Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Interactive Kernel Performance

Kernel Performance in Desktop and Real-time Applications

Robert Love
MontaVista Software
rml@tech9.net, http://tech9.net/rml

Abstract

The 2.5 development kernel introduced multiple changes intent on improving the interactive performance of Linux. Unfortunately, the term "interactive performance" is rather vague and lacks proper metrics with which to measure. Instead, we can focus on five key elements:

- fairness
- scheduling latency
- interrupt latency
- process scheduler decisions
- I/O scheduler decisions

In short, these attributes help constitute the feel of Linux on the desktop and the performance of Linux in real-time applications. As Linux rapidly gains market share both on the desktop and in embedded solutions, quick system response is growing in importance. This paper will discuss these attributes and their effect on interactive performance.

Then, this paper will look at the responses to these issues introduced in the 2.5 development kernel:

- O(1) scheduler
- Anticipatory/Deadline I/O Scheduler
- Preemptive Kernel
- Improved Core Kernel Algorithms

Along the way, we will look at the current interactive performance of 2.5.

1 Introduction

Interactive performance is important to a wide selection of computing classes: desktop, multimedia, gamer, embedded, and real-time. These application types benefit from quick system response and deterministic or bounded behavior. They are generally characterized as having explicit timing constraints and are often I/O-bound. The range of applications represented in these classes, however, varies greatly—word processors, video players, Quake, cell phone interfaces, and data acquisition systems are all very different applications. But they all demand specific response times (although to varying degrees) to various stimuli (whether its the user at the console or data from a device) and all of these applications find good interactive performance important.

But what *is* interactive performance? How can we say whether a kernel has good interactive performance or not? For real-time applications, it is easy: "do we meet our timing constraints or not?" For multimedia applications, the task is harder but still possible: for example, "does our audio or video skip?" For

desktop users, the job is even harder. How does one express the interactive performance of their text editor or mailer? Worse, how does one *quantify* such performance? All too often, interactive performance is judged by the seat of one's pants. While actually perceiving good interactive performance is an important part of their actually existing good interactive performance, a qualitative experience is not regimented enough for extensive study and risks suffering from the placebo effect.

For the purpose of this paper, we break down the vague term "interactive performance" and define five attributes of a kernel which benefit the aforementioned types of applications: fairness, scheduling latency, interrupt latency, process scheduler decisions, and I/O scheduler decisions.

We then look at four major additions to the 2.5 kernel which improve these qualities: the O(1) scheduler, the deadline and anticipatory I/O schedulers, kernel preemption, and improved kernel algorithms.

2 Interactive Performance

2.1 Fairness

Fairness describes the ability of tasks to all make not only forward progress but to do so relatively evenly. If a given task fails to make any forward progress, we say the task is starved. Starvation is the worst example of a lack of fairness, but any situation in which some tasks make a relatively greater percentage of progress than other tasks lacks fairness.

Fairness is often a hard attribute to justify maintaining because it is often a tradeoff between overall global performance and localized performance. For example, in an effort to provide maximum disk throughput, the 2.4 block I/O scheduler may starve older requests in order to continue processing newer requests at the current disk head position. This minimizes seeks and thus provides maximum overall disk throughput—at the expense of fairness to all requests.

Since the starved task may be interactive or otherwise timing-sensitive, ensuring fairness to all tasks (or at least all tasks of a given importance) is a very important quality of good interactive performance. Improving fairness throughout the kernel is one of the biggest changes made during the 2.5 development kernel.

2.2 Scheduling Latency

Scheduling latency is the delay between a task waking up (becoming runnable) and actually running. Assuming the task is of a sufficiently high priority, this delay should be quite small: an interrupt (or other event) occurs which wakes the task up, the scheduler is invoked to select a new task and selects the newly woken up task, and the task is executed. Poor scheduling latency leads to unmet timing requirements in real-time applications, perceptible lag in application response in desktop applications, and dropped frames or skipped audio in multimedia applications.

Both maximum and average scheduling latency is important, and both need to be minimized for superior interactive performance. Nearly all applications benefit from minimal average scheduling latency, and Linux provides exceptionally good average-case performance. Worst-case performance is a different issue: it is an annoyance to desktop users when, for example, heavy disk I/O or odd VM operations cause their text editor to go catatonic. If the event is relatively rare enough, however, they may overlook it. Both real-time and multimedia applications, however, require a specific bound on worst-case scheduling la-

tencies to ensure functionality.

The preemptive kernel and improved algorithms (reducing lock hold time or reducing the algorithmic upper bound) result in a reduction in both average and worst-case scheduling latency in 2.5.

2.3 Interrupt Latency

Interrupt latency is the delay between a hardware device generating an interrupt and an interrupt handler running and processing the interrupt. High interrupt latency leads to poor system response as the actions of hardware are not readily perceived by the kernel.

Interrupt latency in Linux is basically a function of interrupt off time—the time in which the local interrupt system is disabled. This only occurs inside the kernel and only for short periods of time reflecting critical regions which must execute without risk of an interrupt handler running. Linux has always had comparatively small interrupt latencies—on modern machines, less than 100 microseconds.

Consequently, reducing interrupt latency was not a primary goal of 2.5, although it undoubtedly occurred as lock hold times were reduced and the global kernel lock (`cli()`) was finally removed.

2.4 Process Scheduler Decisions

The behavior and decisions made by the process scheduler (the subsystem of the kernel that divides the resource of CPU time among runnable processes) are important to maintaining good interactive performance. This should go without saying: poor decisions can lead to starvation and poor algorithms can lead to scheduling latency. The process scheduler also enforces static priorities and can issue dynamic priorities based on a rule-set or heuristic.

The process scheduler in 2.5 provides deterministic scheduling latency via an `O(1)` algorithm and a new interactivity estimator which issues priority bonuses for interactive tasks and priority penalties for CPU-hogging tasks.

2.5 I/O Scheduler Decisions

I/O scheduler (the subsystem of the kernel that divides the resource of disk I/O among block I/O requests) decisions strongly affect fairness. The primary goal of any I/O scheduler is to minimize seeks; this is done by merging and sorting requests. This maximizing of global throughput can directly lead to localized fairness issues. Request starvation, particularly of read requests, can lead to long application delays.

Two new I/O schedulers available for 2.5, the deadline and the anticipatory I/O schedulers, prevent request starvation by attempting to dispatch I/O requests before a configurable deadline has elapsed.

3 Process Scheduler

3.1 Introduction

The process scheduler plays an important role in interactive performance. The Linux scheduler offers three different scheduling policies, one for normal tasks and two for real-time tasks. For normal tasks, each task is assigned a priority by the user (the nice value). Each task is assigned a chunk of the processor's time (a timeslice). Tasks with a higher priority run prior to tasks with a lower priority; tasks at the same priority are round-robined amongst themselves. In this manner, the scheduler prefers tasks with a higher priority but ensures fairness to those tasks at the same priority.

The kernel supports two types of real-time

tasks, first-in first-out (FIFO) real-time tasks and round-robin (RR) real-time tasks. Both are assigned a static priority. FIFO tasks run until they voluntarily relinquish the processor. Tasks at a higher priority run prior to tasks at a lower priority; tasks at the same priority are round-robined amongst themselves. RR tasks are assigned a timeslice and run until they exhaust their timeslice. Once all RR tasks of a given priority level exhaust their timeslice, the timeslices are refilled and they continue running. RR tasks at a higher priority run before tasks at a lower priority. Since real-time tasks can be scheduled unfairly, they are expected to have a sane design which properly utilizes the system.

All scheduling in Linux is done preemptively, except FIFO tasks which run until completition. New in 2.5, preemption of tasks can now occur inside the kernel.

For 2.5, the process scheduler was rewritten. The new scheduler, dubbed the O(1) scheduler, features constant-time algorithms, per-processor runqueues, and a new interactivity estimator. The Linux scheduling policy, however, is unchanged.

3.2 Interactivity Estimator

The 2.5 scheduler includes an interactivity estimator [mingo1] which dynamically scales a task's static priority (nice value) based on its interactivity. Interactive tasks receive a priority bonus while tasks which excessively hog the CPU receive a penalty. Tasks in some theoretical neutral position (neither interactive nor hoggish) receive neither a bonus nor a penalty. By default, up to five priority values are added or removed to reflect the degree of the bonus or the penalty; note this corresponds to 25% of the full -20 to 19 nice value range.

