Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by William Lee Irwin III on Sat, 26 May 2007 15:41:12 GMT
View Forum Message <> Reply to Message

Srivatsa Vaddagiri wrote:
>> Ingo/Peter, any thoughts here? CFS and smpnice probably is "broken"
>> with respect to such example as above albeit for nice-based tasks.

On Sat, May 26, 2007 at 10:17:42AM +1000, Peter Williams wrote:
> See above. I think that faced with cpu affinity use by the system
> administrator that smpnice will tend towards a task to cpu allocation
> that is (close to) the best that can be achieved without violating the
> cpu affinity assignments. (It may take a little longer than normal but
> it should get there eventually.)
> You have to assume that the system administrator knows what (s)he's
> doing and is willing to accept the impact of their policy decision on
> the overall system performance.
> Having said that, if it was deemed necessary you could probably increase
> the speed at which the load balancer converged on a good result in the
> face of cpu affinity by keeping a "pinned weighted load" value for each
> run queue and using that to modify find_busiest_group() and
> find_busiest_queue() to be a bit smarter. But I'm not sure that it
> would be worth the added complexity.

Just in case anyone was looking for algorithms...

Lag should be considered in lieu of load because lag is what the
scheduler is trying to minimize; load is not directly relevant, but
appears to have some sort of relationship. Also, instead of pinned,
unpinned should be considered. It's unpinned that load balancing can
actually migrate. Using the signed minimax pseudonorm (i.e. the highest
signed lag, where positive is higher than all negative regardless of
magnitude) on unpinned lags yields a rather natural load balancing
algorithm consisting of migrating from highest to lowest signed lag,
with progressively longer periods for periodic balancing across
progressively higher levels of hierarchy in sched_domains etc. as usual.
Basically skip over pinned tasks as far as lag goes.

The trick with all that comes when tasks are pinned within a set of
cpus (especially crossing sched_domains) instead of to a single cpu.
There one can just consider a cpu to enter a periodic load balance
cycle, and then consider pushing and pulling, perhaps what could be
called the "exchange lags" for the pair of cpus. That would be the
minimax lag pseudonorms for the tasks migratable to both cpus of the
pair. That makes the notion of moving things from highest to lowest
lag (where load is now considered) unambiguous apart from whether all
this converges, but not when to actually try to load balance vs. when
not to, or when it's urgent vs. when it should be done periodically.

To clarify that, an O(cpus**2) notion appears to be necessary, namely the largest exchange lag differential between any pair of cpus. There is also the open question of whether moving tasks between cpus with the highest exchange lag differential will actually reduce it or whether it runs the risk of increasing it by creating a larger exchange lag differential between different pairs of cpus. A similar open question is raised by localizing balancing decisions to sched_domains. What remains clear is that any such movement reduces the worst-case lag in the whole system. Because of that, the worst-case lag in the whole system monotonically decreases as balancing decisions are made, and that much is subject to an infinite descent argument. Unfortunately, determining the largest exchange lag differential appears to be more complex than merely finding the highest and lowest lags. Bipartite forms of the problem also arise from sched_domains.

I doubt anyone's really paying any sort of attention, so I'll not really bother working out much more in the way of details with respect to load balancing. It may be that there are better ways to communicate algorithmic notions than prose descriptions. However, it's doubtful I'll produce anything in a timely enough fashion to attract or hold interest.

The smpnice affair is better phrased in terms of task weighting. It's simple to honor nice in such an arrangement. First unravel the grouping hierarchy, then weight by nice. This looks like

```
task    nice    hier1   hier2   ...     hierN
t_1     w_n1    w_h11   w_h21   ...     w_hN1
t_2     w_n2    w_h12   w_h22   ...     w_hN2
...
```

For the example of nice 0 vs. nice 10 as distinct users with 10% steppings between nice levels, one would have

```
task    nice   hier1
t_1     1      1
t_2     0.3855 1
```

w_1, the weight of t_1, would be
(w_h11*w_n1/(w_h11*w_n1 + w_h12*w_n2))
= (1*1/(1 + 1*0.3855..))
= 0.7217..
w_2, the weight of t_2, would be
(w_h12*w_n2/(w_h11*w_n1 + w_h12*w_n2))
= (1*0.3855../(1 + 1*0.3855..))
= 0.27826..
This just so happens to work out to being the same as if t_1 and t_2 had their respective nice numbers without the scheduler grouping, which

is basically what everyone wants to happen.

