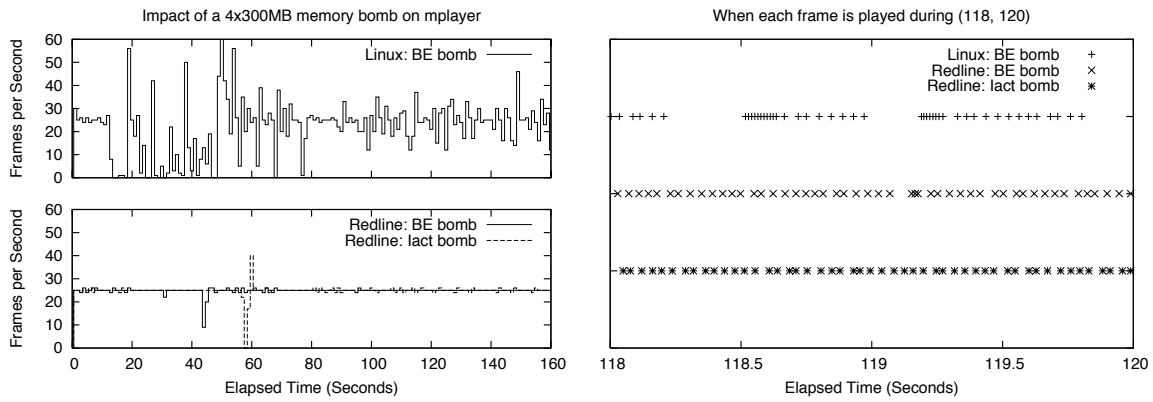

Figure 6: Playing video with 4 x 300 MB memory bomb tasks. The frame rate is severely erratic under Linux, but is steady under Redline.

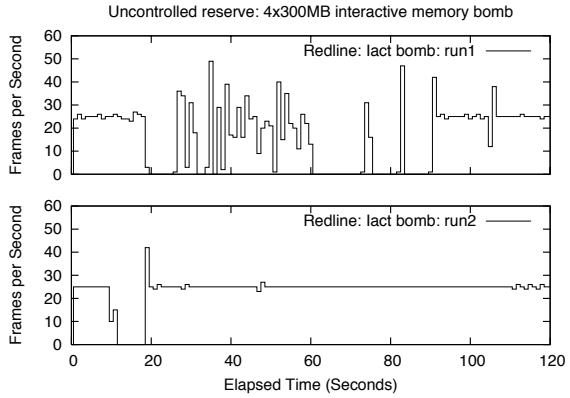

Figure 7: Playing a movie with 4 x 300 MB interactive memory bomb tasks on a Redline system without the rate controlled memory reserve.

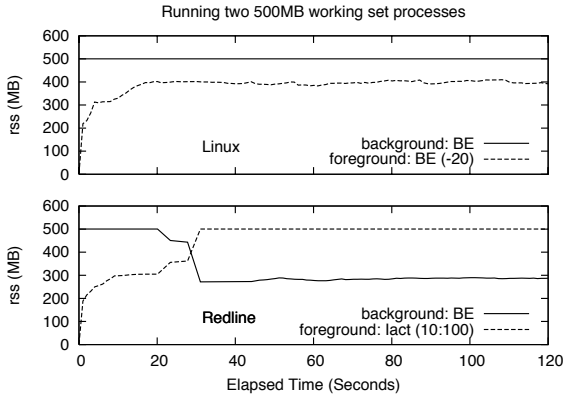

Figure 8: Competing memory bomb tasks. Under Linux, the lower-priority background task prevents the higher-priority foreground task from caching its working set.

used by GUI applications, making the system as a whole unresponsive. Although it appears that the erratic frame rate settles somewhat at around the 80 second mark, Figure 6(b) shows the time at which each frame is played during the period from second 118 to second 120. We see that although the overall frame rate during this period is within a normal range, frames are displayed in bursts, still leaving the user with an undesirable experience.

As shown by the lower part of Figure 6(a), under Redline, *mplayer* successfully survives both the best-effort and interactive memory bomb. Each of them only leads to one brief distruption of the frame rate (less than 3 seconds), which the 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 7 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. Thus, the limited isolation among interactive tasks provided by this small rate controlled reserve is crucial to the system's responsiveness.