Interactivity is estimated using a running sleep average. The idea is that interactive tasks are I/O bound. They spend much of their time waiting for user interaction or some other event to occur. Tasks which spend much of their time sleeping are thus interactive; tasks which spend much of their time running (continually exhausting their timeslice) are CPU hogs. These rules are surprisingly simple; indeed, they are essentially the definitions of I/O-bound and CPU-bound. The fact the heuristic is basically following the definition lends credibility to the estimator.

The heuristic determines the actual bonus or penalty based on the ratio of the task's actual sleep average against a constant "maximum" sleep average. The closer the task is to the maximum, the more of the five bonus priority levels it can receive. Conversely, the closer the task is to the negation of the maximum sleep average, the larger its penalty is.

The results of the interactivity estimator are apparent:

```
USER   NI   PRI   %CPU   STAT   COMMAND
rml    0    15    0.0    S      vim
rml    0    18    0.4    S      bash
rml    0    25    91.7   R      infloop
```

First, note the kernel priority values (the `PRI` column) correspond to a mapping of the default nice value of zero to the value 20. The lowest priority nice value, 19, is priority 39. The highest priority nice value, -20, is zero. Thus, lower priority values are higher in priority. A text editor, vim, has received the full negative five priority bonus. Since it initially had a nice value of zero, it now has a priority of 15. Conversely, a program executing an infinite loop received the full positive five priority penalty; it now has a priority of 25. Bash, which is basically interactive but performs computation in scripts, has received a smaller bonus and now has a priority of 18.

Higher priority (that is, lower priority val-

ued) tasks are scheduled prior to lower priority (higher priority valued) tasks. They also receive a larger timeslice. This implies that interactive tasks are usually runnable; they are scheduled first and generally have plenty of timeslice with which to run. This ensures that the text editor is capable of responding to a keypress instantly, even if the system is under load.

The interactivity estimator does not apply to real-time tasks, which occupy a fixed priority in a higher priority range than any normal task. The estimator benefits interactive desktop programs, such as a text editor or mailer.

3.3 Reinsertion of Interactive Tasks

All process schedulers implement a mechanism of recalculating and refilling process timeslices. In the most rudimentary of schedulers, this work occurs when all processes have exhausted their timeslice and then the timeslice and priority of each process is recalculated and reassigned. Scheduling then continues as before, until again all processes exhaust their timeslice and this work repeats.

The O(1) scheduler implements an `O(1)` algorithm for timeslice recalculation and refilling. Instead of performing a large `O(n)` recalculation when all processes exhaust their timeslice, the O(1) scheduler implements two arrays of tasks, the active array and the expired array. When a task exhausts its timeslice, it is moved to the expired array and its timeslice is refilled. When the active array is empty, the two arrays are switched (via a simple pointer swap) and the scheduler begins executing tasks out of the new active array. This algorithm guarantees a deterministic and constant-time solution to timeslice recalculation.

Another benefit of this approach is it provides a simple prevention to processor starvation of interactive tasks. In the 2.4 scheduler, when a task exhausts its timeslice it does not have a chance to run again until the remaining tasks also exhaust their timeslice and timeslices are globally recalculated. This allows starvation of the task, which might lead to perceptable delays. To prevent this, the 2.5 scheduler will reinsert interactive tasks into the active array when they expire their timeslice. *How* interactive a task need be in order to be reinserted into the active array and not expired depends on the task's priority. To prevent indefinite starvation of non-interactive tasks in the expired array, interactive tasks are only reinserted into the active array so long as the tasks on the expired array have run recently.

3.4 Finegrained timeslice distribution

A final behavioral change in the O(1) scheduler is a more finegrained timeslice distribution and calculation [mingo2]. Currently, this behavior is only present in the 2.5-mm tree but will likely be merged into the mainline 2.5 tree soon.

Normally, tasks are round-robined with other tasks of the same priority (tasks with a higher priority are run earlier and tasks with a lower priority will run later). When they exhaust their timeslice, their priority is recalculated (taking into effect the interactivity estimator) and they are either placed on the expired list or inserted into the back of the queue for their priority level.

This leads to two problems. First, the scheduler may give an interactive task a large timeslice in order to ensure it is always runnable. This is good, but it also results in a long timeslice that may prevent other tasks from running. Second, since priority is recalculated only when a task exhausts its timeslice, a task with a large timeslice may go some time without a priority recalculation. The task's behavior may change

in this time, reversing whether or not the task is deemed interactive. Recognizing this, the scheduler was modified to split timeslices into small pieces—by default, 20ms chunks. Tasks do not receive any less timeslice, instead a task of equal priority may preempt the running task every 20ms. The task is then requeued to the end of the list for its priority and it continues to run round robin with other tasks at its priority level. In addition to this finer distribution of timeslices, the task's priority is recalculated every 20ms as well.

3.5 An `O(1)` Algorithm

An important property of real-time systems is deterministic behavior. Time-sensitive applications demand consistent behavior that they can understand a priori. Critical algorithms, therefore, need to operate in constant time or at least within predefined bounds.

It is important that scheduling behavior (especially process selection and process wake up) operate deterministically, as time-sensitive applications demand minimal latency from wake up to process selection to actual execution. If a scheduling algorithm is dependent on the total number of processes (or runnable processes) even a high priority task cannot make an assumption about scheduling latency. Worse, with a sufficiently large number of processes, the time required to wake up and schedule a task may be far larger than acceptable (e.g., your mp3 may skip or the nuclear power plant may meltdown).

By introducing `O(1)`—constant time—algorithms for all scheduler functions, the O(1) scheduler offers not only deterministic but constant scheduler performance. The scheduler can wake up a task, select it to run, and execute it in the same amount of time regardless of whether there are five or five hundred thousand processes on the system. More so, since the O(1) scheduler has exceptionally quick `O(1)` algorithms, scheduling latency may be reduced for a given `n` over previous scheduling algorithms. Thus, the 2.5 scheduler offers deterministic, constant, and (perhaps) reduced scheduling latency over previous Linux kernel schedulers.

4 I/O Scheduler

4.1 Introduction

The primary job of any I/O scheduler (sometimes called an elevator) is to merge adjacent requests together (to minimize requests) and to sort incoming requests seek-wise along the disk (to minimize disk head movement). Reducing the number of requests and minimizing disk head movement is critical for overall disk throughput. Disk seeks (moving the disk head from one sector to another) are very slow. If the number and distance of seeks are minimized by reordering requests, disk transfer rates are kept closer to their theoretical maximum.

In the interest of global throughput, however, I/O scheduler decisions can introduce local fairness problems. Sorting requests can lead to the starvation of requests that are not near other requests on the disk. If a heavy writeout is underway, the incoming write requests are inserted near each other in the request queue and dealt with quickly, minimizing seeks. A request to a far-off sector may not receive attention for some time. This request starvation is detrimental to system response as it is unfair. Request starvation is a shortcoming in the 2.4 I/O scheduler.

The general issue of request starvation leads to a more specific case of starvation, writes-starving-reads. Write operations can usually occur whenever the I/O scheduler wishes to commit them, asynchronous with respect to the

submitting application or filesystem. Read operations, however, almost always involve a process waiting for the read to complete—that is, read requests are usually synchronous with respect to the submitting application or filesystem. Because system response is largely unaffected by write latency (the time required to commit a write) but is strongly affected by read latency (the time required to commit a read), prioritizing reads over writes will prevent write requests from starving read requests and increase the responsiveness of the system.

Unfortunately, minimizing seeks and preventing unfairness from request starvation are largely conflicting goals. With a proper solution, however, the fairness issues are resolvable without a large drop in global disk throughput.

4.2 Request Starvation

To prevent starvation of requests, a new I/O scheduler, the deadline I/O scheduler, was introduced [axboe1, axboe2]. The deadline I/O scheduler works by assigning tasks an expiration time and trying to ensure (although not guaranteeing) that requests are dispatched before they expire.

The 2.4 I/O scheduler [arcangeli] implements a single queue, which is sorted ascendingly by sector. Requests are either merged with adjacent requests or sorted into the proper location in the queue; requests are appended to the tail if they have no proper insertion point. The I/O scheduler then dispatches requests as the block devices request them from the head of the queue.

The deadline I/O scheduler augments this sorted queue with two more queues, a first-in first-out (FIFO) queue of read requests and a FIFO queue of write requests. Each request in the FIFO queues is assigned an expiration time. By default, this is 500 milliseconds for read requests and 5 seconds for write requests. When a request is submitted to the deadline I/O scheduler, it is added to both the sorted queue and the appropriate FIFO queue. In the case of the sorted queue, the request is merged or otherwise inserted sector-wise where it fits. In the case of the FIFO queues, the request is assigned an expiration value and placed at the tail of the queue.

Normally, the deadline I/O scheduler services requests from the sorted queue, to minimize seeks. If a request expires at the head of either FIFO queue (the requests at the head are the oldest), however, the scheduler stops dispatching items from the sorted queue and begins dispatching from the FIFO queues. This behavior ensures that, in general, seeks are minimized and thus global throughput is maximized. Fairness is maintained, however, as the I/O scheduler attempts to dispatch requests within the specified expiration time. The deadline I/O scheduler provides an upper bound on request latency—ensuring fairness—at the expense of a small degradation in overall throughput.

4.3 Writes-Starving-Reads

Usually, read operations are synchronous while writes operations are asynchronous. Basically, when an application issues a read request, it cannot continue until the operation completes and the application is given the requested value. The completion of write operations, on the other hand, usually has no bearing on the progress of the application. Aside from worrying about power failures, an application is unconcerned as to whether a write commits to disk in one second or five minutes. In fact, most applications are probably unaware if the data is ever committed! Conversely, an application usually needs the results of a read operation and will block until the data is returned. Worse, read requests are often issued en masse and each read is dependent on the previous.

The application or filesystem will not submit read request `N` until read request `N-1` completes.

In the 2.4 I/O scheduler, read and write requests are treated equal. The 2.4 I/O scheduler tries to minimize seeks by sorting requests on insert. If a request is issued that is between (seek-wise) two other requests in the queue, it is inserted there. If there is no suitable place to insert the request (perhaps because no other operations are occurring to the same area of the disk), the request is appended to the end of the queue. Consequently, something like

```
cat * > /dev/null
```

where there is even only a moderate number of files in the current directory results in hundreds of dependent read requests. If a heavy write is underway, each individual read request will be inserted at the tail of the queue. Assuming the queue can hold a maximum of about one second's worth of requests, each individual read request takes a second to reach the head of the queue. That is, the heavy write operation continually keeps the queue full with write operations to some part of the disk. When the read request is submitted, there is no suitable insertion point so it is appended to the tail of the queue. After a second, the read is finally at the head of the queue, and it is dispatched. This repeats for each and every individual read. Since each read is dependent on the next, the requests are issued in serial. Thus the previous `cat` takes hundreds of seconds to complete in 2.4 when the system is also under write pressure.

Recognizing that the asynchrony and interdependency of read operations highlights their much stronger latency requirements over writes, various patches were introduced [akpm1] to solve the problem. Acknowledging that appending reads to the tail of the queue is detrimental to performance, these modifications insert reads (failing a proper insertion elsewhere) near the head of the queue. This drastically improves application performance—more than ten-fold improvements—as it prevents writes from starving reads.

The deadline I/O scheduler, the current default I/O scheduler in 2.5, addresses this issue as well. The deadline I/O scheduler provides a separate (generally much smaller) expiration timeout for read requests. Consequently, the I/O scheduler tries to submit reads requests within a rather short period, ignoring write requests that may be adjacent to the disk head's current location or that have been waiting longer. This prevents the starvation of reads.

Unfortunately, not all is well. While the deadline I/O scheduler solves the read latency problem, the increased attention to read requests results in a seek storm. For each submitted read request, any pending writes are delayed, the disk seeks to the location of the reads and performs the operation, and then it seeks back and continues with the writes. This results in two seeks for each read request (or group of adjacent read requests) that are issued during write operations.

Compounding the problem, reads are issued in groups of dependent requests, as discussed. Not long after seeking back and continue the writes, another read request comes in and the whole mess is repeated.

The goal of a research interest in I/O schedulers, anticipatory I/O scheduling [iyer], is to prevent this seek storm. When an application submits a read request, it is handled within the usual expiration period, as usual. After the request is submitted, however, the I/O scheduler does not immediately return to handling any pending write requests. Instead, it does nothing at all for a few milliseconds (the actual value

is configurable; it defaults to 6ms). In those few milliseconds, there is a good chance the application will submit another read request. If any read request is issued to adjacent areas of the disk, the I/O scheduler immediately handles them. In this case, the I/O scheduler prevented another pair of seeks. It is important to note that the few milliseconds spent waiting is well worth the prevention of the seeks—this is the point of anticipatory I/O scheduling. If a request is not issued in time, however, the I/O scheduler times out and returns to processing any write requests. In that case, the anticipatory I/O scheduler loses and we lost a few milliseconds.