It's more obvious how to extend it to more tasks than levels of
hierarchy. An example of that follows:

task	nice	hier1	hier2	...	hierN
t_1	0.3	0.6	*	...	*
t_2	0.7	0.4	*	...	*

hier2 through hierN are ignorable since t_1 and t_2 are both the only
members at those levels of hierarchy. We then get something just like
the above example, w_1 = 0.3*0.6/(0.3*0.6+0.7*0.4) = 0.3913.. and
w2 = 0.7*0.4/(0.3*0.6+0.7*0.4) = 0.6087..

It's more interesting with enough tasks to have more meaningful levels
of hierarchy.

task	nice	hier1	hier2
t_1	0.7	0.6	0.6
t_2	0.3	0.6	0.4
t_3	0.7	0.4	0.6
t_4	0.3	0.4	0.4

where t_1 and t_2 share a hier1 grouping and t_3 and t_4 also share
a hier1 grouping, but the hier1 grouping for t_1 and t_2 is distinct
from the hier1 grouping for t_3 and t_4. All hier2 groupings are
distinct. So t_1 would have pre-nice weight 0.6*0.6, t_2 0.6*0.4,
t_3 0.6*0.4, and t_4 0.4*0.4 (the numbers were chosen so denominators
conveniently collapse to 1). Now that the hierarchy is flattened,
nice numbers can be factored in for t_1's final weight being
0.7*0.36/(0.7*0.36+0.3*0.24+0.7*0.24+0.3*0.16) = 0.252/0.54 = 0.467..
and the others being 0.133.. (t_2), 0.311.. (t_3), and 0.0889.. (t_4).

In such a manner nice numbers obey the principle of least surprise.

-- wli

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Peter Williams on Sun, 27 May 2007 01:29:51 GMT
View Forum Message <> Reply to Message

William Lee Irwin III wrote:

> Srivatsa Vaddagiri wrote:
>>> Ingo/Peter, any thoughts here? CFS and smpnice probably is "broken"
>>> with respect to such example as above albeit for nice-based tasks.
>
> On Sat, May 26, 2007 at 10:17:42AM +1000, Peter Williams wrote:
>> See above. I think that faced with cpu affinity use by the system
>> administrator that smpnice will tend towards a task to cpu allocation
>> that is (close to) the best that can be achieved without violating the
>> cpu affinity assignments. (It may take a little longer than normal but
>> it should get there eventually.)
>> You have to assume that the system administrator knows what (s)he's
>> doing and is willing to accept the impact of their policy decision on
>> the overall system performance.
>> Having said that, if it was deemed necessary you could probably increase
>> the speed at which the load balancer converged on a good result in the
>> face of cpu affinity by keeping a "pinned weighted load" value for each
>> run queue and using that to modify find_busiest_group() and
>> find_busiest_queue() to be a bit smarter. But I'm not sure that it
>> would be worth the added complexity.
>
> Just in case anyone was looking for algorithms...
>
> Lag should be considered in lieu of load because lag

What's the definition of lag here?

> is what the
> scheduler is trying to minimize;

This isn't always the case. Some may prefer fairness to minimal lag.
Others may prefer particular tasks to receive preferential treatment.

> load is not directly relevant, but
> appears to have some sort of relationship. Also, instead of pinned,
> unpinned should be considered.

If you have total and pinned you can get unpinned. It's probably
cheaper to maintain data for pinned than unpinned as there's less of it
on normal systems.

> It's unpinned that load balancing can
> actually migrate.

True but see previous comment.

> Using the signed minimax pseudonorm (i.e. the highest
> signed lag, where positive is higher than all negative regardless of
> magnitude) on unpinned lags yields a rather natural load balancing

> algorithm consisting of migrating from highest to lowest signed lag,
> with progressively longer periods for periodic balancing across
> progressively higher levels of hierarchy in sched_domains etc. as usual.
> Basically skip over pinned tasks as far as lag goes.
>
> The trick with all that comes when tasks are pinned within a set of
> cpus (especially crossing sched_domains) instead of to a single cpu.

