The OS scheduler: a performance-critical component in Linux cluster environments

Keynote for BPOE-9 @ASPLOS2018
The Ninth Workshop on Big Data Benchmarks, Performance, Optimization and Emerging Hardware

By Jean-Pierre Lozi
Oracle Labs
CLUSTER COMPUTING

- Multicore servers with dozens of cores
  - Common for e.g., a hadoop cluster, a distributed graph analytics engine, multiple apps...
  - High cost of infrastructure, high energy consumption
CLUSTER COMPUTING

- Multicore servers with dozens of cores
  - Common for e.g., a hadoop cluster, a distributed graph analytics engine, multiple apps...
  - High cost of infrastructure, high energy consumption

- Linux-based software stack
  - Low (license) cost, yet high reliability
CLUSTER COMPUTING

- Multicore servers with dozens of cores
  - Common for e.g., a hadoop cluster, a distributed graph analytics engine, multiple apps...
  - High cost of infrastructure, high energy consumption

- Linux-based software stack
  - Low (license) cost, yet high reliability

- Challenge: don’t waste cycles!
  - Reduces infrastructure and energy costs
  - Improves bandwidth and latency
WHERE TO HUNT FOR CYCLES?
WHERE TO HUNT FOR CYCLES?

Applications, libraries: often main focus
WHERE TO HUNT FOR CYCLES?

Applications, libraries: often main focus

Storage: optimized since decades! E.g., many filesystems, RDBMSes bypassing the OS
WHERE TO HUNT FOR CYCLES?

Applications, libraries: often main focus

Storage: optimized since decades! E.g., many filesystems, RDBMSes bypassing the OS

Network stack, NICs, reducing network usage (e.g. HDFS): common optimizations
WHERE TO HUNT FOR CYCLES?

Applications, libraries: often main focus

Storage: optimized since decades! E.g., many filesystems, RDBMSes bypassing the OS

Numerical Memory Access (NUMA), bus usage:
Placement, replication, interleaving, many recent papers

Network stack, NICs, reducing network usage (e.g. HDFS): common optimizations
WHERE TO HUNT FOR CYCLES?

Applications, libraries: often main focus

Storage: optimized since decades! E.g., many filesystems, RDBMSes bypassing the OS

Network stack, NICs, reducing network usage (e.g. HDFS): common optimizations

The scheduler??? Generally trusted!

NUMA, bus usage: Placement, replication, interleaving, many recent papers
IS THE SCHEDULER WORKING IN YOUR CLUSTER?
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:

  “And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:
  
  “And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:

> “And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.

- But would you notice if some cores remained idle intermittently, when they shouldn’t?
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:

  “And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.

- But would you notice if some cores remained idle intermittently, when they shouldn’t?
- **Do you keep monitoring tools (htop) running all the time?**
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:

  “And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.

- But would you notice if some cores remained idle intermittently, when they shouldn’t?
  - **Do you keep monitoring tools (htop) running all the time?**
  - **Even if you do, would you be able to identify faulty behavior from normal noise?**
IS THE SCHEDULER WORKING IN YOUR CLUSTER?

- It must be! 15 years ago, Linus Torvalds was already saying:
  
  “And you have to realize that there are not very many things that have aged as well as the scheduler. **Which is just another proof that scheduling is easy.**”

- Since then, people have been running applications on their multicore machines all the time, and they run, CPU usage is high, everything seems fine.

- But would you notice if some cores remained idle intermittently, when they shouldn’t?
  - **Do you keep monitoring tools (htop) running all the time?**
  - Even if you do, would you be able to identify faulty behavior from normal noise?
  - **Would you ever suspect the scheduler?**

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.
**THIS TALK**

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

- **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

**An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

**Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.

**Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn’t find anything that looked off.
THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

- **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.

- **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn’t find anything that looked off.

- **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.
THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn’t find anything that looked off.

  - **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.

- **We ended up suspecting the core behavior of the scheduler.**
THIS TALK

- Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

- **An example:** running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.
  - **Memory locality issue?** Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.
  - **Contention over some resource (spinlock, etc.)?** We investigated this for a long time, but couldn’t find anything that looked off.
  - **Overhead of context switches?** Threads moved a lot but we proved that the overhead was negligible.

- **We ended up suspecting the core behavior of the scheduler.**
  - *We implemented high-resolution tracing tools and saw that some cores were idle while others overloaded.*
Over the past few years of working on various projects, we sometimes saw strange, hard to explain performance results.

An example: running a TPC-H benchmark on a 64-core machine, our runs much faster when pinning threads to cores than when we let the Linux scheduler do its job.

Memory locality issue? Impossible, hardware counters showed no difference in the % of remote memory accesses, in cache misses, etc.

Contention over some resource (spinlock, etc.)? We investigated this for a long time, but couldn't find anything that looked off.

Overhead of context switches? Threads moved a lot but we proved that the overhead was negligible.

We ended up suspecting the core behavior of the scheduler.

We implemented high-resolution tracing tools and saw that some cores were idle while others overloaded...

![Diagram showing number of threads in run queue and core behavior.](image)
This is how we found our first performance bug. Which made us investigate more...
This is how we found our first performance bug. Which made us investigate more...

In the end: four Linux scheduler performance bugs that we found and analyzed.
THIS TALK

- This is how we found our first performance bug. Which made us investigate more...
- In the end: four Linux scheduler performance bugs that we found and analyzed
- Always the same symptom: idle cores while others are overloaded
- The bug-hunting was tough, and led us to develop our own tools
This talk

- This is how we found our first performance bug. Which made us investigate more...
- In the end: four Linux scheduler performance bugs that we found and analyzed
- Always the same symptom: idle cores while others are overloaded
  - The bug-hunting was tough, and led us to develop our own tools
- Performance overhead of some of the bugs:
  - 12-23% performance improvement on a popular database with TPC-H
  - 137× performance improvement on HPC workloads
This is how we found our first performance bug. Which made us investigate more...