The key is properly anticipating the actions of applications and the filesystem. If the I/O scheduler can predict the actions of an application a sufficiently large enough percentage of the time, it can successfully limit seeks (which are terrible to disk performance) and still provide low read latency and high write throughput. A version of the deadline I/O scheduler, the anticipatory scheduler [piggin], is available in 2.5-mm which supports anticipatory I/O scheduling. The anticipatory I/O scheduler implements per-process statistics to raise the percentage of correct anticipations.

The results are very satisfactory. Under a streaming write, such as

```
while true; do
  dd if=/dev/zero of=file bs=1M
done
```

a simple read of a 200MB file completes in 45 seconds on 2.4.20, 40 seconds on 2.5.68-mm2 with the deadline I/O scheduler, and 4.6 seconds on 2.5 with the anticipatory I/O scheduler. In 2.4, the streaming write results in terrible starvation for the read requests. The anticipatory I/O scheduler results in nearly a ten-fold improvement in read throughput.

In 2.4, the effect of a streaming read upon a series of many small individual reads is also devastating. Perform a streaming read via:

```
while true
do
    cat big-file > /dev/null
done
```

and measure how long a read of every file in the current kernel tree takes:

```
    find . -type f -exec \
        cat '{}' ';' > /dev/null
```

2.4.20 required 30 minutes and 28 seconds, 2.5.68-mm2 with the deadline I/O scheduler required 3 minutes and 30 seconds, and 2.5.68-mm2 with the anticipatory I/O scheduler required a mere 15 seconds. That is a 121-times improvement from 2.4 to 2.5.68-mm2 with the anticipatory I/O scheduler.

How much damage does this benefit to read latency do to global throughput, though? It is clear that read throughput is improved, but at what cost to write requests and global throughput? Consider the inverse, under a streaming read such as:

```
while true
do
    cat file > /dev/null
done
```

A simple write and sync of a 200MB file takes 7.5 seconds on 2.4.20, 8.9 seconds on 2.5.68-mm2 with the deadline I/O scheduler, and 13.1 seconds on 2.5.68-mm2 with the anticipatory I/O scheduler. The 2.5 I/O schedulers are slower, but not overly so (certainly not to the degree read latency is decreased). This test does not show global throughput, though,

just write throughput in the presence of heavy reads. Since the streaming read above may operate much quicker, global throughput is often largely unchanged.

The anticipatory I/O scheduler is currently in the 2.5-mm tree. It is expected that it will be merged into the mainline 2.5 tree before 2.6.

5 Preemptive Kernel

5.1 Introduction

The addition of kernel preemption in 2.5 provides significantly lowered average scheduling latency and a modest reduction in worst-case scheduling latency. More importantly, introducing a preemptive kernel installs the initial framework for further lowering scheduling latency by allowing developers to tackle specific locks as the root of scheduling latency as opposed to entire kernel call chains. Conveniently, reducing lock hold time is also a goal for large SMP machines

5.2 Design of a Preemptive Kernel

Evolving an existing non-preemptive kernel into a preemptive kernel is nontrivial; the task is greatly simplified, however, if the kernel is already safe against reentrancy and concurrecny. Therefore, in the case of the Linux kernel, the safety provided by existing SMP locking primitives were leveraged to provide a similar protection from kernel preemptions. SMP spin locks were modified to disable kernel preemption in a nested fashion; after `n` spin locks, kernel preemption is not again enabled until the `n`-th unlock.

The `ret_from_intr` path (the architecture-dependent assembly which returns control from the interrupt handler to the interrupted code) was then modified to allow preemption even if returning to kernel mode. Thus, a task woken up in an interrupt handler (a common occurrence) can then run at the earliest possible moment, as soon as the interrupt handler returns. Consequently, a high priority task will preempt a lower priority task, even if the lower priority task is executing inside the kernel.

The preemption does not occur on return from interrupt, of course, if the interrupted task holds a lock. In that case, the pending preemption will occur as soon as all locks are released—again at the earliest possible moment.

5.3 Improved Core Kernel Algorithms

Changes to core kernel algorithms (primarily in the VM and VFS primarily) were made to improve fairness, provide a better bound on time complexity (and thus a bound on scheduling latency), and reduce lock hold time to take advantage of kernel preemption and reduce latency.