Yes, this makes the cost of maintaining the required data higher which
makes keeping pinned data more attractive than unpinned.

BTW keeping data for sets of CPU affinities could cause problems as the
number of possible sets is quite large (being 2 to the power of the
number of CPUs). So you need an algorithm based on pinned data for
single CPUs that knows the pinning isn't necessarily exclusive rather
than one based on sets of CPUs. As I understand it (which may be
wrong), the mechanism you describe below takes that approach.

> There one can just consider a cpu to enter a periodic load balance
> cycle, and then consider pushing and pulling, perhaps what could be
> called the "exchange lags" for the pair of cpus. That would be the
> minimax lag pseudonorms for the tasks migratable to both cpus of the
> pair. That makes the notion of moving things from highest to lowest
> lag (where load is now considered) unambiguous apart from whether all
> this converges, but not when to actually try to load balance vs. when
> not to, or when it's urgent vs. when it should be done periodically.
>
> To clarify that, an O(cpus**2) notion appears to be necessary, namely
> the largest exchange lag differential between any pair of cpus. There
> is also the open question of whether moving tasks between cpus with the
> highest exchange lag differential will actually reduce it or whether it
> runs the risk of increasing it by creating a larger exchange lag
> differential between different pairs of cpus. A similar open question
> is raised by localizing balancing decisions to sched_domains. What
> remains clear is that any such movement reduces the worst-case lag in
> the whole system. Because of that, the worst-case lag in the whole
> system monotonically decreases as balancing decisions are made, and
> that much is subject to an infinite descent argument. Unfortunately,
> determining the largest exchange lag differential appears to be more
> complex than merely finding the highest and lowest lags. Bipartite
> forms of the problem also arise from sched_domains.
>
> I doubt anyone's really paying any sort of attention, so I'll not
> really bother working out much more in the way of details with respect
> to load balancing. It may be that there are better ways to communicate
> algorithmic notions than prose descriptions. However, it's doubtful I'll
> produce anything in a timely enough fashion to attract or hold interest.
>

> The smpnice affair is better phrased in terms of task weighting. It's
> simple to honor nice in such an arrangement. First unravel the
> grouping hierarchy, then weight by nice. This looks like
>
> task nice hier1 hier2 ... hierN
> t_1 w_n1 w_h11 w_h21 ... w_hN1
> t_2 w_n2 w_h12 w_h22 ... w_hN2
> ...
>
> For the example of nice 0 vs. nice 10 as distinct users with 10%
> steppings between nice levels, one would have
>
> task nice hier1
> t_1 1 1
> t_2 0.3855 1
>
> w_1, the weight of t_1, would be
> (w_h11*w_n1/(w_h11*w_n1 + w_h12*w_n2))
> = (1*1/(1 + 1*0.3855..))
> = 0.7217..
> w_2, the weight of t_2, would be
> (w_h12*w_n2/(w_h11*w_n1 + w_h12*w_n2))
> = (1*0.3855../(1 + 1*0.3855..))
> = 0.27826..
> This just so happens to work out to being the same as if t_1 and t_2
> had their respective nice numbers without the scheduler grouping, which
> is basically what everyone wants to happen.
>
> It's more obvious how to extend it to more tasks than levels of
> hierarchy. An example of that follows:
>
> task nice hier1 hier2 ... hierN
> t_1 0.3 0.6 * ... *
> t_2 0.7 0.4 * ... *
>
> hier2 through hierN are ignorable since t_1 and t_2 are both the only
> members at those levels of hierarchy. We then get something just like
> the above example, w_1 = 0.3*0.6/(0.3*0.6+0.7*0.4) = 0.3913.. and
> w2 = 0.7*0.4/(0.3*0.6+0.7*0.4) = 0.6087..
>
> It's more interesting with enough tasks to have more meaningful levels
> of hierarchy.
>
> task nice hier1 hier2
> t_1 0.7 0.6 0.6
> t_2 0.3 0.6 0.4
> t_3 0.7 0.4 0.6
> t_4 0.3 0.4 0.4

