

[RFC][PATCH] O(1) Entitlement Based Scheduler

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-02/6548.html>

From: John Lee (johnl_at_aurema.com)

Date: 02/25/04

Date: Thu, 26 Feb 2004 01:35:05 +1100 (EST)

To: linux-kernel@vger.kernel.org

Hi everyone,

This patch is a modification of the O(1) scheduler that introduces O(1) entitlement based scheduling for SCHED_NORMAL tasks. This patch is aimed at keeping the scalability and efficiency of the current scheduler, but also provide:

- Specific allocation of CPU resources amongst tasks
- Scheduling fairness and good interactive response without the need for heuristics, and
- Reduced scheduler complexity.

The fundamental concept of entitlement based sharing is that each task has an `_entitlement_` to CPU resources that is determined by the number of `_shares_` that it holds, and the scheduler allocates CPU to tasks so that the `_rate_` at which they receive CPU time is consistent with their entitlement.

The usage rates for each task are estimated using Kalman filter techniques, the estimates being similar to those obtained by taking a running average over twice the filter `_response half life_` (see below). However, Kalman filter values are cheaper to compute and don't require the maintenance of historical usage data.

The use of CPU usage rates also makes it possible to impose `_per task CPU usage rate caps_`. This patch provides both soft and hard CPU usage rate caps per task. The difference between a hard and soft cap is that when unused CPU cycles are available, a hard cap is `_always_` enforced regardless, whereas a soft cap is allowed to be exceeded.

Features of the EBS scheduler

=====

CPU shares

Each task has a number of CPU shares that determine its entitlement. Shares can be read/set directly via the files

/proc/<pid>/cpu_shares
/proc/<tgid>/task/<pid>/cpu_shares

or indirectly via setting the task's nice value using nice or renice. A task may be allocated between 1 and 420 shares with 20 shares being the default allocation. A nice value ≥ 0 is mapped to $(20 - \text{nice})$ shares and a value < 0 is mapped to $(20 + \text{nice} * \text{nice})$ shares. If shares are set directly via /proc/<pid>/cpu_shares then its nice value will be adjusted accordingly.

CPU usage rate caps

A task's CPU usage rate cap imposes a soft (or hard) upper limit on the rate at which it can use CPU resources and can be set/read via the files

/proc/<pid>/cpu_rate_cap
/proc/<tgid>/task/<pid>/cpu_rate_cap

Usage rate caps are expressed as rational numbers (e.g. "1 / 2") and hard caps are signified by a "!" suffix. The rational number indicates the proportion of a single CPU's capacity that the task may use. The value of the number must be in the range 0.0 to 1.0 inclusive for soft caps. For hard caps there is an additional restriction that a value of 0.0 is not permitted. Tasks with a soft cap of 0.0 become true background tasks and only get to run when no other tasks are active.

When hard capped tasks exceed their cap they are removed from the run queues and placed in a "sinbin" for a short while until their usage rate decays to within limits.

Scheduler Tuning Parameters

The characteristics of the Kalman filter used for usage rate estimates are determined by the `_response_half_life_`. The default value for the half life is 5000 msec, but this can be set to any value between 1000 and 100000 msec during a make config. Also, if the `SCHED_DYNAMIC_HALF_LIFE` config option is set to Y, the half life can be modified dynamically on a running system within the above range by writing to /proc/cpu_half_life.

Currently EBS gives all tasks a fixed default timeslice of 100msec. As with the half life, this can be chosen at build time (between 1 and 500msec) and building with the `SCHED_DYNAMIC_TIME_SLICE` option enables on–the–fly changes via /proc/timeslice.

Performance of the EBS scheduler depends on these parameters, as well as the load characteristics of the system. The current default settings, however, have worked well with most loads so far.

Scheduler statistics

Linux–Kernel: [RFC][PATCH] O(1) Entitlement Based Scheduler

Global and per task scheduling statistics are available via /proc. To reduce the size of this post, the details are not listed here but can be seen at <http://ebs.aurema.com>.

