BFS vs. CFS – Scheduler Comparison

Taylor Groves, Jeff Knockel, Eric Schulte

11 December 2009

Abstract

In 2007 a dispute erupted on the Linux kernel mailing list. The dispute was between Con Kolivas, creater of the Staricase Deadline (SD) scheduler and Ingo Molnar, whose newly released Completely Fair Scheduler (CFS) was to be added to the mainline. CFS employed concepts previously envisioned by Kolivas and the dispute ended with Kolivas leaving the mainline kernel development community. Two years later Kolivas released the Brain Fuck Scheduler (BFS) with a focus on a simplistic design, tailored towards interactive workloads on commodity personal computers with a reasonable number of cores. We review the mechanisms of the CFS and BFS schedulers and test both on a variety of single-user hardware over a variety of workloads with a strong focus on the range of "normal" PC use. Our results indicate that scheduler performance varies dramatically according to hardware and workload, and as a result we strongly encourage Linux distributions to take an increased level of responsibility for selecting appropriate default schedulers that best suit the intended usage of the system.

Introduction

Given the wide range of hardware running the Linux kernel it is difficult to provide a single CPU scheduler that performs well in all platforms and use cases. Many metrics of scheduler performance such as latency and turnaround time can be at odds – improving one metric will often reduce performance on another. We investigate the performance of two CPU schedulers – the "Completely Fair Scheduler" (CFS) and the "Brain Fuck Scheduler" (BFS) – available for the modern Linux kernel. Our tests are primarily aimed at the personal computers environment. We find that often the best scheduler depends on the system and expected use cases and as a result we conclude that it is appropriate to select a CPU scheduler with these factors in mind.

Background

Scheduler Overview

CPU/IO Burst Cycle

Processes are constantly alternating between CPU burst, doing low level operations like loads, stores, adds, etc. and I/O burst in which the system is waiting. In general the CPU burst duration is exponential with a large number of short burst and a increasingly small number of large bursts. The type of program that is being run can change this curve substantially, so it is important to tailor the cpu scheduler for the tasks at hand, predicting whether it will be a CPU-bound or I/O bound program.

Filling the IO Gap

When a process makes a request for I/O such as a system call like read(), if the device driver that handles the request is busy then process is put on a wait queue. The driver is responsible for generating an interrupt, which wakes up this thread once the request for I/O has been fulfilled. Once a process has been determined to be idle, its the schedulers job to select a process from the group of ready processes and allocate the CPU to that process.

After an I/O request has been fulfilled an interrupt is triggered and if the thread can run again, a need_resched flag is set for the scheduling data structure. When a process is restarted the dispatcher handles the context switch change in user mode, and ensures that the program restarts from the proper location.

Some sort of data structure manages the order that the scheduler pulls a process from memory for execution. This can be anything from a FIFO queue to a red-black-tree, but all schedulers use this data structure with some unit of measurement such as CPU burst duration, order of requests, or priority.

What's important when comparing schedulers

It is important CPU schedulers to maximize

CPU utilization How saturated the CPU is from 0 to 100%.

Throughput The number of processes completed per unit of time.

It is important for CPU schedulers to minimize

Turnaround time The total time taken for a process to complete once started.

Waiting time The total time a process spends in the ready queue.

Scheduler latency The time between a wakeup signal being sent to a thread on the wait queue and the scheduler executing that thread. Including the dispatch latency – the time taken to handle a context switch, change in user-mode, and jump to starting location.

- Reducing Scheduler Latency

  Running the scheduler more frequently will reduce scheduler latency since a thread will spend less time waiting in the ready state. However, there is a trade-off since running the scheduler more frequently will result in more time spent scheduling and less time actually performing the task at hand.

  Ideally the scheduler will run as soon as possible, after a interrupt has occurred which wakes up a thread.

Different Strokes for Different Folks

Most of the criteria for determining a good scheduling algorithm depends on the type of work being done on the machine. In general this work can be classified into three categories: Batch, Interactive, and Real Time.