>
> where t_1 and t_2 share a hier1 grouping and t_3 and t_4 also share
> a hier1 grouping, but the hier1 grouping for t_1 and t_2 is distinct
> from the hier1 grouping for t_3 and t_4. All hier2 groupings are
> distinct. So t_1 would have pre-nice weight 0.6*0.6, t_2 0.6*0.4,
> t_3 0.6*0.4, and t_4 0.4*0.4 (the numbers were chosen so denominators
> conveniently collapse to 1). Now that the hierarchy is flattened,
> nice numbers can be factored in for t_1's final weight being
> 0.7*0.36/(0.7*0.36+0.3*0.24+0.7*0.24+0.3*0.16) = 0.252/0.54 = 0.467..
> and the others being 0.133.. (t_2), 0.311.. (t_3), and 0.0889.. (t_4).
>
> In such a manner nice numbers obey the principle of least surprise.

Is it just me or did you stray from the topic of handling cpu affinity
during load balancing to hierarchical load balancing? I couldn't see
anything in the above explanation that would improve the handling of cpu
affinity.

Peter
--
Peter Williams pwil3058@bigpond.net.au

"Learning, n. The kind of ignorance distinguishing the studious."
  -- Ambrose Bierce

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

---

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by William Lee Irwin III on Tue, 29 May 2007 10:48:05 GMT
View Forum Message <> Reply to Message

William Lee Irwin III wrote:
>> Lag should be considered in lieu of load because lag

On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
> What's the definition of lag here?

Lag is the deviation of a task's allocated CPU time from the CPU time
it would be granted by the ideal fair scheduling algorithm (generalized
processor sharing; take the limit of RR with per-task timeslices
proportional to load weight as the scale factor approaches zero).
Negative lag reflects receipt of excess CPU time. A close-to-canonical
"fairness metric" is the maximum of the absolute values of the lags of
all the tasks on the system. The "signed minimax pseudonorm" is the
largest lag without taking absolute values; it's a term I devised ad

hoc to describe the proposed algorithm.

William Lee Irwin III wrote:
>> is what the
>> scheduler is trying to minimize;

On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
> This isn't always the case. Some may prefer fairness to minimal lag.
> Others may prefer particular tasks to receive preferential treatment.

This comment does not apply. Generalized processor sharing expresses
preferential treatment via weighting. Various other forms of
preferential treatment require more elaborate idealized models.

>> load is not directly relevant, but
>> appears to have some sort of relationship. Also, instead of pinned,
>> unpinned should be considered.

On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
> If you have total and pinned you can get unpinned. It's probably
> cheaper to maintain data for pinned than unpinned as there's less of it
> on normal systems.

Regardless of the underlying accounting, I've presented a coherent
algorithm. It may be that there's no demonstrable problem to solve.
On the other hand, if there really is a question as to how to load
balance in the presence of tasks pinned to cpus, I just answered it.

William Lee Irwin III wrote:
>> Using the signed minimax pseudonorm (i.e. the highest
>> signed lag, where positive is higher than all negative regardless of
>> magnitude) on unpinned lags yields a rather natural load balancing
>> algorithm consisting of migrating from highest to lowest signed lag,
>> with progressively longer periods for periodic balancing across
>> progressively higher levels of hierarchy in sched_domains etc. as usual.
>> Basically skip over pinned tasks as far as lag goes.
>> The trick with all that comes when tasks are pinned within a set of
>> cpus (especially crossing sched_domains) instead of to a single cpu.

On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
> Yes, this makes the cost of maintaining the required data higher which
> makes keeping pinned data more attractive than unpinned.
> BTW keeping data for sets of CPU affinities could cause problems as the
> number of possible sets is quite large (being 2 to the power of the
> number of CPUs). So you need an algorithm based on pinned data for

> single CPUs that knows the pinning isn't necessarily exclusive rather
> than one based on sets of CPUs. As I understand it (which may be
> wrong), the mechanism you describe below takes that approach.

Yes, the mechanism I described takes that approach.

William Lee Irwin III wrote:
>> The smpnice affair is better phrased in terms of task weighting. It's
>> simple to honor nice in such an arrangement. First unravel the
>> grouping hierarchy, then weight by nice. This looks like
[...]
>> In such a manner nice numbers obey the principle of least surprise.