Implementation

Those interested mainly in EBS performance can skip this section...

Effective entitlement per share

Ideally, the CPU USAGE PER SHARE of tasks demanding as much CPU as they are entitled to should be equal. By keeping track of the HIGHEST CPU usage per share that has been observed and comparing it to the CPU usage per share for each task that runs, tasks that are receiving less usage per share than the one getting the most can be given a better priority, so they can "catch up".

This highest value of CPU usage per share is maintained on each runqueue as that CPU's `_effective entitlement per share_`, and is used as a basis for all priority computations on that CPU.

In a nutshell, the task that is receiving the most CPU usage for each of its shares serves as the yardstick via which the treatment of other tasks on that CPU are measured.

Task priorities

The array switch and interactivity estimator have been removed – a task's eligibility to run is determined purely by its priority. A task's priority is recalculated when it is forked, wakes up, uses up its timeslice or is preempted.

The following ratio:

$$\frac{\text{task's usage_per_share}}{\text{min(task's CPU rate cap per share, rq->effective_entitlement_per_share)}}$$

[where CPU rate cap per share is simply (CPU rate cap / CPU shares)]

is then mapped onto the SCHED_NORMAL priority range. The mapping is such that a ratio of 1.0 is equivalent to the mean SCHED_NORMAL priority, and ratios less than and greater than 1.0 are mapped to priorities less and greater than the mean respectively. This serves to boost tasks using less than their entitlement and penalise those using more than their entitlement or CPU rate cap. It also provides interactive and I/O bound tasks with favourable priorities since such tasks have inherently low CPU usage.

Linux-Kernel: [RFC][PATCH] O(1) Entitlement Based Scheduler

Lastly, the `->prio` field in the task structure has been eliminated. The `runqueue` structure stores the priority of the currently running task, and `enqueue/dequeue_task()` have been adapted to work without `->prio`. The reason for getting rid of `->prio` is to facilitate the O(1) priority promotion of runnable tasks, explained below.

Only one heuristic

All the heuristics used by the stock scheduler have been removed. The EBS scheduler uses only one heuristic: newly forked child tasks are given the same usage rate as their parent, rather than a zero usage. This is done to mollify the "ramp up" effect to some extent. "Ramp up" is the delay between a change in a task's usage rate and the Kalman filter estimating the new rate, which in this case could cause a parent to be swamped by its children. Total elimination of ramp up is undesirable as it is also responsible for good responsiveness of interactive processes.

O(1) task promotion

When the system is busy, tasks waiting on run queues will have decaying usages, which means that eventually tasks which have been waiting long enough will be entitled to a priority boost. Not giving them a boost will result in unfair starvation, but on the other hand periodically visiting every runnable task is an O(n) operation.

By inverting the priority and Kalman filter functions, it is possible to determine at priority calculation time after how long a task will be entitled to be promoted to the next best priority. These "promotion times" for each `SCHED_NORMAL` priority are then divided by the smallest of these times (call this the `promotion_interval`) to obtain the "number of promotion intervals" for each priority. Naturally these are stored in a table indexed by priority.

`enqueue_task()` now places `SCHED_NORMAL` tasks onto an appropriate "promotion list" as well as the run queue. The list is determined by the enqueued task's current priority and the number of promotion intervals that must pass before it is eligible to be bumped up to the next best priority. And of course `dequeue_task()` takes tasks off their promotion list. Therefore, tasks that get to run before they are due for a promotion (which is usually the case) don't get one.