- Batch: this category is less concerned with timing and latency but rather seeks to maximize the amount of work accomplished in a given time. Batch workloads are characterized by lack of user interaction.
- Interactive: being able to service the user quickly is paramount. Schedulers that focus on interactivity respond to input like key strokes consistently with a low variance and low average time to wake up a sleeping process.
- Real-Time: Systems focused on real-time processes such as live video encoding with multimedia playback, or sensor networks and will want to reduce latency and variance and strictly enforced priorities. The requirements of the scheduler for systems that are focused on real-time processes are the most stringent of the three categories.

Con Kolivas and LKML

Con Kolivas – the author of the BFS – is an anesthesiologist by trade. A timeline of his involvement in the Linux kernel development is provided 1. His relationship with the Linux kernel developers was often times fractious, and there is an extensive online record of their public arguments [1] [2].

Scheduler Implementations

Multilevel Feedback Queues And The O(1) Scheduler

Many commodity operating system schedulers use multilevel feedback queues. These include Linux prior to 2.6.23, as well as Windows NT-based operating systems such as Windows XP, Vista, and 7 [3]. These schedulers use separate queues to give preference to interactive and I/O bound tasks. These tasks are prioritized to increase interactivity and I/O utilization.

With multilevel feedback queues, new tasks are initially queued into a middle priority queue. If a task uses its entire timeslice, then it is moved down to a lower priority queue. If a task blocks before its time timeslice is up, then it is placed in a higher priority queue. Depending on the implementation, to reward I/O bound tasks, tasks on higher priority queues can have longer timeslices and be scheduled before tasks on lower priority queues.

Consider Linux prior to 2.6.23, which uses the O(1) scheduler – so named because selecting the next task to run takes at most a constant amount of time. The O(1) scheduler implements a multilevel feedback queue similar to the above. With this scheduler, tasks start with an initial static priority, or niceness, in the range of [-20,20) where a lower niceness is a higher priority. The tasks priority is adjusted depending upon its perceived interactivity. Higher priority tasks a given larger timeslices and preferential scheduling [4].

The O(1) scheduler guarantees constant runtime by maintaining two runqueues per CPU, the active runqueue and the expired runqueue. These queues are implemented using a FIFO list for each possible priority. After a task uses its timeslice its priority is potentially modified, and it is placed on the expired runqueue. After all tasks on the active runqueue have been given time on the CPU the expired runqueue is swapped with the active runqueue. Since all of these operations are O(1), the O(1) scheduler is justified in its naming

[1] http://article.gmane.org/gmane.linux.kernel.ck/7850
[2] http://marc.info/?l=linux-kernel&m=125229579622556&w=2
[3] http://www.ibm.com/developerworks/linux/library/l-scheduler/
[4] http://www.informit.com/articles/printerfriendly.aspx?p=101760&rll=1

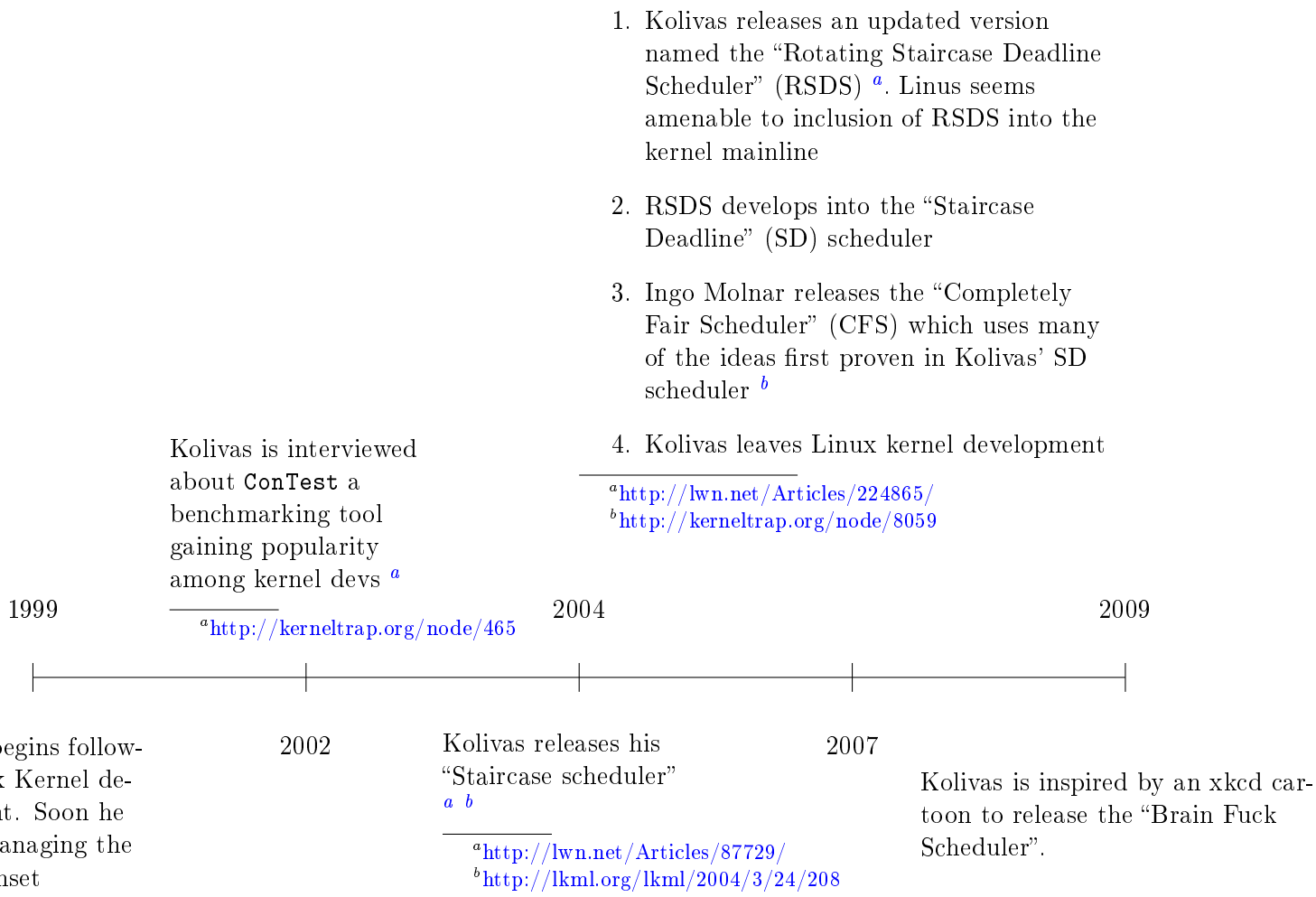


Figure 1: Timeline

Since the O(1) scheduler maintains a runqueue pair per CPU, separate CPUs rarely try to access the same runqueue, and tasks tend to stay on the same processor. Because of these properties, the O(1) scheduler scales well to highly parallel systems; however, the scheduler must occasionally perform explicit load balancing to ensure tasks are evenly distributed over all CPUs [5].

One problem with schedulers implemented using multilevel feedback queues is differentiation of CPU bound tasks from I/O bound tasks. Consider a scheduler that always rewards a task that blocks before its timeslice is up. Then a malicious user could write a task that blocks moments before its time slice has expired in order to be effectively CPU bound but still acquire the privileges of an I/O bound task. To help ameliorate this problem, the O(1) scheduler uses complicated heuristics to detect when a task is I/O bound. However, in practice, this adds more exploitable corner cases and the problem quickly devolves to a game of cat and mouse.

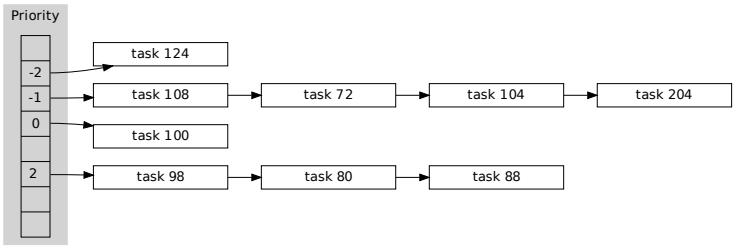


Figure 2: O(1) Scheduler data structure

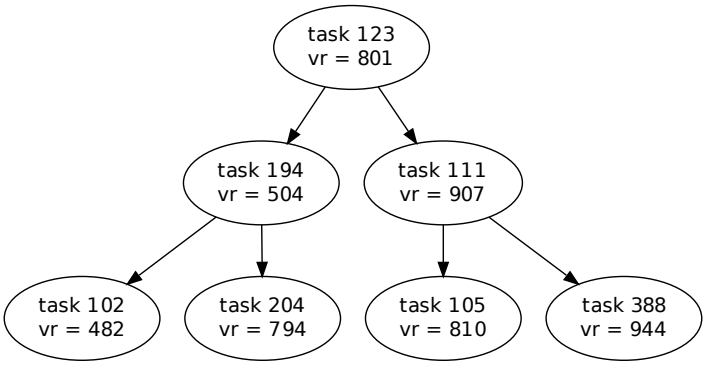


Figure 3: CFS scheduler data structure

Fair Scheduling And The Completely Fair Scheduler

The Completely Fair Scheduler (CFS) was written by Ingo Molnar. It was officially included in Linux 2.6.23, and it seeks to address a number of problems with schedulers, including issues of fairness and malicious users.

CFS is an implementation of fair-share scheduling policy. A fair-share scheduling policy divides the CPU time among entities, such as users, groups, or tasks themselves. Although CFS supports fair-share scheduling on the user and group levels, its name originally refers to fair-share scheduling on the task level.

Ingo Molnar explains this by using the thought experiment of an ideal, multi-tasking CPU [6]. This CPU runs each task in parallel at an equal fraction of the CPU's speed. For instance, if four tasks are running on it, then each task runs at 25% speed. On this hypothetical CPU, there is no state in which one task has gotten more of the CPU than another, since all tasks are running on it at once, so all tasks have a fair share of the CPU.

CFS models this hypothetical processor by keeping track of how unfairly each task has been

[5] http://www.ibm.com/developerworks/linux/library/l-scheduler/
[6] http://lwn.net/Articles/357451/

treated relative to the others. On the hypothetical ideal CPU, unfairness would always be zero but since a real CPU can only schedule a finite number of tasks at a time, unfairness will inevitably be nonzero when there are more tasks than CPUs. When one task is running on a CPU, this increases the amount of CPU time that the CPU owes to all other task. CFS schedules the task with the largest unfairness onto the CPU first.

CFS manages tasks using a red-black tree. Inside of the tree, tasks are in descending order by their total unfairness, i.e., by the time of their future execution – since the task presently being treated the most unfairly will be scheduled next. The scheduler selects the next task by following left nodes to choose the leftmost node in the red-black tree. Because it uses a red-black tree CFS, requires O(logn) time to schedule a task where n is the number of tasks.

When a task has finished running on the CPU, all of the other tasks in the tree need to have their unfairness increase. To prevent having to update all of the tasks in the tree the scheduler maintains a per-task vruntime statistic. This is the amount of total nanoseconds that the task has spent running on a CPU weighted by its niceness. Thus, instead of updating all other tasks to be more unfair when a task has finished running on the CPU, we update the leaving task to be more fair than others by increasing its virtual runtime. The scheduler always selects the most unfairly treated task by selecting the task with the lowest vruntime.

When a new task is created, it is assigned the minimum current vruntime. To facilitate this, the minimum vruntime statistic must also be maintained. This minimum vruntime value is also used to handle overflow of the vruntime value so that vruntime - min_vruntime can be reliably used to sort the red-black tree.