On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
> Is it just me or did you stray from the topic of handling cpu affinity
> during load balancing to hierarchical load balancing? I couldn't see
> anything in the above explanation that would improve the handling of cpu
> affinity.

There was a second issue raised to which I responded. I didn't stray
per se. I addressed a second topic in the post.

-- wli

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Peter Williams on Wed, 30 May 2007 00:09:28 GMT
View Forum Message <> Reply to Message

William Lee Irwin III wrote:
> William Lee Irwin III wrote:
>>> Lag should be considered in lieu of load because lag
>
> On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
>> What's the definition of lag here?
>
> Lag is the deviation of a task's allocated CPU time from the CPU time
> it would be granted by the ideal fair scheduling algorithm (generalized
> processor sharing; take the limit of RR with per-task timeslices
> proportional to load weight as the scale factor approaches zero).

Over what time period does this operate?

> Negative lag reflects receipt of excess CPU time. A close-to-canonical
> "fairness metric" is the maximum of the absolute values of the lags of
> all the tasks on the system. The "signed minimax pseudonorm" is the
> largest lag without taking absolute values; it's a term I devised ad
> hoc to describe the proposed algorithm.

So what you're saying is that you think dynamic priority (or its
equivalent) should be used for load balancing instead of static priority?

>
> William Lee Irwin III wrote:
>>> is what the
>>> scheduler is trying to minimize;
>
> On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
>> This isn't always the case. Some may prefer fairness to minimal lag.
>> Others may prefer particular tasks to receive preferential treatment.
>
> This comment does not apply. Generalized processor sharing expresses
> preferential treatment via weighting. Various other forms of
> preferential treatment require more elaborate idealized models.

This was said before I realized that your "lag" is just a measure of
fairness.

>
>
>>> load is not directly relevant, but
>>> appears to have some sort of relationship. Also, instead of pinned,
>>> unpinned should be considered.
>
> On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
>> If you have total and pinned you can get unpinned. It's probably
>> cheaper to maintain data for pinned than unpinned as there's less of it
>> on normal systems.
>
> Regardless of the underlying accounting,

I was just replying to your criticism of my suggestion to keep pinned
task statistics and use them.

> I've presented a coherent
> algorithm. It may be that there's no demonstrable problem to solve.
> On the other hand, if there really is a question as to how to load
> balance in the presence of tasks pinned to cpus, I just answered it.

Unless I missed something there's nothing in your suggestion that does

anything more about handling pinned tasks than is already done by the
load balancer.

>
>
> William Lee Irwin III wrote:
>>> Using the signed minimax pseudonorm (i.e. the highest
>>> signed lag, where positive is higher than all negative regardless of
>>> magnitude) on unpinned lags yields a rather natural load balancing
>>> algorithm consisting of migrating from highest to lowest signed lag,
>>> with progressively longer periods for periodic balancing across
>>> progressively higher levels of hierarchy in sched_domains etc. as usual.
>>> Basically skip over pinned tasks as far as lag goes.
>>> The trick with all that comes when tasks are pinned within a set of
>>> cpus (especially crossing sched_domains) instead of to a single cpu.
>
> On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
>> Yes, this makes the cost of maintaining the required data higher which
>> makes keeping pinned data more attractive than unpinned.
>> BTW keeping data for sets of CPU affinities could cause problems as the
>> number of possible sets is quite large (being 2 to the power of the
>> number of CPUs). So you need an algorithm based on pinned data for
>> single CPUs that knows the pinning isn't necessarily exclusive rather
>> than one based on sets of CPUs. As I understand it (which may be
>> wrong), the mechanism you describe below takes that approach.
>
> Yes, the mechanism I described takes that approach.
>
>
> William Lee Irwin III wrote:
>>> The smpnice affair is better phrased in terms of task weighting. It's
>>> simple to honor nice in such an arrangement. First unravel the
>>> grouping hierarchy, then weight by nice. This looks like
> [...]
>>> In such a manner nice numbers obey the principle of least surprise.
>
> On Sun, May 27, 2007 at 11:29:51AM +1000, Peter Williams wrote:
>> Is it just me or did you stray from the topic of handling cpu affinity
>> during load balancing to hierarchical load balancing? I couldn't see
>> anything in the above explanation that would improve the handling of cpu
>> affinity.
>
> There was a second issue raised to which I responded. I didn't stray
> per se. I addressed a second topic in the post.

