Improving "global" scheduler decisions

Paul Turner <pjt@google.com>
Overview

- Some CPU scheduling fundamentals
- Challenges
- Results
Linux uses the *Completely Fair Scheduler (CFS)*

**History & Overview**

- Merged in 2.6.23, replaces previous $O(1)$ scheduler.
- Weighted fair queuing scheduler; strong roots in where multiple packet flows must share a link.
- No "queues", uses red-black trees to track *timelines*. 
CFS: Basics

Basics

● "Weight based fair-scheduler"; allocate CPU cycles across period in proportion to each entity's weight.

How does this work in practice?

● Fix a unit period of time (the scheduling period $P$)
● Divide this period amongst tasks proportionally by weight
Basic Example

- 3 equivalent tasks A, B, C

Could choose: A, B, C

Or: C, B, A

Or even: A, B, C, B, A, B, C...

... Not even going to try and draw this one
CFS: Weight-based scheduling

More generally:

\[
\sum time(A) = \sum time(B) = \sum time(C) = \frac{P}{3}
\]

Note: \(P\) is \(~25\)ms on most systems

**But**, we assumed everyone had equal weight. **Hmm.**
CFS: Weight-based scheduling

Previous example assumed weights were uniform, how do we handle asymmetric weights?

By virtualizing time.
CFS: Virtual time

How do we fold weight into time?

Moderate its advancement.

For smaller entities
Time accumulates more quickly.

For larger entities
Vice versa, time accumulates more slowly.
CFS: Hierarchical scheduling

CFS supports the collection of tasks into a group, these groups can be nested to form a hierarchy.

Scheduling decision becomes recursive.
CFS: Timelines

For a **smaller** entity, virtual time proceeds more quickly

![Diagram for smaller entity]

For a **unit** entity, virtual time proceeds normally

![Diagram for unit entity]

For a **larger entity**, virtual time proceeds more slowly

![Diagram for larger entity]
How is vtime (virtual time) defined?

Linear scale:

\[ v_{time} = \frac{u}{w} \cdot time = \frac{1024}{w} \cdot time \]

e.g. Consider 5 elapsed seconds at weight=512

\[ v_{time} = \frac{1024}{512} \cdot 5s = 2 \cdot 5s = 10s \]

Note: "Unit" weight is 1024
CFS: Virtual time

Recall:

\[ \sum \text{time}(A) = \sum \text{time}(B) = \sum \text{time}(C) = \frac{P}{3} \]

Becomes:

\[ \sum v_{\text{time}}(A) = \sum v_{\text{time}}(B) = \sum v_{\text{time}}(C) = \frac{P}{3} \]
CFS: Timelines

As mentioned before, CFS maintains a timeline of all entities, ordered by vruntime. This is represented as a red-black tree.
CFS: Wake-up placement

Introduction of a new entity:

![Diagram of wake-up placement](image)

Placed here
CFS: Pre-emption

Also based on timeline
Scheduling Latency

What is scheduling latency?

Two cases we care about:

- Latency of wake-ups
- Round-robin latency
SMP: Group scheduling

Consider the previous hierarchical scheduling example.
CFS: Hierarchical scheduling

Example

1. Using **root** time line, pick **B**
2. **B** is a group entity, recurse.
3. Pick **T** from **B**'s virtual timeline.
4. **T** is a task, we're finished!
SMP ... makes everything harder.

Turns out scaling frequency is hard.

**Solution:** Scale parallelism! Many cores!

This adds tangles to everything we just talked about. :(
Problem:

The pre-emption decision is inconsistent. Had we chosen to run on CPU0, we would have pre-empted yet on CPU1 we are forced to wait.

Which of these is right?

We'll come back to this.
SMP: Group scheduling

The problem, more generally:

Group entities participate in more than one timeline.

- What weight do we assign each?
- How does the lag of one affect another?
- What does pre-emption between groups look like?
SMP-Group: Weight distribution

Group entities have a weight. But this is a global weight, their entities need a local weight when participating on each CPU's timeline.

Can't we just use the global weight?

Breaks under asymmetric competition :(
Suppose $A$ has 3 tasks of equal weight:


Note: $A[i]$ is the entity for group $A$ on cpu $i$.

Then,

$A[0]$ should be weighted at $2/3$ of $A$.
$A[1]$ should be weighted at $1/3$ of $A$.

We call the weight assigned to a group-entity its "shares".
SMP-Group: Shares distribution

Generalizing this:

\[ A[0]_{weight} = A_{shares} \cdot \frac{load_0}{\sum load_n} \]

But,
This is hard to compute.

- Sum(load_n) is O(n)!
- One load changing affects everyones' weight.
- Haven't even nested groups under groups here!
Shares: Initial approach

- Periodically evaluate this sum explicitly
  - Compute $\text{Sum}(\text{load}_n)$
  - Cache and divide each $\text{load}_i$ against this.

Previously accounted in the top 20 of all CPU cycles (by C/C++ function) consumed at Google.
Key idea
Load varies, instead of tracking the instaneous sum, let's track the average observed load and assign weights against that.
Shares: Current approach

**Average vs Instantaneous load within a period**

- **Instaneous**
- **Average**
Shares: Average history

Then,

Average everything together (with exponential decay)

\[
\text{load}_{A[0]} = \text{load}_0 + \frac{\text{load}_1}{2^1} + \frac{\text{load}_2}{2^2} \ldots
\]
Shares: Using average history

Used today, works fairly well... but..

**Caveat:**
No good way of accounting for load migrated due to load-balancing.

**Other pitfalls:**
Ratios versus current contribution are inconsistent.
Shares: Improving tracking

Each (per cpu) group entity tracks the average sum of its child load.

