Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED_RT and SCHED_NORMAL tasks
Posted by Srivatsa Vaddagiri on Tue, 12 Jun 2007 13:30:45 GMT
View Forum Message <> Reply to Message

On Tue, Jun 12, 2007 at 02:23:38PM +0200, Dmitry Adamushko wrote:
> >mm ..
> >
> > exec_delta64 = this_lrq->delta_exec_clock + 1;
> > this_lrq->delta_exec_clock = 0;
> >
> >So exec_delta64 (and fair_delta64) should be min 1 in successive calls.
> >How can that lead to this_load = 0?
>
> just substitute {exec,fair}_delta == 1 in the following code:
>
> tmp64 = SCHED_LOAD_SCALE * exec_delta64;
> do_div(tmp64, fair_delta);
> tmp64 *= exec_delta64;
> do_div(tmp64, TICK_NSEC);
> this_load = (unsigned long)tmp64;
>
> we'd get
>
> tmp64 = 1024 * 1;
> tmp64 =/ 1;
> tmp64 *= 1;
> tmp64 /= 1000000;
>
> as a result, this_load = 1024/1000000; which is 0 (no floating point calc.).

Ok ..

But isn't that the same result we would have obtained anyways had we
called update_load_fair() on all lrq's on every timer tick? If a user's
lrq was inactive for several ticks, then its exec_delta will be seen as
zero for those several ticks, which means we would compute its 'this_load' to be
zero as well for those several ticks?

Basically what I want to know is, are we sacrificing any accuracy here
because of "deferring" smoothening of cpu_load for a (inactive) lrq
(apart from the inaccurate figure used during load_balance as you point
out below).

> >The idea behind 'replay lost ticks' is to avoid load smoothening of
> >-every- lrq -every- tick. Lets say that there are ten lrqs
> >(corresponding to ten different users). We load smoothen only the currently
> >active lrq (whose task is currently running).

>
> The raw idea behind update_load_fair() is that it evaluates the
> run-time history between 2 consequent calls to it (which is now at
> timer freq. --- that's a sapling period). So if you call
> update_fair_load() in a loop, the sampling period is actually an
> interval between 2 consequent calls. IOW, you can't say "3 ticks were
> lost" so at first evaluate the load for the first tick, then the
> second one, etc. ...

Assuming the lrq was inactive for all those 3 ticks and became active at
4th tick, would the end result of cpu_load (as obtained in my code) be
any different than calling update_load_fair() on all lrq on each tick?

> Anyway, I'm missing the details regarding the way you are going to do
> per-group 'load balancing' so refrain from further commenting so
> far... it's just that the current implementation of update_load_fair()
> is unlikely to work as you expect in your 'replay lost ticks' loop :-)

Even though this lost ticks loop is easily triggered with user-based lrqs,
I think the same "loop" can be seen in current CFS code (i.e say v16)
when low level timer interrupt handler replays such lost timer ticks (say we
were in a critical section for some time with timer interrupt disabled).
As an example see arch/powerpc/kernel/time.c:timer_interrupt() calling
account_process_time->scheduler_tick in a loop.

If there is any bug in 'replay lost ticks' loop in the patch I posted, then
it should already be present in current (i.e v16) implementation of
update_load_fair()?

> >Other lrqs load get smoothened
> >as soon as they become active next time (thus catching up with all lost
> >ticks).
>
> Ok, let's say user1 tasks were highly active till T1 moment of time..
> cpu_load[] of user's lrq
> has accumulated this load.
> now user's tasks were not active for an interval of dT.. so you don't
> update its cpu_load[] in the mean time? Let's say 'load balancing'
> takes place at the moment T2 = T1 + dT
>
> Are you going to do any 'load balancing' between users? Based on what?

Yes, patch #5 introduces group-aware load-balance. It is two-step:

First, we identify busiest group and busiest queue, based on
rq->raw_weighted_load/cpu_load (which is accumulation of weight from all
clases on a CPU). This part of the code is untouched.

Next when loadbalancing between two chosen CPUs (busiest and this cpu), move_tasks() is iteratively called on each user/group's lrq on both cpus, with the max_load_move argument set to 1/2 the imabalnce between that user's lrqs on both cpus. For this lrq imbalance calculation, I was using lrq->raw_weighted_load from both cpus, though I agree using lrq->cpu_load is a better bet.

> If it's user's lrq :: cpu_load[] .. then it _still_ shows the load at
> the moment of T1 while we are at the moment T2 (and user1 was not
> active during dT)..

Good point. So how do we solve this? I really really want to avoid running update_load_fair() on all lrq's every tick (it will be a massive overhead). I am assuming that lrqs don't remain inactive for a long time (given CFS's fairness promise!) and hence probably their cpu_load[] also won't be -that- stale in practice?

--
Regards,
vatsa

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers