Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Contents

CPU bandwidth control for CFS **11**
P. Turner, B. B. Rao, N. Rao

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

CPU bandwidth control for CFS

Paul Turner
Google
pjt@google.com

Bharata B Rao
IBM India Software Labs, Bangalore
bharata@linux.vnet.ibm.com

Nikhil Rao
Google
ncrao@google.com

Abstract

Over the past few years there has been an increasing focus on the development of features which deliver resource management within the Linux kernel. The addition of the fair group scheduler has enabled the provisioning of proportional CPU time through the specification of group weights. As the scheduler is inherently work-conserving in nature, a task or a group may consume excess CPU share in an otherwise idle system. There are many scenarios where this unbounded CPU share may lead to unacceptable utilization or latency variation. CPU bandwidth control approaches this problem by allowing an explicit upper bound for allowable CPU bandwidth to be defined in addition to the lower bound already provided by shares.

There are many enterprise scenarios where this functionality is useful. In particular are the cases of pay-per-use environments, and user facing services where provisioning is latency bounded.

In this paper we detail the motivations behind this feature, the challenges involved in incorporating into CFS (Completely Fair Scheduler), and the future development road map.

1 CPU as a manageable resource

Before considering the aspect of bandwidth provisioning let us first review some of the basic existing concepts currently arbitrating entity management within the scheduler.

There are two major scheduling classes within the Linux CPU scheduler, `SCHED_RT` and `SCHED_NORMAL`. When runnable, entities from the former, the *real-time* scheduling class, will always be elected to run over those from the *normal* scheduling class.

Prior to *v2.6.24*, the scheduler had no notion of any entity larger than that of single task[1]. The available management APIs reflected this and the primary control of bandwidth available was `nice(2)`.

In *v2.6.24*, the *completely fair scheduler* (CFS) was merged, replacing the existing `SCHED_NORMAL` scheduling class. This new design delivered *weight* based scheduling of CPU bandwidth, enabling arbitrary partitioning. This allowed support for *group scheduling* to be added, managed using *cgroups* through the *CPU controller* sub-system.

This support allows for the flexible creation of scheduling groups, allowing the fraction of CPU resources received by a group of tasks to be arbitrated as a whole. The addition of this support has been a major step in scheduler development, enabling Linux to align more closely with enterprise requirements for managing this resouce.

The hierarchies supported by this model are flexible, and groups may be nested within groups. Each group entity's bandwidth is provisioned using a corresponding `shares` attribute which defines its weight. Similarly, the `nice(2)` API was subsumed to control the weight of an individual task entity.

Figure 1 shows the hierarchical groups that might be created in a typical university server to differentiate CPU bandwidth between users such as professors, students, and different departments.

One way to think about *shares* is that it provides lower-bound provisioning. When CPU bandwidth is scheduled at capacity, all runnable entities will receive bandwidth in accordance with the ratio of their share weight. It's key to observe here that not all entities may be runnable

[1] Recall that under Linux any kernel-backed thread is considered individual task entity, there is no typical notion of a *process* in scheduling context.

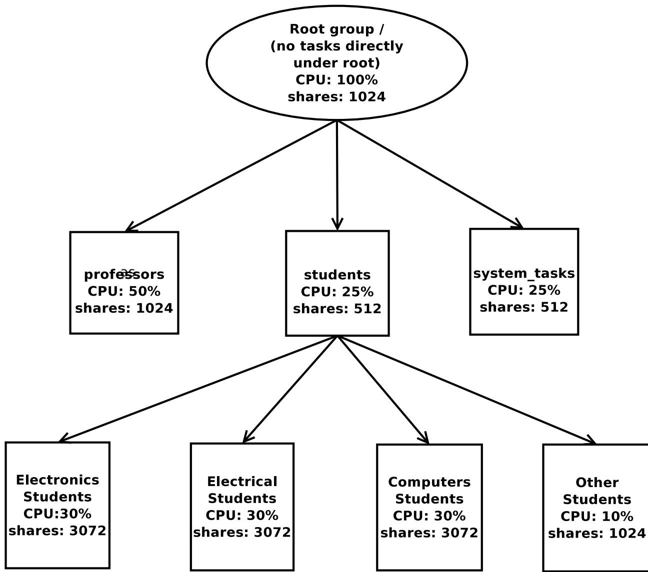

Figure 1: An example hierarchy applicable to a university server.

in this situation; this means that CPU bandwidth is comparatively available in abundance and the entities that are running will be able to consume bandwidth at a higher rate than their weight would permit were more entities runnable.

It should be noted that the concept of proportional shares is different from a guarantee. Assigning share to a group doesn't guarantee that it will get a particular amount of CPU. It only means that the available CPU bandwidth will be divided as per the shares. Hence depending on the number of groups present, the actual amount of CPU time obtained by groups can vary. [2]

For example: If there were 3 groups with 1024 shares each, then each would receive $\frac{1024}{1024+1024+1024} = 33.3\%$ of the CPU when all were runnable[3]. If a 4$^{\text{th}}$ group on the same level were to become active with a share of 2048, then the other groups would now receive only 20% of available bandwidth. The CPU bandwidth available to a group (by weight) is *always* relative.

[2]Also recall: These ratios are only relative to the time avai lable to SCHED_NORMAL. Time spent in SCHED_RT execution is independent of this mo del.

[3]Since group entities are containers, for a group entity to be runnable it must have an active child entity, the leaves of this tree must thusly all be task entities.

2 Motivation for bandwidth control

As discussed above, the scheduler is work conserving by nature; when idle cycles are available in the system, it is because there were no runnable entities available (on that cpu) to consume them. While for many use-cases, efficient use of idle CPU cycles like this might be considered optimal, there are two key side effects that must be considered:

1. The actual amount of CPU time available to a group is highly variable as it is dependent on the presence and execution patterns of other groups, a machine can the not be *predictably partitioned* without intimately understanding the behaviors of all co-scheduled applications.

2. The maximum amount of CPU time available to a group is not predictable. While this is closely related to the first point, the distinction is worth noting as this directly affects capacity planning.

While not of concern to most desktop users, these are key requirements in certain enterprise scenarios. Bandwidth control aims to address this by allowing upper limits on group bandwidths to be set. This allows both capacity and the maximal effect on other groups to be predicted.

This feature is already available for the *real-time* group scheduler (SCHED_RT). The first attempt to add this functionality to CFS (SCHED_NORMAL) was posted in June 2009 by the RFC post [2] to the Linux Kernel Mailing List (LKML). In the subsequent sections of this paper, we discuss this initial approach and how it has evolved into CFS Bandwidth Control below.

3 Example use cases

Bandwidth provisioning is commonly found useful in the following scenarios:

- Pay-per-use:

  In enterprise systems that cater to multiple clients/customers, a customer pays for, and is provisioned with a specific share of CPU resources. In such systems, customers would object should they receive less and its in the provider's interest that

they not provided more. In this case CPU bandwidth provisioning could be used directly to constrain the customers usage and provide soft bandwidth to interval guarantees. Such pay-per-use scenarios are frequently seen in *cloud* systems where service is priced by the required CPU capacity.

- Virtual Machines

  For (integrated) Linux based hypervisers such as KVM, bandwidth limits at the scheduler level may be useful to control the CPU entitlements of hosted VMs.

- Latency provisioning

  The explicit provisioning of containers within the machine allows for expectations to be set with respect to latency and worst-case access to CPU time. This becomes particularly important with non-homogenous collections of latency sensitive tasks where it is difficult to restrict co-scheduling.

- Guarantees

  In addition to maximum CPU bandwidth, in many situations applications may need CPU bandwidth guarantees. Currently this is not directly supported by the scheduler. In such cases, hard limits settings of different groups may be derived to reach a minimum (soft) guarantee for every group. An example of how to obtain guarantees for groups by using hard limit settings is provided in the OpenVZ wiki [1].

4 Interfaces

As discussed above the *cgroups* interface has been leveraged for managing the CPU controller subsystem. Our work extends these interfaces.

In case of `SCHED_RT`, bandwidth is specified using two control parameters: the enforcement interval (`cpu.rt_period_us`) and allowable consumption (`cpu.rt_runtime_us`) within that interval. Accounting is performed on a *per-CPU* basis. For example if there were 8 CPUs in the system [4], the group would be allowed to consume 8 times the `cpu.rt_runtime_us` within an interval of `cpu.rt_period_us`. This is enabled by allowing unconsumed time to be transferred from CPUs present in the `root_domain` span that have unconsumed bandwidth available.

In our initial approach [3], the bandwidth specification exposed for `SCHED_NORMAL` class was based on this model. However for the reasons described in the subsequent sections, we have now opted for global specifcation of both enforcement interval (`cpu.cfs_period_us`) and allowable bandwidth (`cpu.cfs_quota_us`). By specifying this, the group as a whole will be limited to `cpu.cfs_quota_us` units of CPU time within the period of `cpu.cfs_period_us`.

Of note is that these limits are hierarchical, unlike `SCHED_RT` we do not currently perform feasibility evaluaion regarding the defined limits. If a child has a more permissive bandwidth allowance than its parent, it will be indirectly throttled when the parent's quota is exhausted.