Booking: To examine the effectiveness of Redline's page booking 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 8 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. In contrast, under Redline, the second task is quickly allocated space for its full working set, stealing pages from the first, best-effort task.

8.3 Disk I/O Management

Finally, we examine the effectiveness of Redline's disk I/O management by running disk I/O intensive tasks.

Writing: For disk writing experiments, we launch two background tasks meant to interfere with responsiveness. Specifically, each repeatedly writes to an existing, 200 MB file using buffered writes. Additionally, we use *vim* to perform small, sporadic write requests and *iowrite* to perform large write requests. It is the responsiveness of these two applications in the presence of the two background tasks that we measure.

We modified *vim* to report the elapsed time for invoking its write command, and we use it to save a 30 KB file. Additionally, *iowrite* reports the time needed to write 100 MB to a file in buffered write mode. Each of these task is set to the highest priority (-20) under Linux, while each is launched as an interactive task with a specification of 5:100 under Redline.

For Linux, when *vim* writes using BW, each transaction in the journaling file system is heavily loaded with dirty pages from the background tasks. Thus, the call to `fsync()` performed by *vim* causes it to block 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 commited 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). Figure 9 shows the number frame rate of *mplayer* over time for both Linux and Redline. Under Linux, the heavy contention for I/O bandwidth the frame rate to be severely degraded and erratic. Here, *mplayer* blocks frequently for data being transferred from disk. Redline solves this problem by automatically assigning higher I/O priorities to interactive tasks. Under Redline, the frame rate is not at all degraded.

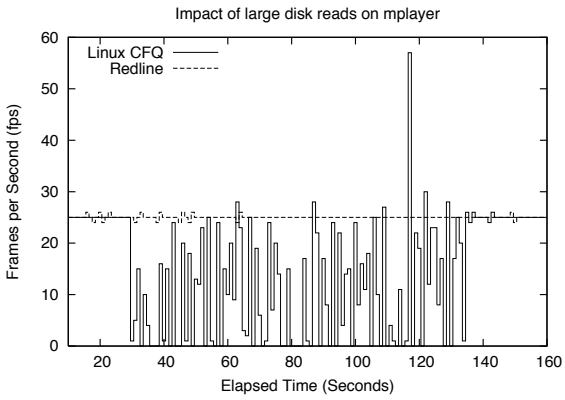

Figure 9: The impact of massive reads on *mplayer*.

9 Related Work

We now briefly discuss the previous efforts in areas that are related with Redline. Table 2 summaries the characteristics of several representive CPU schedulers and operating systems, and compares them with Redline.

CPU Scheduling: Neither time-sharing schedulers (e.g. used by Linux, FreeBSD, Solaris, Windows) or proportional share schedulers [10, 31, 27, 28, 22] that simulate the ideal GPS model [24] have adequate support for addressing response time reqirements. Heuristics are often applied to handle interactive tasks better, which makes the approaches ad-hoc. For instance, Window Vista boosts the priorities of tasks in its newly introduced MM_CLASS into real time region (16–30). Furthermore, the CPU bandwidth received by each task is relative, and thus there is no performance isolation among taks. Hierarchical SFQ [10] partially solves this problem by dividing tasks into classes. A-SFQ [26] dynamically adjusts the weights to retain stable CPU bandwidth for the class serving soft real-time tasks. BVT [8] and BERT [3] achieve the similar goal by adjusting virtual time or deadline. BEST [1] and SMART [21] incorporate EDF algorithm into proportional share schedulers.