With CFS, tasks have no concept of timeslice but rather run until they are no longer the most unfairly treated task. To reduce context switching overhead. CFS divides time into a minimum granularity. The granularity value can be used to control the tradeoff between overhead due to context switching and latency [7].

When a task is awakened from blocking, its new vruntime is the larger of its old vruntime and min_vruntime - sched_latancy, where sched_latency is a time constant. Thus, if a task has been blocking long enough, then its vruntime will be sched_latency smaller than the currently scheduled task, assuring that the currently running task will be preempted and the awakened task will be scheduled. Using min_vruntime - sched_latency as a lower bound on an awakening task's vruntime prevents a task that blocked for a long time from monopolizing the CPU [7].

CFS also supports fair-share scheduling on the user and group levels, but these are not enabled by default. When enabled, this prevents a user or group from unfairly monopolizing the processor by greedily creating more threads.

Like the O(1) scheduler, CFS maintains separate data structures for each CPU. This reduces lock contention but requires explicit load balancing to evenly spread tasks across processors.

A modular scheduler framework was introduced into the kernel along with CFS. A new scheduler can be implemented in the Linux kernel by the following procedure:

- Create your own `sched_foo.c` file in kernel/ and implement sched_class with functions for the kernel's scheduler to call.
- Associate your scheduler with a '#define SCHED_FOO N' constant in include/linux/sched.h.

To pair your scheduler with a running task, you may then call the `sched_setscheduler()` system call with the process id and your scheduler's constant.

[7] http://lkml.org/lkml/2008/7/10/70

Brain Fuck Scheduler – An Alternative

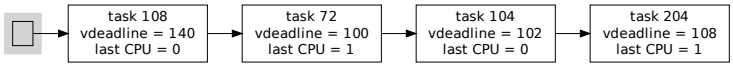


Figure 4: BFS data structure

The Brain Fuck Scheduler (BFS) was written by Con Kolivas as an alternative to the CFS scheduler. Although it is not in the mainline Linux kernel, Kolivas maintains BFS patches against the latest version of the kernel. BFS does not use and removes the modular scheduler framework introduced by the CFS patches.

BFS takes a different approach than both the O(1) scheduler and CFS. BFS uses runqueues like O(1); however, unlike O(1), which has both an active and an expired runqueue per CPU, BFS has only one system-wide runqueue containing all non-running tasks.

Schedulers with multilevel feedback queues generally have to use complex heuristics for determining whether a task is I/O bound. Moreover, schedulers that maintain different data structures per CPU have to use complex algorithms for load balancing across CPU's. BFS removes the need for these complicated heuristics and algorithms by using a single system-wide queue to determine the next scheduled task

BFS implements an earliest effective virtual deadline first policy and keeps track of the virtual deadline of each task. A virtual deadline is the longest time that any two tasks with the same niceness will have to wait before running on the CPU. When a task requests CPU time it is given a timeslice length and a virtual deadline. The deadline is virtual because there is no actual guarantee that a task will be scheduled by that time. However, tasks with earlier virtual deadlines will always be scheduled before tasks with later virtual deadlines. Tasks with higher priority are given earlier virtual deadlines

When a task runs out of its timeslice, it is rescheduled according to the algorithm above; however, when a task blocks, it keeps the remainder of its timeslice and its virtual deadline. This grants the task higher priority when the task is rescheduled, increasing the interactivity of the system [8].

To help improve cache performance, BFS weights the virtual deadline of a task as seen by a CPU if that CPU does not share a cache with the CPU on which the task was previously running. This creates an incentive for a task to stay with its cache [8].