=> Can't determine a child's load contribution when moving it to another cpu!

Revised
What if each entity tracked its own runnable contribution? A group entities load would then be the sum of its childrens' contributions.
So why didn't we do this in the first place?

Hard to get right!
- We don't hold the right locks around wake-ups
- Hard to update sleeping entities
- Higher overheads
Shares: Tracking at the entity level

Instead of tracking the average of children, now tracking a contribution to parent.
Re-thinking shares averaging
Re-thinking shares averaging
Shares: Tracking at the entity level

How do we compute an entity's contribution?

\[ A[0]_{load} = \text{load}_0 \cdot y^0 + \text{load}_1 \cdot y^1 + \text{load}_2 \cdot y^2 + \ldots \]

Then normalize against period:

\[ A[0]_{period} = \sum p \cdot y^i \]

Finally:

\[ A[0]_{contrib} = \frac{A[0]_{contrib}}{A[0]_{period}} \]
Shares: Updating blocked entities

Still a problem
How do we handle updates against blocked entities?

Previously:

\[ A[0]_{\text{load}} = \sum load_i \cdot y^i \]

But, if idle, \( \text{load}_0 = 0 \)! So..

\[ A[0]'_{\text{load}} = \sum load_i \cdot y^{i+1} = y \cdot A[0]_{\text{load}} \]
Separate the sums maintained on a group entity into *runnable* and *blocked*.

The *runnable* sum is updated by the *active* entities making the contribution.

The *blocked* sum is updated periodically, using the previous decay trick.
What does this get us?
Load tracking: New

New tracking: 'avg' vs ideal (80% of 1024)
Well..

That wasn't very exciting.

But wait, what about the axes, let's overlay the two.
Load tracking: New vs Old

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Avg</th>
<th>Stdde</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>760</td>
<td>983</td>
<td>828</td>
<td>827</td>
<td>27.3</td>
</tr>
<tr>
<td>new</td>
<td>610</td>
<td>1878</td>
<td>1097</td>
<td>1070</td>
<td>183.5</td>
</tr>
</tbody>
</table>
Increasing the old shares window

Load tracking: New vs Old

- ideal
- new
- old, 10ms
- old, 20ms
- old, 50ms
Re-thinking shares averaging

Average vs Instantaneous load within a period

- Blue: Instantaneous
- Green: Average
Problem:

Still don't have an answer as to which choice was right!

Possibly worse: Nothing we've covered lets you tune this behavior.
Timeline Spread

Suppose \( \{A, B, C\} \) have equal weight

When we move \( B \) we preserve lag relative to \( A \).

But \( C \) should have **negative lag** relative to both \( A \) and \( B \)!
Handling "global" pre-emption?

The root of the problem is that we are using separate entities to track a single object.

Idea: Could we use a single (global) entity tree to track groups relative to one another?

Pitfall: Convergence of the spread within a scheduling level depends on only one entity being able to accumulate run-time.

In the absence of this restriction we are unable to bound latencies or have entities join the tree.
Timeline Spread

CFS latencies are implicitly bounded by vruntime spread:
Take #2

Idea:
Use bandwidth control style tracking of used run-time.

Pitfalls:
- We still want to be work-conserving. (easy)
- We need decay to be continuous... Discrete tracking of accumulated run-time will NOT result in consistent behavior. (really hard)
Take #3

Idea:
Treat group entities as the average behavior of their per-cpu entities.

Pitfalls:
• We need the averages to be accurate / up-to-date.
• May have problems if the distributions are uniformly "odd"
• We need to avoid starvation.
Lag is the difference between the time that an entity has received and the proportion its weight entitles it to.

\[ \text{lag}_i = S_i - s_i \]

Where:
- \( S_i \) is the ideal time by weight
- \( s_i \) is the actual received time.
Virtual Time: Lag

Positional comparison (wake-up) on time-line is actually trying to approximate lag delta using local information.

Instead use the global information to re-approximate this as part of placement. Wake-ups happen as before, but with a globally lag preserving placement scheme instead of a local one.
Results

Synthetic latency test (latt)
Results: Synthetic latency

Baseline

Whoops! Looks like we've got something to fix!
Results: Synthetic latency

New load tracking, 40% utilized
Results: Synthetic latency

New load tracking, 80% utilized
Results: Synthetic latency

Using global lag for entity placement, 40%
Results: Synthetic latency

Using global lag for entity placement, 80%
Results: Synthetic latency

Tail latencies

80% utilization: Max latency

- base-u80
- newload-u80
- global-u80
Results

OLTP vs Antagonists
Results: OLTP

Baseline

OLTP Baseline Latency

- avg-base
- global-avg
- 95th-base
- 95th-idle-lag
Results: OLTP

Baseline vs 40% antagonist

OLTP - Baseline vs 40% antagonist

- avg-base
- avg-u40-ba...
- 95th-base
- 95th-u40-b...
Results: OLTP

Baseline vs 80% antagonist

![Graph showing OLTP Baseline vs 80% Antagonist with lines for avg-base, avg-u80-base, 95th-base, and 95th-u80-base](image)
Results: OLTP

Global-lag w/ 40% vs Baseline

![Graph showing OLTP Global lag: Baseline vs 40% Antagonist](image-url)
Results: OLTP

Global-lag w/ 80% vs baseline
Results: OLTP

Global-lag w/ 40% vs baseline w/ 40%

OLTP Latency vs 40% Antagonist
Results: In group thread lags

Google RPC latency benchmark

Tail latency improved from ~55.4ms to ~48.5ms
What's next?

- Publish/merge load tracking patches
- Continue evaluating latency performance
- Some local fairness evaluations needed
Thanks for attending LPC 2011!

Further questions?
pjt@google.com