Additionally, there is the global control: `/proc/sys/kernel/sched_cfs_bandwidth_slice_us`

This *sysctl* interface manages how many units are transferred from the global pool each time a local pool requires additional quota. The current default is 10ms. The details of this transfer process are discussed in later sections.

5 Existing Approaches

5.1 CFS hard limits

This was the first approach [3] at implementing bandwidth controls for CFS and was modelled on the existing bandwidth control scheme in use by the *real-time* scheduling class. The mechanism employed here is quite direct. Each group entity (specifically `cfs_rq` here) is provisioned locally with `cfs_runtime_us` units of time. CPUs are then allowed to borrow from one another within a given `root_domain`. This means that the *externally* visible bandwidth of the group is effectively the weight of the `root_domain` CPU mask multiplied by `cfs_runtime_us` (per `cfs_period_us`). When a CFS group consumes all its runtime and when there is nothing left to borrow from the other CPUs, the group is then throttled. At the end of the enforcement interval, the bandwidth gets replenished and the throttled group becomes eligible to run once again.

[4]Assuming the `root_domain` has not been partitioned via *cpusets*

A typical time line for a process that runs as part of a bandwidth controlled group under *CFS Hard Limits* appears as shown in Figure 2.

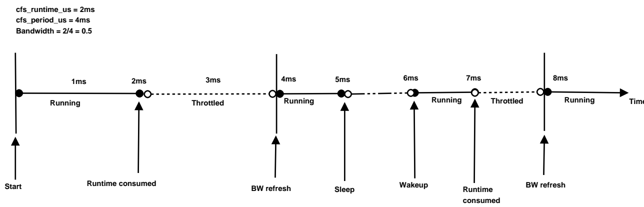

Figure 2: Progress of a task in bandwidth controlled group

Figure 3 shows the same situation with borrowing of quota from other cpus.

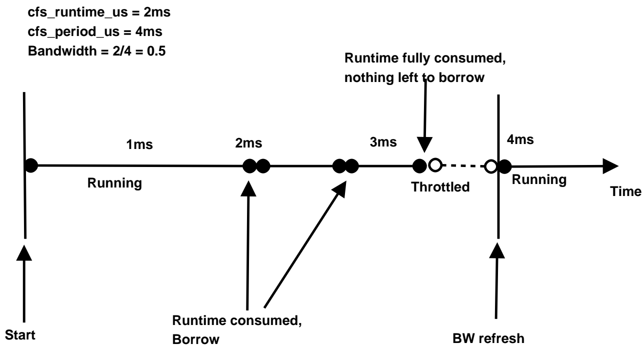

Figure 3: Progress of a task in bandwidth controlled group with runtime borrowing

The strategy for time-redistribution is to visit all neighbor CPUs and transfer $\frac{1}{n}$ of their remaining run-time, where n is the weight of the root_domain span. There is an implicit assumption made within this scheme that we will be able to converge quickly above to `cfs_period_us` while borrowing. This is indeed true for the *real-time* class as, by default, `SCHED_RT` is provisioned with 95% of total system time, allowing an individual CPU to reach to `rt_period_us` with only a single partial iteration of borrowing in the default configuration. This can also be expected to hold more generally as the nature of entities requiring this scheduling class is to be well-provisioned.

In the more general case, where the reservation may represent only a small-to-medium fraction of system resources, this convergence breaks down. Each iteration is able to maximally consume $\frac{n-1}{n}$ additional time, at the expense of taking every `rq->lock`. Moreover, as most cpus will not be allowed to reach the upper bound of the period we will have an extremely long tail of redistribution.

5.2 Hybrid global pool:

The primary scalability issue with the local pool approach is that there there is a many-to-many relationship in the computation and storage of remaining quota. This is acceptable provided either the existence of a strong convergence condition or 'small' SMP systems.

Tracking quota globally is also not a solution that scales with machine size due to the large contention the global store then experiences. One of the advantages of the local quota model above is that, when within quota, consumption is very efficient since it can potentially be accounted locklessly and involves no 'remote' queries.

Our design for the distribution of quota is a hybrid model which attempts to combine both local and global quota tracking. To each `task_group` a new `cfs_bandwidth` structure has been added. This tracks (globally) the allocated and consumed quota within a period. However, consumption does not occur against this pool directly; as in the local pool approach above there is a local, per `cfs_rq`, store of granted and consumed quota. This quota is acquired from the global pool in a (user configurable) batch sizer. When there is no quota available to re-provision a running `cfs_rq`, it is locally throttled until the next quota refresh. Bandwidth refresh is a periodic operation that occurs once per quota period within which all throttled run-queues are unthrottled and the global bandwidth pool is replenished.