In the end: four Linux scheduler performance bugs that we found and analyzed

Always the same symptom: idle cores while others are overloaded
  - The bug-hunting was tough, and led us to develop our own tools

Performance overhead of some of the bugs:
  - 12-23% performance improvement on a popular database with TPC-H
  - $137 \times$ performance improvement on HPC workloads

Not always possible to provide a simple, working fix...
  - Intrisic problems with the design of the scheduler?
Main takeaway of our analysis: more research must be directed towards implementing an efficient scheduler for multicore architectures, because contrary to what a lot of us think, this is *not* a solved problem!
**Main takeaway of our analysis:** more research must be directed towards implementing an efficient scheduler for multicore architectures, because contrary to what a lot of us think, this is *not* a solved problem!

Need convincing? Let’s go through it together...
Main takeaway of our analysis: more research must be directed towards implementing an efficient scheduler for multicore architectures, because contrary to what a lot of us think, this is *not* a solved problem!

Need convincing? Let’s go through it together...

...starting with a bit of background...
Main takeaway of our analysis: more research must be directed towards implementing an efficient scheduler for multicore architectures, because contrary to what a lot of us think, this is *not* a solved problem!

Need convincing? Let’s go through it together...

...starting with a bit of background...
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by runtime.
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by *runtime*

When a thread is done running for its *timeslice*: enqueued again

---

Core 0
Core 1
Core 2
Core 3

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by runtime

When a thread is done running for its timeslice, enqueued again

Some tasks have a lower niceness and thus have a longer timeslice (allowed to run longer)
**THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT**

One runqueue where threads are globally sorted by *runtime*

Cores get their next task from the global runqueue

When a thread is done running for its *timeslice*: enqueued again

Some tasks have a lower *niceness* and thus have a longer *timeslice* (allowed to run longer)

---

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by runtime

Cores get their next task from the global runqueue

Of course, cannot work with a single runqueue because of contention

When a thread is done running for its timeslice: enqueued again

Some tasks have a lower niceness and thus have a longer timeslice (allowed to run longer)

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS: IN PRACTICE

- One runqueue per core to avoid contention

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS: IN PRACTICE

- One runqueue per core to avoid contention
- CFS periodically balances “loads”:

\[
\text{load(task)} = \text{weight}^1 \times \% \text{ cpu use}^2
\]

\(^1\)The lower the niceness, the higher the weight
CFS: IN PRACTICE

- One runqueue per core to avoid contention
- CFS periodically balances “loads”:

\[
\text{load}(\text{task}) = \text{weight}^1 \times \% \text{ cpu use}^2
\]

1 The lower the niceness, the higher the weight
2 We don’t want a high-priority thread that sleeps a lot to take a whole CPU for itself and then mostly sleep!
CFS: IN PRACTICE

- One runqueue per core to avoid contention
- CFS periodically balances “loads”:

\[
\text{load(task)} = \text{weight}^1 \times \% \text{ cpu use}^2
\]

1 The lower the niceness, the higher the weight
2 We don’t want a high-priority thread that sleeps a lot to take a whole CPU for itself and then mostly sleep!

- Since there can be many cores: hierarchical approach!

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

Core 0
L=2000
L=1000

Core 1
L=1000
L=1000
L=3000

Core 2
L=6000
L=1000
L=1000
L=1000
L=1000
L=1000

Core 3
L=1000

Balanced!

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

The OS Scheduler: A Performance-Critical Component in Linux Cluster Environments
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

AVG(L)=2500

L=2000

L=3000

L=3000

L=4000

L=3000

AVG(L)=3500

Core 0

Core 1

Core 2

Core 3

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

AVG(L)=3000

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

**AVG(L)=3000**

**L=3000**

**Balanced!**

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered.
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level.
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered.
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level.

```
<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=0</td>
<td>L=6000</td>
<td>L=3000</td>
<td>L=3000</td>
</tr>
<tr>
<td>L=3000</td>
<td>L=100</td>
<td>L=1000</td>
<td>L=1000</td>
</tr>
<tr>
<td>L=1000</td>
<td>L=1000</td>
<td>L=1000</td>
<td>L=1000</td>
</tr>
</tbody>
</table>
```

AVG(L)=3000
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered.
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level.

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered.
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level.

![Diagram showing load balancing in Linux cluster environments.](image-url)

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

- Note that only the average load of groups is considered.
- If for some reason the lower-level load-balancing fails, nothing happens at a higher level.

![Diagram showing hierarchical load balancing]

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.
- **Objective**: making sure that launching lots of threads from one terminal doesn’t prevent other processes on the machine (potentially from other users) from running.
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.
- **Objective:** making sure that launching lots of threads from one terminal doesn’t prevent other processes on the machine (potentially from other users) from running.
  - Otherwise, easy to use more resources than other users by spawning many threads...
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.

```
L=1000
L=1000
L=1000
L=1000
L=1000
L=1000
L=1000
L=1000
```

Session (tty) 1

Session (tty) 2

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS 13
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.

Session (tty) 1
50% of a CPU 😞

Session (tty) 2
150% 😊

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.

50% of a CPU

Unfair!
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its tty...
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its "tty"...
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its `tty`...

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- Solution: divide the load of a task by the number of threads in its tty...
CFS IN PRACTICE: MORE HEURISTICS

- Load calculations are actually more complicated, use more heuristics.
- **Solution:** divide the load of a task by the number of threads in its tty...