Some of the most important changes were to fix fairness issues in the VM, in code paths such as the page allocator. These changes prevent VM pressure caused by one process from unfairly affecting VM performance of other processes.

Many small changes were made to kernel functions in known high latency code paths in the kernel. These changes involved modifying the algorithm to have a minimized or fixed bound on time complexity and to reduce lock hold so as to allow kernel preemption sooner.

5.4 Reducing Scheduling Latency

Measurements of scheduling latency are highly dependent on both machine and workload (workload being a crucial element—one workload may show no perceptible scheduling la-

tency while another may introduce horrid scheduling latencies). Nonetheless, worst-case scheduling latencies of under 500 microseconds are commonly observed in 2.5.

Even on a 2.4 kernel patched with the preemptive kernel (which undoubtedly does not benefit from some of the algorithmic improvements in 2.5), a recent whitepaper [williams] noted a five-fold improvement in worst-case scheduling latency and a 1.6-time improvement in average case scheduling latency. A long-term test, part of the same whitepaper, testing the kernel for over 12 hours (to exercise many high scheduling latency paths) showed a reduction from over 200 milliseconds worst-case latency in the period to 1.5 milliseconds with a combination of the preemptive kernel and the low-latency patch [akpm2]. This drastic reduction in worst-case latency over a long period with a complex workload demonstrates the ability of the preemptive kernel and optimal algorithms to provide both excellent average and worst-case scheduling latency.

One useful benchmark is the Audio Latency Benchmark [sbenno], which simulates keeping an audio buffer full under various loads. A test of a 2.4 kernel vs. a 2.4 preemptive kernel shows a reduction in worst-case scheduling latency from 17.6 milliseconds to 1.5 milliseconds [rml]. The same test on 2.5.68 yields a maximum scheduling latency of 0.4 milliseconds.

On a modern machine, scheduling latency is low enough to prevent any perceptible stalls during typical desktop computing and multimedia work.

Further, the 2.5 kernel provides a base sufficient for guaranteeing sub one millisecond worst-case latency for demanded embedded and real-time computing needs.

6 Acknowledgments

I would like to thank the OLS program committee for providing the opportunity to write this paper and MontaVista Software for providing the means by which I work on the kernel.

Andrew Morton deserves credit for an abnormally large amount of the interactivity work which went into the 2.5 kernel. Jens Axboe was the primary developer of the deadline scheduler. Nick Piggin was the primary developer of the anticipatory scheduler, which is based on the deadline scheduler. Ingo Molnar was the primary developer of the O(1) scheduler. Various others played significant roles in the design and implementation of other related kernel bits.

References

[akpm1] Andrew Morton, *Patch: read-latency2*, `http://www.zip.com.au/~akpm/linux/patches/2.4/2.4.19-pre5/read-latency2.patch`.

[akpm2] Andrew Morton, *Patch: low-latency*, `http://www.zipworld.com.au/~akpm/linux/schedlat.html`.

[arcangeli] Andrea Arcangeli and Jens Axboe, *Source: 2.4 Elevator* , `linux/drivers/block/elevator.c`.

[axboe1] Jens Axboe, *Email: [PATCH] deadline io scheduler*, `http://www.cs.helsinki.fi/linux/linux-kernel/2002-38/0912.html`.

[axboe2] Jens Axboe, *Source: Deadline I/O Scheduler*, `linux/drivers/block/deadline-iosched.c`.

[iyer] S. Iyer and P. Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*. ACM Symposium on Operating System Principals (SOSP), 2001.

[mingo1] Ingo Molnar, *Source: O(1) scheduler*, `linux/kernel/drivers/sched.c`.

[mingo2] Ingo Molnar, *Patch: sched-2.5.64-D3*, `http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.68/2.5.68-mm2/broken-out/sched-2.5.64-D3.patch`.

[piggin] Nick Piggin and Jens Axboe, *Source: Anticipatory I/O Scheduler*, `linux/drivers/block/as-iosched.c`.

[rml] Robert Love, *Lowering Latency in Linux: Introducing a Preemptible Kernel*, Linux Journal (June 2002).

[sbenno] Benno Senoner, *Audio Latency Benchmark*, `http://www.gardena.net/benno/linux/audio/`.

[williams] Clark Williams, *Linux Scheduler Latency*, Whitepaper, Red Hat, Inc., 2002.