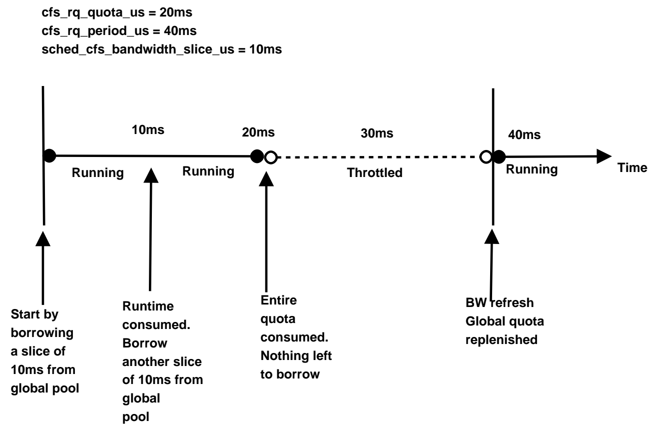

Figure 4: progress of a task in bandwidth controlled group with global quota

6 Design

CFS bandwidth control implements the above discussed hybrid approach to bandwidth accounting. The global quota is distributed in 'slices' to local per-CPU caches, where it is then consumed. The local accounting for these quota slices closely resembles the `SCHED_RT` case, however, the many-to-many CPU interactions on refresh and expiration are avoided. The batching of quota distribution also allows for linear convergence to quota within a provisioned period.

A summary of specific key changes is provided below.

6.1 Data-structures

`struct cfs_bandwidth`:

This is the top-level group representation of bandwidth, encapsulated within the corresponding `task_group` structure. The remaining `quota` within each period, as well as the total `runtime` assigned per period, and quota `period` length are managed here.

`struct cfs_rq`:

This is the per-group runqueue of all runnable entities present in `SCHED_NORMAL` group. Each group has one such representative runqueue on every CPU. The entities within a group are arranged in a time-ordered RB tree. Time ordering is done by *vruntime* or virutal runtime, which is a rough indication of the amount of CPU time obtained by an entity. We have annotated this structure with the new variables `quota_assigned` and `quota_used`, which track the total bandwidth allocated from the global pool to this runqueue and the amount consumed respectively.

6.2 Bandwidth distribution and constraint

`update_curr()`:

This function is periodically called from scheduler ticks as well as during other scheduler events like enqueue and dequeue to update the amount of time the currently running entity has received. Time since the last invocation is charged against the entity. The weight-normalized vruntime which forms the basis for fair-share scheduling is also updated here.

The accounting here is extended to call the new function `account_cfs_rq_quota`. This ensures that quota accounting will occur at the same instance at which execution time is charged.

`account_cfs_rq_quota()`:

This function forms the basis for quota distribution and tracking. The rough control flow is as shown in Figure 5:

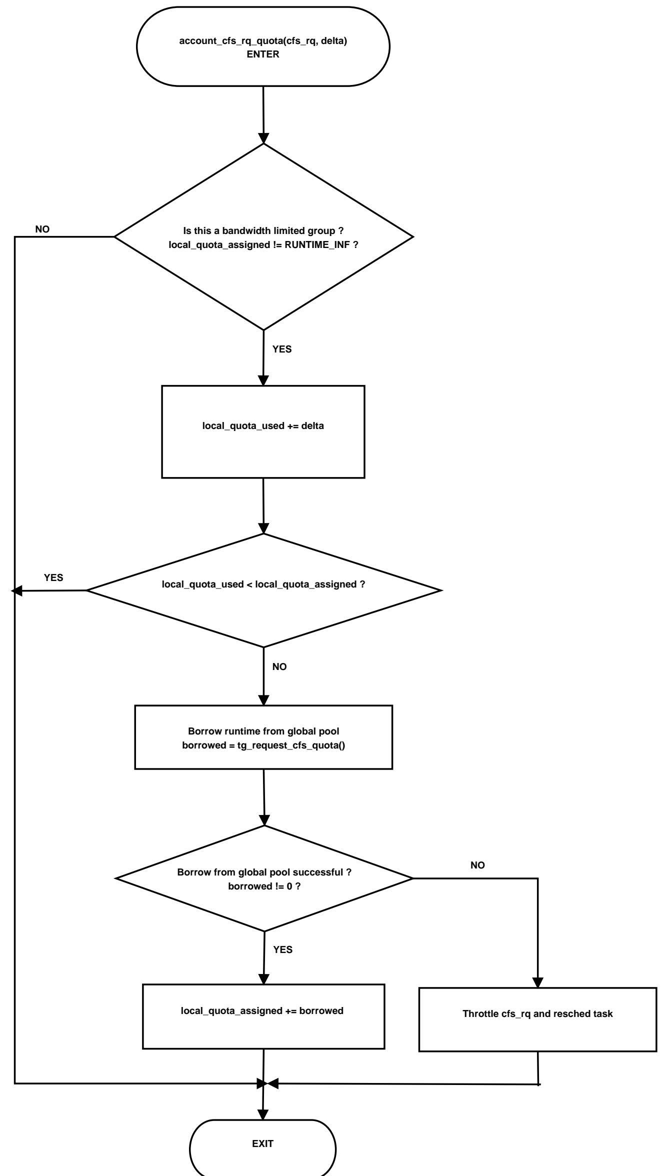

Figure 5: Control flow diagram for account_cfs_rq_quota()

Entity throttling

A throttled `cfs_rq` is one that has run out of local bandwidth (specifically, `local_quota_used` ≥

local_quota_assigned). By extension this means there was no global bandwidth availble to 'top-up' or refresh the pool. At this point the entity is no longer schedulable in the current quota period as it has reached its bandwidth limit. [5]

Such a cfs_rq is considered *throttled*. This state is tracked explicitly using the equivalently named attribute. Issuing a throttle operation (throttle_cfs_rq()) will dequeue the sched_entity from its parent cfs_rq and set the above throttled flag. Note that for an entity to cross this threshold it must be running, and thus, be the current entity. This means that the dequeue operation on the throttled entity will just update the accounting since the currently running entity is already dequeued form RB tree as per CFS design.

We must however, ensure that it is not able to re-enter the tree due to either a *put* operaton, or thread wake-up. The first case is handled naturally as the decision to return an entity to the RB tree is based off the entity->on_rq flag, which has been unset as a consequence of accounting during dequeue. The task wakeup case is handled directly by ceasing to enqueue past a throttled entity within enqueue_task_fair().

When the lone throttled entity of a parent is dequeued, the parent entity will suddenly become non-runnable, since it no longer has any runnable child entities. Even though the parent is not throttled, it need not remain on the RB tree. The natural solution to this is to continue dequeuing past the throttled entity until we reach an entity with load weight remaining. This is analogus to the ancestor dequeue that may occur when a nested task sleeps.

The unthrottle case is symmetric, the entity is re-enqueued and the throttled flag is cleared. Parenting entities must then potentially be enqueued up the tree hierarchy until we reach either the root, or an ancestor undergoing its own throttling operations.

Quota Refresh

task_group quota is refreshed periodically using *hrtimers* by programming the hrtimer to expire every cfs_period_us seconds. A struct hrtimer period_timer is embedded in cfs_bandwidth structure for this purpose. If there is a quota interval within which no bandwidth is consumed then the timer will not be re-programmed on expiration.

The refresh operation first refreshes the global quota pool, stored in the cfs_bandwidth structure. We then iterate over the per-CPU cfs_rq structures to determine whether any of them have been throttled. In the case of a throttled cfs_rq, we attempt to assign more bandwidth to it and – if successful – unthrottle it. Unlike SCHED_RT case, we could do this check for throttled cfs_rqs speculatively to reduce the contention on rq->lock. Future development here could involve refresh timer consolidation to further reduce overhead in the many *cgroup* case.

Locking Considerations

Since the locally assigned bandwidth is maintained on the cfs_rq, we are able to nest tracking and modification under the parent rq->lock. Since this lock is already held for existing CFS accounting, this allows the local tracking of quota to be performed with no additional locking.

Explicit locking is required to synchronize modification to the assigned or remaining bandwidth available in the global pool. Such an operation occurs when a local pool exceeds its (locally) assigned quota or through configuration change.

6.3 Challenges

- **Slack time handling**

  One caveat of our chosen approach is that the time locally assigned may have been allocated from the global pool in a previous quota interval. This represents potential local over-commit when bandwidth is expressed versus reservation. The maximum outstanding over-subscription within a given set of consecutive intervals is constant at $num_cpus \cdot batch_slice$. It is of note that this holds true for any number of consecutive observed interval since the input rate of the system is bounded, the over-commits occurs from remaining *slack* time that may be left over from the interval immediately prior to the first measured period. One potential approach to mitigate this for environments where

[5] Since bandwidth is defined on at the group level it should be noted that an individual task entity will never be throttled, only its parent.

harder limits are required is to use generation counters during the allocation and distribution of quota.

