Subject: Re: [RFC][PATCH 4/6] Fix (bad?) interactions between SCHED_RT and SCHED_NORMAL tasks
Posted by Dmitry Adamushko on Tue, 12 Jun 2007 12:23:38 GMT
View Forum Message <> Reply to Message

On 12/06/07, Srivatsa Vaddagiri <vatsa@linux.vnet.ibm.com> wrote:
> On Tue, Jun 12, 2007 at 11:03:36AM +0200, Dmitry Adamushko wrote:
> > I had an idea of per-sched-class 'load balance' calculator. So that
> > update_load() (as in your patch) would look smth like :
> >
> > ...
> > struct sched_class *class = sched_class_highest;
> > unsigned long total = 0;
> >
> > do {
> > total += class->update_load(..., now);
> > class = class->next;
> > } while (class);
> > ...
> >
> > and e.g. update_load_fair() would become a fair_sched_class ::
> > update_load().
> >
> > That said, all the sched_classes would report a load created by their
> > entities (tasks) over the last sampling period. Ideally, the
> > calculation should not be merely based on the 'raw_weighted_load' but
> > rather done in a similar way to update_load_fair() as in v17.
>
> I like this idea. It neatly segregates load calculation across classes.
> It effectively replaces what update_load() function I introduced in
> Patch #4.

Good.

(a minor disclaimer :)
We discussed it a bit with Ingo and I don't remember who first
expressed this idea in written words (although I seem to remember, I
did have it in mind before -- it's not rocket science after all :)

>
> Btw what will update_load_rt() return?

Well, as a _temporary_ stub - just return the 'raw_weighted_load'
contributed by the RT tasks..
Ideally, we'd like a similar approach to the update_fair_load() --
i.e. we need the run-time history of rt_sched_class's (like of any
other class) tasks over the last sampling period, so e.g. we do
account periodic RT tasks which happen to escape accounting through

'raw_weghted_load' due to the fact that they are not active at the
moment of timer interrupts (when 'raw_weighted_load' snapshots are
taken).

>
> > > static void entity_tick(struct lrq *lrq, struct sched_entity *curr)
> > > {
> > > struct sched_entity *next;
> > > struct rq *rq = lrq_rq(lrq);
> > > u64 now = __rq_clock(rq);
> > >
> > >+ /* replay load smoothening for all ticks we lost */
> > >+ while (time_after_eq64(now, lrq->last_tick)) {
> > >+ update_load_fair(lrq);
> > >+ lrq->last_tick += TICK_NSEC;
> > >+ }
> >
> > I think, it won't work properly this way. The first call returns a
> > load for last TICK_NSEC and all the consequent ones report zero load
> > ('this_load = 0' internally)..
>
> 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.).

> 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. ...

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 :-)

> 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?
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)..

>
> --
> Regards,
> vatsa
>

--
Best regards,
Dmitry Adamushko

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