OK.

To reiterate, I don't think that my suggestion is really necessary. I

think that the current load balancing (stand fast a small bug that's
being investigated) will come up with a good distribution of tasks to
CPUs within the constraints imposed by any CPU affinity settings.

Peter
--
Peter Williams pwil3058@bigpond.net.au

"Learning, n. The kind of ignorance distinguishing the studious."
-- Ambrose Bierce

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

---

Subject: Re: [ckrm-tech] [RFC] [PATCH 0/3] Add group fairness to CFS
Posted by Srivatsa Vaddagiri on Wed, 30 May 2007 17:14:05 GMT
View Forum Message <> Reply to Message

On Sat, May 26, 2007 at 08:41:12AM -0700, William Lee Irwin III wrote:
> The smpnice affair is better phrased in terms of task weighting. It's
> simple to honor nice in such an arrangement. First unravel the
> grouping hierarchy, then weight by nice. This looks like
>
> task nice hier1 hier2 ... hierN
> t_1 w_n1 w_h11 w_h21 ... w_hN1
> t_2 w_n2 w_h12 w_h22 ... w_hN2
> ...
>
> For the example of nice 0 vs. nice 10 as distinct users with 10%
> steppings between nice levels, one would have
>
> task nice hier1
> t_1 1 1
> t_2 0.3855 1
>
> w_1, the weight of t_1, would be
> (w_h11*w_n1/(w_h11*w_n1 + w_h12*w_n2))
> = (1*1/(1 + 1*0.3855..))
> = 0.7217..
> w_2, the weight of t_2, would be
> (w_h12*w_n2/(w_h11*w_n1 + w_h12*w_n2))
> = (1*0.3855../(1 + 1*0.3855..))
> = 0.27826..
> This just so happens to work out to being the same as if t_1 and t_2
> had their respective nice numbers without the scheduler grouping, which
> is basically what everyone wants to happen.

>
> It's more obvious how to extend it to more tasks than levels of
> hierarchy. An example of that follows:
>
> task nice hier1 hier2 ... hierN
> t_1 0.3 0.6 * ... *
> t_2 0.7 0.4 * ... *
>
> hier2 through hierN are ignorable since t_1 and t_2 are both the only
> members at those levels of hierarchy. We then get something just like
> the above example, w_1 = 0.3*0.6/(0.3*0.6+0.7*0.4) = 0.3913.. and
> w2 = 0.7*0.4/(0.3*0.6+0.7*0.4) = 0.6087..
>
> It's more interesting with enough tasks to have more meaningful levels
> of hierarchy.
>
> task nice hier1 hier2
> t_1 0.7 0.6 0.6
> t_2 0.3 0.6 0.4
> t_3 0.7 0.4 0.6
> t_4 0.3 0.4 0.4
>
> where t_1 and t_2 share a hier1 grouping and t_3 and t_4 also share
> a hier1 grouping, but the hier1 grouping for t_1 and t_2 is distinct
> from the hier1 grouping for t_3 and t_4. All hier2 groupings are
> distinct. So t_1 would have pre-nice weight 0.6*0.6, t_2 0.6*0.4,
> t_3 0.6*0.4, and t_4 0.4*0.4 (the numbers were chosen so denominators
> conveniently collapse to 1). Now that the hierarchy is flattened,
> nice numbers can be factored in for t_1's final weight being
> 0.7*0.36/(0.7*0.36+0.3*0.24+0.7*0.24+0.3*0.16) = 0.252/0.54 = 0.467..
> and the others being 0.133.. (t_2), 0.311.. (t_3), and 0.0889.. (t_4).

Hmm ..so do you think this weight decomposition can be used to flatten
the tree all the way to a single level in case of cfs? That would mean we can
achieve group fairness with single level scheduling in cfs ..I am
somewhat skeptical that we can achieve group fairness with a single
level rb-tree (and w/o substantial changes to pick_next_task logic in cfs
that is), but if it can be accomplished would definitely be a great win.

> In such a manner nice numbers obey the principle of least surprise.

--
Regards,
vatsa