**Wait, does that work?**
BUG 1/4: GROUP IMBALANCE

Session (tty) 1

Session (tty) 2

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
**BUG 1/4: GROUP IMBALANCE**

\[
\text{Load(thread)} = \frac{\%cpu \times \text{weight}}{\#\text{threads}}
\]

\[
= 100 \times 10 \quad / \quad 1
\]

\[
= 1000
\]

Session (tty) 1

\[
\text{Load(thread)} = \frac{\%cpu \times \text{weight}}{\#\text{threads}}
\]

\[
= 100 \times 10 \quad / \quad 8
\]

\[
= 125
\]

Session (tty) 2

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
**BUG 1/4: GROUP IMBALANCE**

Session (tty) 1

```
L=125
L=125
L=125
L=125
```

Load(thread) \[= \frac{\%cpu \times weight}{\#threads}\]

\[= 100 \times 10 / 1\]

\[= 1000\]

Session (tty) 2

```
L=125
L=125
L=125
L=125
```

Load(thread) \[= \frac{\%cpu \times weight}{\#threads}\]

\[= 100 \times 10 / 8\]

\[= 125\]

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
BUG 1/4: GROUP IMBALANCE

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
Bug 1/4: Group Imbalance

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
**BUG 1/4: GROUP IMBALANCE**

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
BUG 1/4: GROUP IMBALANCE

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
BUG 1/4: GROUP IMBALANCE

The OS Scheduler: A Performance-Critical Component in Linux Cluster Environments
**BUG 1/4: GROUP IMBALANCE**

Balanced!
BUG 1/4: GROUP IMBALANCE

AVG(L)=500

L=0

Balanced!

L=1000

L=0

Balanced!

L=1000

AVG(L)=500

L=500

Balanced!

L=500

L=1000

L=125

L=125

L=125

L=125

L=125

L=125

L=125

L=125

L=125

L=125

L=125

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
BUG 1/4: GROUP IMBALANCE

AVG(L)=500

L=0 Balanced!
L=1000 Balanced!
L=500 Balanced!
L=500 Balanced!

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
BUG 1/4: GROUP IMBALANCE

![Diagram showing Core 0, Core 1, Core 2, Core 3 with varying load (L) values and AVG(L) = 500 for balanced and imbalanced cases.]

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
BUG 1/4: GROUP IMBALANCE

AVG(L) = 500

Balanced!

L = 0

Balanced!

L = 1000

Balanced!

L = 500

Balanced!

L = 500

The OS Scheduler: A performance-critical component in Linux cluster environments
Bug 1/4: Group Imbalance

- Another example, on a 64-core machine, with load balancing:
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes
BUG 1/4: GROUP IMBALANCE

- Another example, on a 64-core machine, with load balancing:
- First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
- Then between NUMA nodes

- User 1 launches:
  ssh <machine> R &
  ssh <machine> R &
BUG 1/4: GROUP IMBALANCE

- Another example, on a 64-core machine, with load balancing:
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes

- **User 1 launches**:
  - `ssh <machine> R &`
  - `ssh <machine> R &`

- **User 2 launches**:
  - `ssh <machine> make -j 64 kernel`
BUG 1/4: GROUP IMBALANCE

- Another example, on a 64-core machine, with load balancing:
  - First between pairs of cores (Bulldozer architecture, a bit like hyperthreading)
  - Then between NUMA nodes

- **User 1 launches**:
  ssh <machine> R &
  ssh <machine> R &

- **User 2 launches**:
  ssh <machine> make -j 64 kernel

- **The bug happens at two levels**:
  - Other core on pair of core idle
  - Other cores on NUMA node less busy...
The bug happens at two levels:
- Other core on pair of core idle
- Other cores on NUMA node less busy
- The bug happens at two levels:
  - Other core on pair of core idle
  - Other cores on NUMA node less busy...
A simple solution: balance the minimum load of groups instead of the average.
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the minimum load of groups instead of the average
A simple solution: balance the minimum load of groups instead of the average.
A simple solution: balance the *minimum load* of groups instead of the average.
A simple solution: balance the minimum load of groups instead of the average.
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the *minimum load* of groups instead of the average

---

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS

---

L = 1000

Core 0

Core 1

L = 500

Core 2

Core 3
A simple solution: balance the minimum load of groups instead of the average

\[
\begin{align*}
\text{MIN}(L) &= 0 \\
\text{MIN}(L) &= 500
\end{align*}
\]
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the minimum load of groups instead of the average

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
A simple solution: balance the minimum load of groups instead of the average load.

\[ \text{MIN}(L) = 250 \]

Balanced!
A simple solution: balance the minimum load of groups instead of the average

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
A simple solution: balance the *minimum load* of groups instead of the average.

The minimum load is 250 units.
A simple solution: balance the **minimum load** of groups instead of the average.

\[
\begin{align*}
\text{MIN}(L) &= 250 \\
L &= 250 \\
L &= 1000 \\
\end{align*}
\]

**Core 0**

**Core 1**

**Core 2**

**Core 3**

\[
\begin{align*}
L &= 250 \\
L &= 500 \\
\end{align*}
\]
A simple solution: balance the minimum load of groups instead of the average load.

**MIN(L) = 250**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=250</td>
<td>L=1000</td>
</tr>
<tr>
<td>L=125</td>
<td>L=125</td>
</tr>
</tbody>
</table>

Balanced!

**MIN(L) = 250**

<table>
<thead>
<tr>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=250</td>
<td>L=500</td>
</tr>
<tr>
<td>L=125</td>
<td>L=125</td>
</tr>
<tr>
<td>L=125</td>
<td>L=125</td>
</tr>
<tr>
<td>L=125</td>
<td>L=125</td>
</tr>
</tbody>
</table>
A simple solution: balance the minimum load of groups instead of the average load.