Task lookup requires an O(n) scan over the entire queue. Since BFS maintains only a single queue for all processors and since virtual deadline is CPU-relative, there is no absolute ordering of tasks by their virtual deadline. Thus, tasks cannot be placed in a tree. As a result, BFS scales poorly with large amounts of tasks. Moreover, a shared data structure among all CPU's increases lock contention. However, as a benefit to sharing a data structure among CPU's and because of a lack of any load balancing algorithms, tasks can be quickly scheduled on a different CPU that has become idle, often providing lower latency.

[8].

Experiment

Methodology

Kernel Patching

We installed two kernels: one for testing CFS and one for testing BFS. To install the kernels, we downloaded the 2.6.31.6 source code from http://www.kernel.org and made two copies. For one, we left the source code unmodified, and for the other we patched it with version 311 of Kolivas's BFS patch. For configuration of both kernels we used Ubuntu 9.04's configuration file /boot/config-2.6.28-16-generic and then used the default values for the new configuration options instroduced since Ubuntu 9.04's version. For compilation and installation of the kernels we used the debian tool make-kpkg. The make-kpkg tool compiles the kernel source and outputs debian packages. We installed these debian packages and thus were able to choose from either our CFS kernel or our BFS kernel from the grub boot menu when the machine starts up.

Test Execution

- `latt.c`

  Latt is a tool for benchmarking schedulers. It was written by kernel hacker Jens Axboe with input from Ingo Molnar and Con Kolivas. It benchmarks latency and turnaround time under various workloads. We used the version from the latest commit on 2009-09-10 from the tool's git repository.

  Latt benchmarks latency and turnaround time by gzipping random data in n background processes. Each background process begins by blocking on a read() from a pipe created by latt. To measure latency, latt writes to each background process through the pipe the time at which the write() took place. Each background process then records the time that their read from the pipe unblocks. Latency then is the time read() unblocked minus the time latt wrote to it. To measure turnaround time, each background process then gzips a large array of random data. Turnaround time then is the time the gzipping finished minus the time at which latt originally called write() on the pipe. The background process then writes its statistics back to latt through the pipe and returns to blocking on trying to read() from the pipe. Latt will allow the process to sleep for a short, random amount of time, and then it will restart the test by writing to the pipe again.

  Latt also requires that a shell command be specified for latt to run in addition to running the n background processes. To have latt just test its n background processes for 20 seconds, one can pass it 'sleep 20s' to have the shell command be a trivial process that sleeps for 20 seconds. Otherwise, other commands can be run simultaneously with latt's background processes to test their performance under heavy load.

- Make - Turnaround time focused tests

  Our tests compiling VLC from source is designed to test the turnaround time of a scheduler. The VLC source is composed of 691 files of C source code, containing more than 416 thousand

lines of code. This test of make is aimed at systems which require minimal interactivity with a high volume of work. For our tests we use the command:

```
make -j<number_of_jobs>
```

The -j option launches the specified number of processes, for independent rules in the make file, so that the compiling can occur in parallel. For our tests we used a j ranging from one to four.

- Video - Latency focused tests

In order to test some of the requirements of real-time systems, one series of our tests looks at dropped frames during multimedia playback with increasing client workloads. A single video was chosen at two scales for the tests: 853X480 and 1280X720. The video was the Big Buck Bunny trailer [9], which attempts playback at 25 frames per second and 48KHz audio, compressed with H.264/MPEG-4 AAC. The clips' duration is 32 seconds.

To select a video application to use for this test's playback, we examined two players initially: Videolan's VLC and Mplayer. We chose to use Mplayer for our final benchmarks. As we examined VLC, it was discovered that the player lacked some of the features we desired, such as command-line output of lost frame and audio buffer statistics. This was remedied by patching vlc, so that it printed out these statistics as they were output to the GUI. Unfortunately this was not frequent enough for our demands. Further investigations looked at printing out lost frames as they came in late in the decoding process, but this had the effect of causing additional frame loss. Fortunately, Mplayer has a built-in benchmarking mode, which when set, outputs dropped frame and audio buffer statistics among other things. This was coupled with a flag, -hardframedrop, so that the player would not stall playback and buffer late frames. Looking at a comparison between the two players, neither seemed to show a particular bias towards a scheduler that was not reflected by the other. The only notable difference was that VLC appeared to have a slightly lower frame loss when running on either BFS and CFS scheduler. In the test cases used for benchmarking, we used the following command with 1 to 15 clients adding additional workload to the cpu; see the section on latt for more details:

```
latt -c<number_of_clients> \
    "mplayer -benchmark -hardframedrop <video_to_playback>"
```

In the case of the desktop, "fire", 25 clients were required to see a significant amount of dropped frames and contention for time on the CPU.