Real time systems often use pessimistic admission control and enforcement mechanism to ensure strict performance guarantees. Deng et al. [7] and PSheED [17] provide a uniformly slower processor abstraction using EDF based schedulers. Each real time task is executed as if it is on a slower processor. The schedulers restrict each task to use no more than its assigned bandwidth, and thus are not work conserving. The *CPU reservations* mechanism used by Redline can be implemented in various ways. Nemesis [16] and CPU service class [6] use EDF, Linux/RK [23] uses RM, the scheduler by Lin et al. [11] is table driven, and Rialto [15] assigns CPU time intervals using a tree-based data structure. Intended to provide strict guarantees, these schedulers perform admission test based on task specifica-

		Admission control	Performance isolation		Intg. Mgmt.		Without app. mod.
			interclass	intraclass	mem	I/O	
CPU Scheduler	Stride [31],EEVDF [28], VTRR [22],PD [27]	×	×	×			√
	SFQ [10], A-SFQ [26]	×	strong	×			√
	BVT [8], BERT [3]	×	strong	weak			×
	BEST [1]	×	weak	weak			√
	SMART [21]	×	strong	strong			×
	PSheED [17], Deng et al. [7]	pessimistic	strict	strict			×
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 CPU schedulers and operating sytems to Redline

tions, which seriously limits the number of response time tasks they can simultaneously support. While Redline uses a semi-optimistic admission test combined with a dynamic load control to accommodate as many periodic and aperiodic tasks as possible.

Memory Management: The Windows virtual memory manager adopts a per-process working set model. It uses a kernel thread to take pages away from a process's working set either periodically or in the face of memory pressure. In Zhou et al. [33], virutal 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 the token for a period of time and build up its working set. CRAMM [32] and Bookmark GC [12] use operating system support for garbage collected applications to avoid page swapping. However, None of them offer enough protection for the working sets of interactive applications.

Disk I/O Management: Traditional disk I/O subsystems are designed to maximize the overall throughput, not 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. 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], have more precise control over I/O bandwidth, and could be incorporated with Redline for better guarantees. But ensuring better responsive time involves more than the I/O scheduler.

Integrated Resource Management: Similar to Redline, Nemesis [16], Rialto [15] and Linux-SRT [5] uses *CPU reservations* to provide response time guarantees and have specialized I/O schedulers. But their pessimistic admission tests makes them less capable of handling a large amount of aperiodic tasks with unpredictable workloads. Nemesis allocates a certain amount of physical memory to each task according to its contract and let task manage them (Self-Paging). Rialto simply locks pages for real time tasks. QLinux [29] divides tasks into classes and uses a hierarchical SFQ scheduler to manage CPU and network bandwidth and Cello as its I/O scheduler. None of them has *truly* integrated memory management.

Solaris Container [19] is the combination of system resource control and the boundary seperation provided by *zones*. Each zone is configured to have dedicated amount of resources and act as a completely isolated virtual server. Eclipse [4] and SPU [30] follow the similar approach. They are desgined for providing high level isolation (e.g., users, application groups) by partitioning resources, not for responsiveness.

Aiming for providing real time service to multiple clients, IRS [9] and Resource Container [2] also provide integrated management of multiple resources (memory is not included in IRS). However, they require all activities in the system to be completely self resource aware, which makes them too restrictive for general purpose operating systems. Furthermore, most of these systems require modifications to existing applications while Redline does not.

10 Conclusion

We present Redline, a specification driven system designed to support as many interactive applications as possible by managing resources in an integrated manner. Redline has the ability to allow a large number interactive applications to co-exist in the system, effectively sharing the system resources while still maintain necessary isolation among these interactive applications, as well as against best-effort applications. Redline's integrated resource management of CPU, memory and disk I/O ensures that excessive overloading never pushes the system into catatrophic states and user never loses the control over system. It effectively maintains the responsiveness in the face of various extreme workloads.

The Redline system is open-source software and may be downloaded at `http://www.cs.umass.edu/~tingy/Projects/Redline/Redline.htm`.

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* 11^{th} *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* 3^{rd} *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* 7^{th} *RTAS*, pages 135–140, 2001.

[6] H.-H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proc. of the* 6^{th} *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* 17^{th} *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* 9^{th} *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* 2^{nd} *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* 2^{nd} *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* 18^{th} *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* 7^{th} *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* 21^{st} *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* 15^{th} *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* 5^{th} *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* 7^{th} *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* 8^{th} *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* 8^{th} *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* 7^{th} *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* 11^{th} *ASPLOS*, pages 177–188, Boston, MA, Oct. 2004.