**Example:**

- **Minimum Load (MIN(L)) = 250**
  - Core 0: $L=250$
  - Core 1: $L=1000$
  - Balanced

- **Minimum Load (MIN(L)) = 325**
  - Core 2: $L=325$
  - Core 3: $L=325$

---

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
A simple solution: balance the *minimum load* of groups instead of the average.

**MIN(L)=250**

- Core 0: L=250
- Core 1: L=1000

**Balanced!**

**MIN(L)=325**

- Core 2: L=325
- Core 3: L=325

**Balanced!**
A simple solution: balance the minimum load of groups instead of the average.

**MIN(L)=250**

L=250  
L=125  
L=125  

Balanced!

**MIN(L)=325**

L=325  
L=125  
L=125  

Balanced!
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the minimum load of groups instead of the average

![Diagram showing load balancing]

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
**BUG 1/4: GROUP IMBALANCE**

- A simple solution: balance the *minimum load* of groups instead of the *average*.
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the minimum load of groups instead of the average

- After the fix, make runs 13% faster, and R is not impacted
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the *minimum load* of groups instead of the average

- After the fix, make runs 13% faster, and R is not impacted

- A simple solution, but is it ideal? Minimum load more volatile than average...
BUG 1/4: GROUP IMBALANCE

- A simple solution: balance the minimum load of groups instead of the average
- After the fix, make runs 13% faster, and R is not impacted
- A simple solution, but is it ideal? Minimum load more volatile than average...
  - May cause lots of unnecessary rebalancing. Revamping load calculations needed?
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- Hierarchical load balancing is based on groups of cores named scheduling domains.
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- **Hierarchical load balancing** is based on groups of cores named *scheduling domains*
- Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.
Hierarchical load balancing is based on groups of cores named *scheduling domains*. Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc. Each scheduling domain contains groups that are the lower-level scheduling domains.
Hierarchical load balancing is based on groups of cores named scheduling domains. Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc. Each scheduling domain contains groups that are the lower-level scheduling domains. For instance, on our 64-core AMD Bulldozer machine:
Hierarchical load balancing is based on groups of cores named scheduling domains. Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.

Each scheduling domain contains groups that are the lower-level scheduling domains.

For instance, on our 64-core AMD Bulldozer machine:

At level 1, each pair of core (scheduling domains) contain cores (scheduling groups).
**BUG 2/4: SCHEDULING GROUP CONSTRUCTION**

- **Hierarchical load balancing** is based on groups of cores named *scheduling domains*
  - Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.

- Each scheduling domain contains groups that are the lower-level scheduling domains

- **For instance, on our 64-core AMD Bulldozer machine:**
  - At level 1, each *pair of core* (scheduling domains) contain *cores* (scheduling groups)
  - At level 2, each *CPU* (s.d.) contain *pairs of cores* (s.g.)
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- **Hierarchical load balancing** is based on groups of cores named *scheduling domains*
  - Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc.
  - Each scheduling domain contains groups that are the lower-level scheduling domains

- **For instance, on our 64-core AMD Bulldozer machine:**
  - At level 1, each pair of core (scheduling domains) contain cores (scheduling groups)
  - At level 2, each CPU (s.d.) contain pairs of cores (s.g.)
  - At level 3, each group of directly connected CPUs (s.d.) contain CPUs (s.g.)
Hierarchical load balancing is based on groups of cores named scheduling domains. Based on affinity, i.e., pairs of cores, dies, CPUs, NUMA nodes, etc. Each scheduling domain contains groups that are the lower-level scheduling domains.

For instance, on our 64-core AMD Bulldozer machine:
- At level 1, each pair of core (scheduling domains) contain cores (scheduling groups).
- At level 2, each CPU (s.d.) contain pairs of cores (s.g.)
- At level 3, each group of directly connected CPUs (s.d.) contain CPUs (s.g.)
- At level 4, the whole machine (s.d.) contains group of directly connected CPUs (s.g.)
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

Bulldozer 64-core: Eight CPUs, with 8 cores each, non-complete interconnect graph!
At the first level, the first core balances load with the other core on the same pair (because they share resources, high affinity).
At the 2\textsuperscript{nd} level, the \textit{first pair} balances load with other pairs on the same CPU.
At the 3rd level, the first CPU balances load with directly connected CPUs.
At the 4th level, the first group of directly connected CPUs balances load with the other groups of directly connected CPUs.
Groups of CPUs built by:

(1) picking first CPU and looking for all directly connected CPUs
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

Groups of CPUs built by:

(2) picking first CPU not in a group and looking for all directly connected CPUs
And then stop, because all CPUs are in a group.
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

And then stop, because all CPUs are in a group.

Wait, does that work?
Suppose we taskset an application on these two CPUs, two hops apart (16 threads)
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

And threads are created on this core.

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
Load gets correctly balanced on the pair of cores.
Load gets correctly balanced on the CPU
No stealing at level 3, because nodes not directly connected (1 hop apart)
At level 4, stealing between the red and green groups...