- **Fairness issues**

  The waterfall distribution of quota from global to local pools is also potentially accompanied by risks involving the fairness of consumption. Consider for example the case of a multi-CPU machine. Since the allocation of quota is currently in batch sized amounts it is possible for a multi-threaded application to experience reduced parallelism. It is also possible for quota to be 'stranded' as load-balancing leaves local quota unavailable for consumption due to no runnable entities. The latter can also potentially be addressed by a generational model as it would then be possible to return quota to the global pool on a voluntary sleep. However as systems scale the former potentially becomes a real problem. A short term mitigation strategy could be to reduce the batch-sizing used for the propagation of quota from global to local pools. Longer term strategies for resolving this issue might include a graduated scheme for batch slice sizing and subdividing the global pool at the `sched_domain` level to allow for finer granularity of control on distribution.

- **Load-balancer interactions**

  CFS bandwidth control currently supports only a very primitive model for load balancer interactions. Throttled run-queues are excised from load-balancing decisions; it turns out that it is hard to improve this model without undesirable emergent behaviors. At first inspection it may appear that migrating threads from a locally throttled run-queue to an unthrottled one would be a sensible decision. This quickly breaks down when the case of insufficient quota is considered. Here the last run-queue to have quota available will resemble that last seat in the game 'musical chairs', inadvertantly creating a 'herd' of executing threads. Likewise, it does not make sense to migrate a thread to a run-queue that has already been throttled as it being runnable indicates there is local quota still available.

  While improving the actual mechanics of load balancing in these conditions may be a large technical challenge, there may be an easier case to make for improvement in the distribution of share weight. Currently, when a run-queue is throttled its participation in group share distribution is also halted. This may result in undesirable rq weight fluctuations. One avenue that has been considered in this area is to continue allowing throttled entities to participate in weight calculations for re-distribution. This will allow entities to re-wake with their correct weight and prevent large swings in distribution. For this to work effectively however some stronger guarantees that quota will expire relatively concurrently are desired to avoid skews.

7 Results

This section describes the experiments we have undertaken to validate CFS bandwidth control. We describe our test setup and the benchmarks run. We then present our results and discuss some limitations in the current approach. Finally we explore the effects of changing parameters like `sched_slice` and enforcement periods in the system.

We performed all our tests on a 16-core AMD machine with no restrictions to affinity. We ran our tests with our patches based on top of the *v2.6.34* kernel, which as of writing this paper was the most recent stable kernel release. Outside the default x86 Kconfig, we enabled `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_CFS_BANDWIDTH_CONTROL`. We configured our test machine with the following *cgroup* configuration for all experiments.

Container	Shares	Notes
system	1024	Contains system tasks such as sshd, measurement tasks, etc.
protag	65536	Benchmark runs in this container. The protag container is large enough to mitigate system interference.

We monitored system utilization in a monitoring thread that was part of the system container. This thread woke up once a second and read `/proc/stat` for busy usage and idle time for all CPUs in the system. We used busy time measured as a fraction of the total system time as a metric to measure the effectiveness of CPU bandwidth control.

Two simple benchmarks were used to validate the bandwidth control mechanism - while-1 soakers and sysbench. Both these benchmarks are multi-threaded applications and can completely saturate the machine in

the absence of bandwidth control. We would also like to note that system daemon/thread interference was minimal and can be ignored for the purposes of this discussion.

When the runtime allocated to a cgroup is the same as its period, it can be expected to receive one CPU's worth of wall time. When the runtime is twice the period it gets about two CPUs worth, and so on. The same bandwidth limits were explicitly attained using the `cpuset` subsystem and CPU affinities as a control group. It should be noted that due to the lack of affinity and that time was observed on all runnable CPUs in the first case there is nothing special about the integral CPU case for *CFS Bandwidth Control*, it merely enables the use of cpusets as an `OPT` control parameter.

Each benchmark was run with three different bandwidth enforcement periods - 100ms, 250ms, and 500ms. In each set of runs, we allocated an integer core worth of runtime to the protag cgroup ranging from 1 core worth upto the full 16 cores worth of runtime. We compare these results with the baseline measurement for the respective benchmark.