Data Analysis

Script based test execution resulted in the generation of uniform results across all experimentation platforms. As such it was possible to computationally collect and collate the experimental results. The following snippet of Ruby code 5 was used to collect all of our experimental results into a single spreadsheet environment amenable to visualization and analysis.

Spreadsheet tools were used to search for meaningful trends in the experimental results and gnuplot was used for visualization.

[9] http://kerneltrap.org/node/465

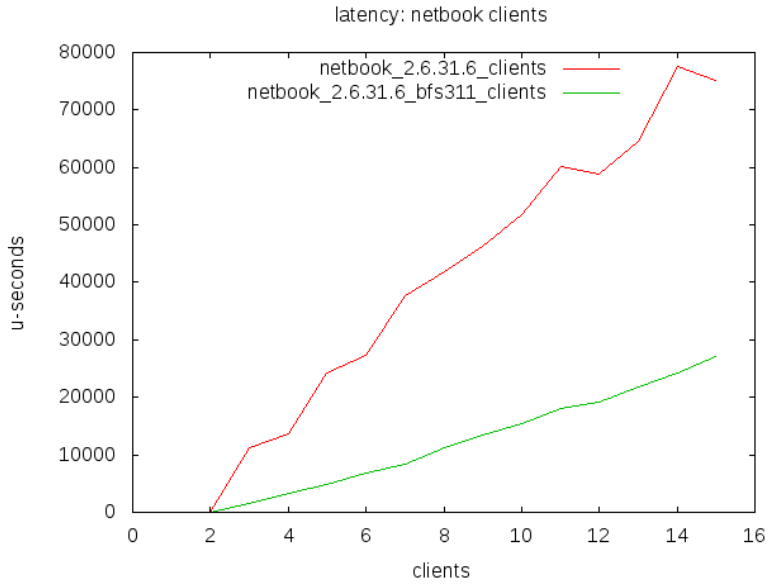
```
["netbook", "laptop", "desktop"].each do |machine|
  ["2.6.31.6", "2.6.31.6-bfs311"].each do |scheduler|
    ["clients", "make", "mplayer", "mplayer_sched"].each do |test|
      base = File.expand_path(File.join(machine, scheduler, test))
      # print results in tabular format
      puts "| #{base} |"
      results_for(base).each{ |l| puts "|"+l.join(" | ")+"|" }
    end
  end
end
```

Figure 5: data collection in Ruby

Results

Latency

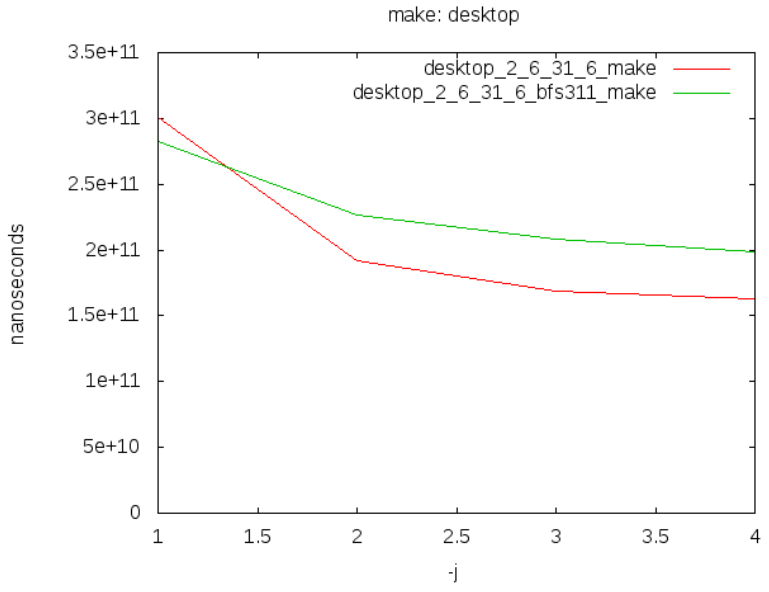
Across all three testing platforms and under all load conditions the BFS scheduler demonstrated lower latency. This is not surprising given that the BFS scheduler targets a smother experience for graphical applications which require low latency. The following graph shows the latency in nanoseconds as reported by the latt.c test script running on a netbook. These results are indicative of the results across all three machines.