Overloaded node in both groups!
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

load(red) = 16 * load(thread)

load(green) = 16 * load(thread)
**BUG 2/4: SCHEDULING GROUP CONSTRUCTION**

**Load (red)** = $16 \cdot \text{load(thread)}$

**Load (green)** = $16 \cdot \text{load(thread)}$

Load is "balanced": nothing happens

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
Bug 2/4: Scheduling Group Construction

Load (red) = 16 * load (thread)

Load (green) = 16 * load (thread)

Load is "balanced": nothing happens
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

load(red) = 16 * load(thread)

load(green) = 16 * load(thread)

Fundamental issue with the scheduling hierarchy!
Bug 2/4: Scheduling Group Construction

- Fix: build the domains by creating one “directly connected” group for every CPU
- Instead of the first CPU and the first one not “covered” by a group
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

- Fix: build the domains by creating one “directly connected” group for every CPU
  - Instead of the first CPU and the first one not “covered” by a group

- Performance improvement of NAS applications on two nodes:

<table>
<thead>
<tr>
<th>Application</th>
<th>With bug</th>
<th>After fix</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>99</td>
<td>56</td>
<td>1.75x</td>
</tr>
<tr>
<td>CG</td>
<td>42</td>
<td>15</td>
<td>2.73x</td>
</tr>
<tr>
<td>EP</td>
<td>73</td>
<td>36</td>
<td>2x</td>
</tr>
<tr>
<td>LU</td>
<td>1040</td>
<td>38</td>
<td>27x</td>
</tr>
</tbody>
</table>
**BUG 2/4: SCHEDULING GROUP CONSTRUCTION**

- **Fix:** build the domains by creating one “directly connected” group for every CPU
- Instead of the first CPU and the first one not “covered” by a group
- Performance improvement of NAS applications on two nodes:

<table>
<thead>
<tr>
<th>Application</th>
<th>With bug</th>
<th>After fix</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>99</td>
<td>56</td>
<td>1.75x</td>
</tr>
<tr>
<td>CG</td>
<td>42</td>
<td>15</td>
<td>2.73x</td>
</tr>
<tr>
<td>EP</td>
<td>73</td>
<td>36</td>
<td>2x</td>
</tr>
<tr>
<td>LU</td>
<td>1040</td>
<td>38</td>
<td>27x</td>
</tr>
</tbody>
</table>

- **Very good improvement for LU because more threads than cores if can’t use 16 cores**
- Solves spinlock issues (incl. potential convoys)
BUG 3/4: MISSING SCHEDULING DOMAINS

- In addition to this, when domains re-built, levels 3 and 4 not re-built...
In addition to this, when domains re-built, levels 3 and 4 not re-built...

I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
BUG 3/4: MISSING SCHEDULING DOMAINS

- In addition to this, when domains re-built, levels 3 and 4 not re-built...
- I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core
In addition to this, when domains re-built, **levels 3 and 4 not re-built**...

- I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core

- **Launch an application, first thread created on CPU 1**
In addition to this, when domains re-built, levels 3 and 4 not re-built...

- I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core

Launch an application, first thread created on CPU 1
- First thread will stay on CPU 1, next threads will be created on CPU 1 (default Linux)
**BUG 3/4: MISSING SCHEDULING DOMAINS**

- In addition to this, when domains re-built, **levels 3 and 4 not re-built**...
- I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core

**Launch an application, first thread created on CPU 1**
- First thread will stay on CPU 1, next threads will be created on CPU 1 (default Linux)
- All the threads will be on CPU 1 forever!
**BUG 3/4: MISSING SCHEDULING DOMAINS**

- In addition to this, when domains re-built, **levels 3 and 4 not re-built**...
- I.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core

- **Launch an application, first thread created on CPU 1**
  - First thread will stay on CPU 1, next threads will be created on CPU 1 (default Linux)
  - All the threads will be on CPU 1 forever!
## Bug 3/4: Missing Scheduling Domains

- In addition to this, when domains re-built, **levels 3 and 4 not re-built**...
- i.e., no balancing between directly connected or 1-hop CPUs (i.e. any CPU)
- Happens for instance when disabling and re-enabling a core

### Launch an application, first thread created on CPU 1

- First thread will stay on CPU 1, next threads will be created on CPU 1 (default Linux)
- All the threads will be on CPU 1 forever!

<table>
<thead>
<tr>
<th>Application</th>
<th>With bug</th>
<th>After fix</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>122</td>
<td>23</td>
<td>5.2x</td>
</tr>
<tr>
<td>CG</td>
<td>134</td>
<td>5.4</td>
<td>25x</td>
</tr>
<tr>
<td>EP</td>
<td>72</td>
<td>18</td>
<td>4x</td>
</tr>
<tr>
<td>LU</td>
<td>2196</td>
<td>16</td>
<td>137x</td>
</tr>
</tbody>
</table>

### Table: Application Performance Improvement

<table>
<thead>
<tr>
<th>Application</th>
<th>With bug</th>
<th>After fix</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>122</td>
<td>23</td>
<td>5.2x</td>
</tr>
<tr>
<td>CG</td>
<td>134</td>
<td>5.4</td>
<td>25x</td>
</tr>
<tr>
<td>EP</td>
<td>72</td>
<td>18</td>
<td>4x</td>
</tr>
<tr>
<td>LU</td>
<td>2196</td>
<td>16</td>
<td>137x</td>
</tr>
</tbody>
</table>

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
Until now, we analyzed the behavior of the periodic, (buggy) hierarchical load balancing that uses (buggy) scheduling domains.
Bug 4/4: Overload-on-wake up

- Until now, we analyzed the behavior of the periodic, (buggy) hierarchical load balancing that uses (buggy) scheduling domains

- But there is another way load is balanced: threads get to pick on which core they get woken up when they are done blocking (after a lock acquisition, an I/O)...

The OS Scheduler: A Performance-Critical Component in Linux Cluster Environments
Until now, we analyzed the behavior of the periodic, *(buggy)* hierarchical load balancing that uses *(buggy)* scheduling domains

But there is another way load is balanced: threads get to pick on which core they get woken up when they are done blocking (after a lock acquisition, an I/O)...