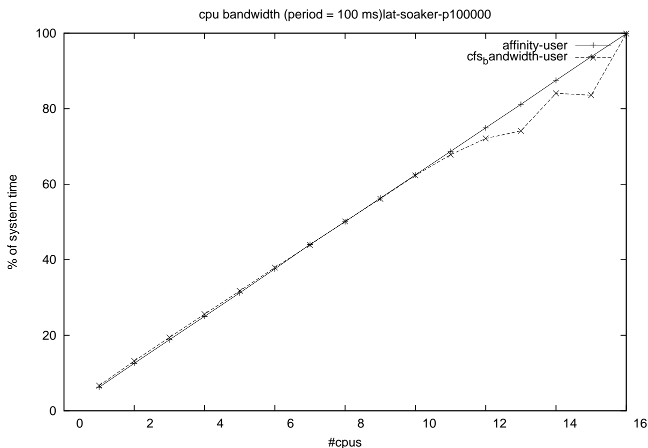

Figure 6: while-1 soakers, CFS Bandwidth Control, 100ms

Figures 6, 7 and 8 show the comparision of CPU times obtained by while-1 soakers when run with affinities and when run with bandwidth control. We see that the average deviation from the baseline benchmark is very small in each of these cases. We also notice that the average deviation from baseline decreases as the enforcement period increases.

Figures 9, 10 and 11 show the results for sysbench. The benchmark was to run the CPU sysbench test with 16

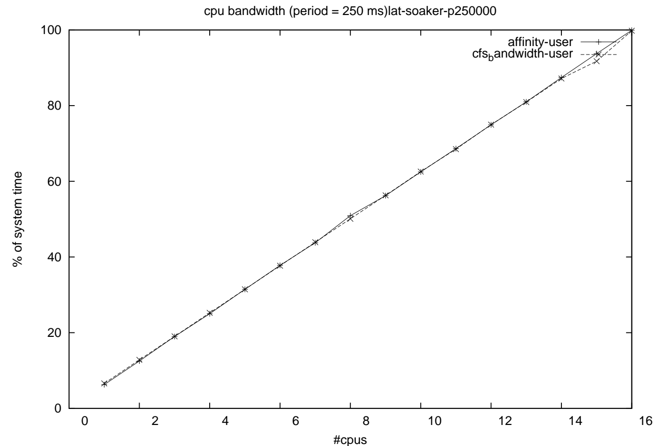

Figure 7: while-1 soakers CFS Bandwidth Control, 250ms

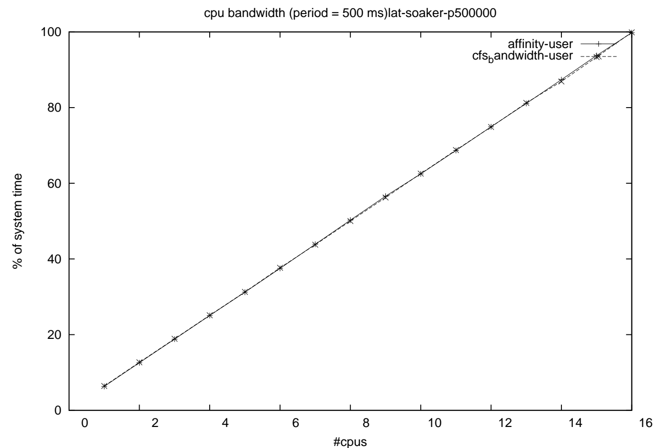

Figure 8: while-1 soakers, CFS Bandwidth Control, 500ms

worker threads. Each thread computed all prime numbers lesser than 100000. Again, we see that the deviation from the baseline benchmark is very small in most cases and improves as the enforcement period increases.

7.1 Overhead

CFS bandwidth control adds a minimal overhead to the enqueue/dequeue fast paths. We used tbench to measure overhead on these paths as it exercises these paths very frequently (450K times a second). We used on a vanilla 2.6.34 kernel with affinity masks as the baseline. The tabular data below shows the overhead with enforcement period at 100ms.

As described earlier, quota is distributed in `batch_slice` amounts of runtime. This is set to 10ms by