Turnaround Time

The CFS scheduler had better performance in terms of turnaround time on all three machines and under all load conditions. The difference in performance was more dramatic on more powerful machines like the multi-core desktop for which results are shown.

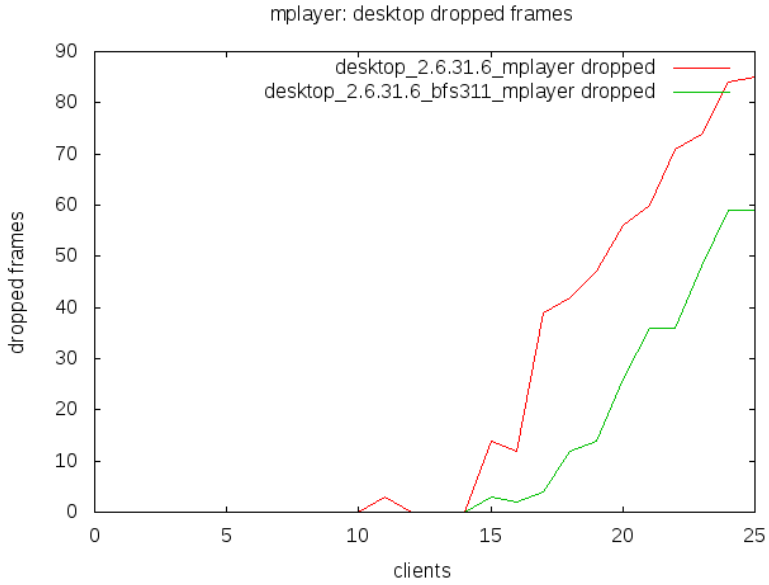
To further demonstrate the superiority of CFS on batch processes we compared make results run on a desktop using various numbers of jobs. These results contradict the claims of Con Kolvias, the creator of BFS. Kolivas claims that the BFS scheduler performs best when the `-j` level of make is equal to the number of CPUs. His claim goes against common wisdom – that `make` best utilizes CPU when `-j` is greater than the number of CPUs available – in effect that it is worthwhile to oversubscribe the number of CPUs. Our results reinforce the common wisdom and show that BFS **does** run faster when oversubscribed, albeit to a lesser degree than CFS.



Interactivity

To test the relative interactive performance of BFS and CFS, we ran mplayer over each scheduler under increasing amounts of background load. The following graphs shows the number of dropped

frames as reported by mplayer. As this graph demonstrates, the BFS scheduler drops significantly less frames under large amounts of load.



Conclusion

The results indicate that CFS outperformed BFS with minimizing turnaround time but that BFS outperformed CFS for minimizing latency. This indicates that BFS is better for interactive tasks that block on I/O or user input and that CFS is better for batch processing that is CPU bound.

Many distros like Ubuntu already have separate kernel packages for desktops and servers optimized for those common use cases. To improve the average desktop experience, distros could patch their kernel to use the BFS scheduler. If desktop users do perform a lot of batch processing, distros could provide two different kernel packages alternatives.