Here is how it works: when a thread wakes up, it looks for non-busy cores on the same CPU in order to decide on which core it should wake up.
Until now, we analyzed the behavior of the the periodic, *(buggy)* hierarchical load balancing that uses *(buggy)* scheduling domains.

But there is another way load is balanced: threads get to pick on which core they get woken up when they are done blocking (after a lock acquisition, an I/O)... 

Here is how it works: when a thread wakes up, it looks for non-busy cores on the same CPU in order to decide on which core it should wake up.

Only cores that are on the same CPU, in order to improve data locality...
BUG 4/4: OVERLOAD-ON-WAKEUP

- Until now, we analyzed the behavior of the periodic, (buggy) hierarchical load balancing that uses (buggy) scheduling domains

- But there is another way load is balanced: threads get to pick on which core they get woken up when they are done blocking (after a lock acquisition, an I/O)...

- Here is how it works: when a thread wakes up, it looks for non-busy cores on the same CPU in order to decide on which core it should wake up.

- Only cores that are on the same CPU, in order to improve data locality...

Wait, does that work?
BUG 4/4: OVERLOAD-ON-WAKEUP

- Commercial DB with TPC-H, 64 threads on 64 cores, nothing else on the machine.
Bug 4/4: Overload-on-wakeup

- Commercial DB with TPC-H, 64 threads on 64 cores, nothing else on the machine.
- With threads pinned to cores, works fine. With Linux scheduling, execution much slower, phases with overloaded cores while there are long-term idle cores!
Commercial DB with TPC-H, 64 threads on 64 cores, nothing else on the machine.

With threads pinned to cores, works fine. **With Linux scheduling, execution much slower, phases with overloaded cores while there are long-term idle cores!**

Number of threads in run queue: 0 1 2 3

Extra thread moves across cores (from periodic or idle rebalancing)

Extra thread back on idle core

Overloaded core (#15)

Idle core (#13)

Slowed down execution
Bug 4/4: Overload-on-wake-up

- Commercial DB with TPC-H, 64 threads on 64 cores, nothing else on the machine.
- With threads pinned to cores, works fine. With Linux scheduling, execution much slower, phases with overloaded cores while there are long-term idle cores!

Number of threads in run queue: 0 1 2 3

What is happening?

Slowed down execution
 Beginning: 8 threads / CPU, cores busy
BUG 4/4

- Beginning: 8 threads / CPU, cores busy

- Occasionally, **1 DB thread migrated to other CPU** because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
BEGINNING: 8 THREADS / CPU, CORES BUSY

- Occasionally, 1 DB THREAD MIGRATED TO OTHER CPU because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)

NOW, 9 THREADS ON ONE CPU, AND 7 ON ANOTHER ONE. CPU WITH 9 THREADS SLOW, SLOWS DOWN ALL EXECUTION BECAUSE ALL THREADS WAIT FOR EACH OTHER (BARRIERS), I.E. IDLE CORES EVERYWHERE...
BUG 4/4

- Beginning: 8 threads / CPU, cores busy
- Occasionally, 1 DB thread migrated to other CPU because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- Now, 9 threads on one CPU, and 7 on another one. CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...
- Barriers: threads keep sleeping and waking up, but extra thread never wakes up on idle core, because waking up algorithm only considers local CPU!
Beginning: 8 threads / CPU, cores busy

Occasionally, 1 DB thread migrated to other CPU because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)

Now, 9 threads on one CPU, and 7 on another one. CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...

Barriers: threads keep sleeping and waking up, but extra thread never wakes up on idle core, because waking up algorithm only considers local CPU!

Periodic rebalancing can’t rebalance load most of the time because many idle cores ⇒ Hard to see an imbalance between 9-thread and 7-thread CPU...
### BUG 4/4

- **Beginning:** 8 threads / CPU, cores busy
- **Occasionally,** 1 DB thread migrated to other CPU because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- **Now,** 9 threads on one CPU, and 7 on another one. CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...
- **Barriers:** threads keep sleeping and waking up, but extra thread never wakes up on idle core, because waking up algorithm only considers local CPU!
- **Periodic rebalancing can’t rebalance load most of the time because many idle cores**
  - Hard to see an imbalance between 9-thread and 7-thread CPU...
- **“Solution”:** wake up on core idle for the longest time (not great for energy)

---

**THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS**
BUG 4/4

- Beginning: 8 threads / CPU, cores busy
- Occasionally, 1 DB thread migrated to other CPU because transient thread appeared during rebalancing which looked like imbalance (only instant loads considered)
- Now, 9 threads on one CPU, and 7 on another one. CPU with 9 threads slow, slows down all execution because all threads wait for each other (barriers), i.e. idle cores everywhere...
- Barriers: threads keep sleeping and waking up, but extra thread never wakes up on idle core, because waking up algorithm only considers local CPU!
- Periodic rebalancing can’t rebalance load most of the time because many idle cores ⇒ Hard to see an imbalance between 9-thread and 7-thread CPU...
- “Solution”: wake up on core idle for the longest time (not great for energy)

<table>
<thead>
<tr>
<th>Bug fixes</th>
<th>TPC-H request #18</th>
<th>Full TPC-H benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>55.9s</td>
<td>542.9s</td>
</tr>
<tr>
<td>Group Imbalance</td>
<td>48.6s (−13.1%)</td>
<td>513.8s (−5.4%)</td>
</tr>
<tr>
<td>Overload-on-Wakeup</td>
<td>43.5s (−22.2%)</td>
<td>471.1s (−13.2%)</td>
</tr>
<tr>
<td>Both</td>
<td>43.3s (−22.6%)</td>
<td>465.6s (−14.2%)</td>
</tr>
</tbody>
</table>

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS

28
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem
- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named \textit{load},
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem.

- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named *load*,
  - to periodically balance threads between *scheduling domains*. 

---

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS  

29
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem

- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named load,

- to periodically balance threads between scheduling domains.

- In addition to this, threads balance load by selecting core where to wake up.
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem

**To recap, on Linux, load balancing works that way:**
- Hierarchical rebalancing uses a metric named *load*,
  
  **Found fundamental issue here**
  
  to periodically balance threads between *scheduling domains*.

- In addition to this, threads balance load by *selecting core where to wake up*. 

---

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS 29
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem

- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named *load*,
  - *Found fundamental issue here*
  - to periodically balance threads between *scheduling domains*.
  - *Found fundamental issue here*
  - In addition to this, threads balance load by *selecting core where to wake up*.
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem

- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named *load*,
  - to periodically balance threads between *scheduling domains*.
  - In addition to this, threads balance load by *selecting core where to wake up*.
WHERE DO WE GO FROM HERE?

- Load balancing on a multicore machine usually considered a solved problem
- To recap, on Linux, load balancing works that way:
  - Hierarchical rebalancing uses a metric named load,
  ↑ Found fundamental issue here
  - to periodically balance threads between scheduling domains.
  ↑ Found fundamental issue here
  - In addition to this, threads balance load by selecting core where to wake up.
  ↑ Found fundamental issue here

Wait, was anything working at all? 😊
WHERE DO WE GO FROM HERE?

Many major issues went unnoticed for years in the scheduler...

How can we prevent this from happening again?
WHERE DO WE GO FROM HERE?

Many major issues went unnoticed for years in the scheduler...

How can we prevent this from happening again?

- Code testing
  - No clear fault (no crash, no deadlock, etc.)
  - Existing tools don’t target these bugs
WHERE DO WE GO FROM HERE?

Many major issues went unnoticed for years in the scheduler...

How can we prevent this from happening again?

- Code testing
  - No clear fault (no crash, no deadlock, etc.)
  - Existing tools don’t target these bugs

- Performance regression
  - Usually done with 1 app on a machine to avoid interactions
  - Insufficient coverage
WHERE DO WE GO FROM HERE?

Many major issues went unnoticed for years in the scheduler...

How can we prevent this from happening again?

- **Code testing**
  - No clear fault (no crash, no deadlock, etc.)
  - Existing tools don’t target these bugs

- **Performance regression**
  - Usually done with 1 app on a machine to avoid interactions
  - Insufficient coverage

- **Model checking, formal proofs**
  - Complex, parallel code: so far, nobody knows how to do it...
WHERE DO WE GO FROM HERE?

- Idea 1: short-term hack — implemented a sanity checker
WHERE DO WE GO FROM HERE?

- **Idea 1:** short-term hack — implemented a sanity checker

---

Idle core while a core is overloaded?

Yes ➔ Monitor thread migrations, creations, destructions

Imbalance not fixed ➔ Report a bug

---

Every second
WHERE DO WE GO FROM HERE?

- **Idea 1:** short-term hack — implemented a sanity checker

```
Idle core while a core is overloaded?

Yes

Monitor thread migrations, creations, destructions

100ms

Imbalance not fixed

Report a bug
```

Not an assertion/watchdog: might not be a bug

Every second

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
WHERE DO WE GO FROM HERE?

- **Idea 1**: short-term hack — implemented a sanity checker

### Diagram:

1. **Idle core while a core is overloaded?**
   - Yes
   - Monitor thread migrations, creations, destructions
   - Imbalance not fixed
2. **Report a bug**
3. **Not an assertion/watchdog**: might not be a bug
   - situation has to last for a long time

- Every second

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
WHERE DO WE GO FROM HERE?

- **Idea 2:** fine-grained tracers!
WHERE DO WE GO FROM HERE?

- **Idea 2:** fine-grained tracers!
  - Built a simple one, turned out to be the only way to really understand what happens
  - Aggregate metrics (CPI, cache misses, etc.) not precise enough
WHERE DO WE GO FROM HERE?

- **Idea 2:** fine-grained tracers!
  - Built a simple one, turned out to be the only way to really understand what happens
  - Aggregate metrics (CPI, cache misses, etc.) not precise enough

- Could really be improved!
WHERE DO WE GO FROM HERE?

- **Idea 3:** produce a dedicated profiler!
WHERE DO WE GO FROM HERE?

- **Idea 3:** produce a dedicated profiler!
- Lack of tools!
WHERE DO WE GO FROM HERE?

- **Idea 3:** produce a dedicated profiler!
  - Lack of tools!
  - Possible to detect if slowdown comes from scheduler or application?
  - Would avoid a lot of wasted time!
WHERE DO WE GO FROM HERE?

- **Idea 3:** produce a dedicated profiler!
  - Lack of tools!

- Possible to detect if slowdown comes from scheduler or application?
- Would avoid a lot of wasted time!

- Follow threads, and see if often on overloaded cores when shouldn’t have?
- Detect if threads unnecessarily moved to core/node that leads to many cache misses?
WHERE DO WE GO FROM HERE?

- **Idea 4**: produce good scheduler benchmarks!
WHERE DO WE GO FROM HERE?

- **Idea 4:** produce good scheduler benchmarks!
  - Really needed, and virtually inexistent!
WHERE DO WE GO FROM HERE?

- **Idea 4**: produce good scheduler benchmarks!
  - Really needed, and virtually inexistent!

- **Not an easy problem**: insane coverage needed!
- **Using combination of many real applications**: configuration nightmare!
WHERE DO WE GO FROM HERE?