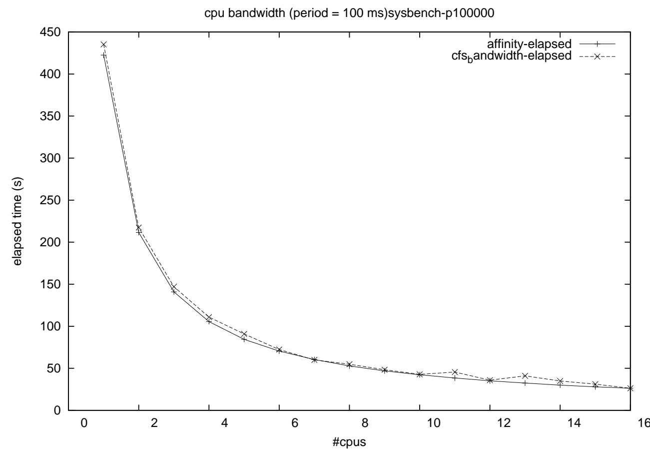

Figure 9: sysbench bandwidth control, 100ms

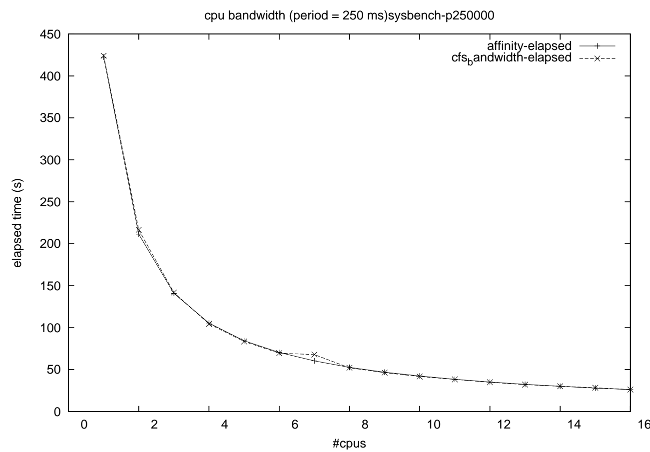

Figure 10: sysbench bandwidth control, 250ms

default in our current system, but we expose this as a tunable that can be set at runtime via a procfs tunable. While decreasing this value increases the frequency at which CPUs request for quota from the global pool, we did not notice any measurable impact on performance.

8 Conclusions and Futures

CFS Bandwidth Control is a light-weight and flexibile mechanism for bandwidth control and specification. We

cputime	baseline	bandwidth control
1	219.022	213.415
8	1668.12	1653.7
16	2451.82	2421.36

Table 1: Overhead of CFS bandwidth control, tbench 10 procs

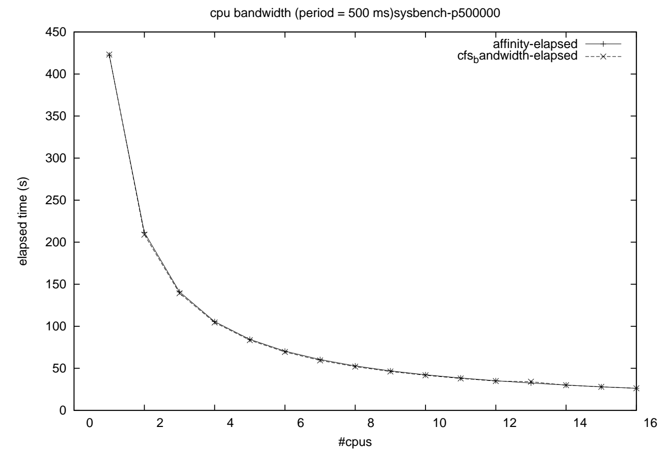

Figure 11: sysbench bandwidth control, 500ms

are currently in the process of attempting to collect test-data on a wider variety of both proprietary and open-source workloads and hope to publish this data as it becomes available.

Our patchset is in a stable state and we encourage any customers interested in this requirement to evaluate whether it meets their needs and provide feedback. The current posting is version 2, available at [4].

We are currently pursuing peer review with the hopes of merging this feature into the mainline scheduler tree. Looking forwards we are attempting to deliver improvements such as generational quota to mitigate potential slack time issues and improve fairness. For simplicity's sake given the review process however, formal consideration of this should be post-poned until the original approach reaches maturation within the community.

9 Acknowledgements

We would like to thank Ken Chen, Dhaval Giani, Balbir Singh and Srivatsa Vaddagiri for discussion related to the development of this work. Much credit is also due to the original bandwidth mechanisms present in `SCHED_RT` as they have both inspired and formed the basis for much of this work. We would also like to thank Google and IBM for funding this development.

10 Legal Statements

© International Business Machines Corporation 2010. Permission to redistribute in accordance with Linux

Symposium submission guidelines is granted; all other rights reserved.

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM, IBM logo, ibm.com, andWebSphere, are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION âĂIJAS ISâĂİ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time with

References

[1] Guarantees in OpenVZ. `http://wiki.openvz.org/Containers/Guarantees_for_resources`.

[2] Bharata B Rao. CFS hard limits - v0, June 2009. `http://lkml.org/lkml/2009/6/4/24`.

[3] Bharata B Rao. CFS hard limits - v5, January 2010. `http://lkml.org/lkml/2010/1/5/44`.

[4] Paul Turner. CFS bandwidth control - v2, April 2010. `http://lkml.org/lkml/2010/4/28/88`.