Every `promotion_interval` jiffies, `scheduler_tick()` looks at only the promotion lists that are now due for a priority bump, and anything on the lists is given the required boost. (The highest `SCHED_NORMAL` and background priorities are ignored, as these tasks don't need promotion). Regardless of the number that need promoting, this is done in O(1) time. The promotion code exploits the fact that tasks of the same priority that are due for promotion at the same time, ie. are contiguous on a promotion list, ARE ALSO CONTIGUOUS ON THE RUN QUEUE. Therefore, promoting N tasks at priority X is simply a matter of "lifting" these tasks out of their current place on the runqueue in one

Linux–Kernel: [RFC][PATCH] O(1) Entitlement Based Scheduler

chunk, and appending this chunk to the $(X - 1)$ priority list. These tasks are then placed onto a new promotion list according to their new $(X - 1)$ priority.

This simple list operation is made possible by not having to update each task's \rightarrow prio field (now that it has been removed) when moving them to their new position on the runqueue.

Benchmarks

=====

Benchmarking was done using contest. The following are the results of running on a dual PIII 866MHz with 256MB RAM.

no_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	78	166.7	0	26.9	1.00
2.6.2–EBS	3	74	175.7	0	16.2	1.00

cacheron:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	75	173.3	0	24.0	0.96
2.6.2–EBS	3	71	183.1	0	12.7	0.96

process_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	93	138.7	35	54.8	1.19
2.6.2–EBS	3	91	141.8	33	53.8	1.23

ctar_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	99	141.4	1	5.1	1.27
2.6.2–EBS	3	95	145.3	1	4.2	1.28

xtar_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	93	147.3	1	9.7	1.19
2.6.2–EBS	3	89	152.8	1	6.7	1.20

io_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	99	144.4	4	14.1	1.27
2.6.2–EBS	3	91	153.8	3	10.9	1.23

read_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	193	73.6	13	7.8	2.47
2.6.2–EBS	3	136	101.5	7	6.6	1.84

list_load:

Kernel	[runs]	Time	CPU%	Loads	LCPU%	Ratio
2.6.2	3	83	160.2	0	4.8	1.06

Linux–Kernel: [RFC][PATCH] O(1) Entitlement Based Scheduler

2.6.2–EBS 3 80 165.0 0 2.5 1.08

mem_load:

Kernel [runs] Time CPU% Loads LCPU% Ratio

2.6.2 3 159 87.4 129 3.1 2.04

2.6.2–EBS 3 126 110.3 50 2.4 1.70

dbench_load:

Kernel [runs] Time CPU% Loads LCPU% Ratio

2.6.2 3 123 109.8 1 21.1 1.58

2.6.2–EBS 3 130 103.1 1 20.0 1.76

The big winners are read_load and mem_load, with all the others except dbench being slightly faster than the stock kernel. Only dbench was slightly worse.

X Windows Performance

=====

The X server isn't strictly an interactive process, but it does have a major influence on interactive response. The fact that it services a large number of clients means that its CPU usage rate can be quite high, and this negates the above mentioned favourable treatment of interactive and I/O bound processes.

Therefore, for best interactive feel, it is recommended that the X server run with a nice value of at least –15. From my own testing, doing a window wiggle test with a make –j16 in the background and X reniced was slightly better than for the stock kernel.

When running apps such as xmms, I recommend that they should be reniced as well when the background load is high. With the above setup and xmms reniced to –9, there were no sound skips at all (without renicing, a few skips could be detected).

Getting the patch

=====

The patch can be downloaded from

<<http://sourceforge.net/projects/ebs–linux/>>

Please note that there are 2 patches: the basic patch and the full patch. The above description applies to the full patch. The basic patch only features setting shares via nice, a fixed half life and timeslice, no statistics and soft caps only. This basic patch is for those who are mainly interested in looking at the core EBS changes to the stock scheduler.

The patches are against 2.6.2 (2.6.3 patches will be available shortly).

Comments, suggestions and testing are much appreciated.

Linux-Kernel: [RFC][PATCH] O(1) Entitlement Based Scheduler

The mailing list for this project is at
<<http://lists.sourceforge.net/lists/listinfo/ebs-linux-devel>>.

Cheers,

John

—

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
the body of a message to majordomo@vger.kernel.org

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>