- **Idea 4:** produce good scheduler benchmarks!
  - Really needed, and virtually inexisten!

- **Not an easy problem:** insane coverage needed!
- **Using combination of many real applications:** configuration nightmare!

- Simulated workloads?
  - Have to do *elaborate work:* spinning and sleeping not efficient
  - Have to be *representative of reality,* have to *cover corner cases*
  - *Use machine learning? Genetic algorithms?*
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

  Let’s take a step back: *why* did we end up in this situation?
WHERE DO WE GO FROM HERE?

- **Idea 5**: switch to simpler schedulers, easier to reason about!

Let's take a step back: *why* did we end up in this situation?

- Linux used for many classes of applications (big data, n-tier, cloud, interactive, DB, HPC...)

THE OS SCHEDULER: A PERFORMANCE-CRITICAL COMPONENT IN LINUX CLUSTER ENVIRONMENTS
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

Let’s take a step back: *why* did we end up in this situation?

- Linux used for many classes of applications (big data, n-tier, cloud, interactive, DB, HPC...)
- Multicore architectures increasingly diverse and complex!
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

Let's take a step back: *why* did we end up in this situation?

- Linux used for many classes of applications (big data, n-tier, cloud, interactive, DB, HPC...)
- Multicore architectures increasingly diverse and complex!
- **Result:** very complex monolithic scheduler supposed to work in all situations!
  - Many heuristics interact in complex, unpredictable ways
  - Some features greatly complexify, e.g., load balancing (tasksets, cgroups/autogroups...)
WHERE DO WE GO FROM HERE?

- **Idea 5**: switch to simpler schedulers, easier to reason about!

Let’s take a step back: *why* did we end up in this situation?

- Linux used for many classes of applications (big data, n-tier, cloud, interactive, DB, HPC...)
- Multicore architectures increasingly diverse and complex!
- **Result**: very complex monolithic scheduler supposed to work in all situations!
  - Many heuristics interact in complex, unpredictable ways
  - Some features greatly complexify, e.g., load balancing (tasksets, cgroups/autogroups...)
- Keeps getting worse!
- **E.g., task_struct**: 163 fields in Linux 3.0 (07/2011), 215 fields in 4.6 (05/2016)
- **20,000 lines of C!**
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

- **Proving the scheduler implementation correct:** not doable!
  - Way too much code for current technology
  - We’d need to detect high-level abstractions from low-level C: a challenge!
  - Even if we managed that, how do we keep up with updates?
    - Code keeps evolving with new architectures and application needs...
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

- **Proving the scheduler implementation correct: not doable!**
  - Way too much code for current technology
  - We’d need to detect high-level abstractions from low-level C: a challenge!
  - Even if we managed that, how do we keep up with updates?
    - Code keeps evolving with new architectures and application needs...

- **We need another approach...**
WHERE DO WE GO FROM HERE?

- **Idea 5**: switch to simpler schedulers, easier to reason about!

- Write simple, schedulers with proven properties!
  - A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion, load balancing criterion, state machine with events...
  - **Machine partitioned into sets of cores that run ≠ schedulers**
  - Scheduler deployed together with (an) application(s) on a partition
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

- Write simple, schedulers with proven properties!
  - A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion, load balancing criterion, state machine with events...
  - **Machine partitioned into sets of cores that run ≠ schedulers**
  - Scheduler deployed together with (an) application(s) on a partition

- **Through a DSL, for two reasons:**
  - Much easier, safer and less bug-prone than writing low-level C kernel code!
  - Clear abstractions, possible to reason about them and prove properties
    - Work conservation, load balancing live and in finite # or rounds, valid hierarchy...
WHERE DO WE GO FROM HERE?

- **Idea 5:** switch to simpler schedulers, easier to reason about!

- **Write simple, schedulers with proven properties!**
  - A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion, load balancing criterion, state machine with events...
  - **Machine partitioned into sets of cores that run ≠ schedulers**
  - Scheduler deployed together with (an) application(s) on a partition

- **Through a DSL, for two reasons:**
  - Much easier, safer and less bug-prone than writing low-level C kernel code!
  - Clear abstractions, possible to reason about them and prove properties
  - **Work conservation, load balancing live and in finite # or rounds, valid hierarchy...**
WHERE DO WE GO FROM HERE?

- **Idea 6:** ???
- Any other ideas?
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis:** fundamental issues in the load metric, scheduling domains, scheduling choices...
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis:** fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis:** fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
- Can’t ensure simple “invariant”: no idle cores while overloaded cores
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis**: fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
- **Can’t ensure simple “invariant”: no idle cores while overloaded cores**
- **Proposed fixes**: not always satisfactory
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis**: fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
- Can’t ensure simple “invariant”: no idle cores while overloaded cores
- **Proposed fixes**: not always satisfactory
- **What can we do?** Many things to explore!
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.
- **Analysis**: fundamental issues in the load metric, scheduling domains, scheduling choices...
- Very bug-prone implementation following years of adapting to hardware
- Can’t ensure simple “invariant”: no idle cores while overloaded cores
- **Proposed fixes**: not always satisfactory
- **What can we do?** Many things to explore!

**Our takeaway**: *more research must be directed towards implementing efficient and reliable schedulers for multicore architectures!*
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) was thought to be a solved problem.

- **Analysis:** fundamental issues in the load metric, scheduling domains, scheduling choices...

- Very bug-prone implementation following years of adapting to hardware

- Can’t ensure simple “invariant”: no idle cores while overloaded cores

- Proposed fixes: not always satisfactory

- What can we do? Many things to explore!

- **Our takeaway:** more research must be directed towards implementing efficient and reliable schedulers for multicore architectures!

*Your turn!*