THE LINUX SCHEDULER: A DECADE OF WASTED CORES

Jean-Pierre Lozi
jplozi@unice.fr

Baptiste Lepers
baptiste.lepers@epfl.ch

Fabien Gaud
me@fabiengaud.net

Université Nice Sophia Antipolis

Ècole Polytechnique Fédérale de Lausanne

COHO DATA

Alexandra Fedorova
sasha@ece.ubc.ca

Justin Funston
jfunston@ece.ubc.ca

Vivien Quéma
vivien.quema@imag.fr

THE LINUX SCHEDULER: A DECADE OF Wasted CORES

1/16
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)

- **Run two CPU-intensive processes in two terminals (e.g. R scripts):**
  ```
  R < script.R --nosave &
  R < script.R --nosave &
  ```
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)

- Run two CPU-intensive processes in two terminals (e.g. R scripts):
  R < script.R --nosave &
  R < script.R --nosave &

- Compile your kernel in a third terminal:
  make -j 62 kernel
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)
- Run two CPU-intensive processes in two terminals (e.g. R scripts):
  \[
  \text{R < script.R --nosave \\n  R < script.R --nosave}
  \]
- Compile your kernel in a third terminal:
  make -j 62 kernel
- Here is what might happen:
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)

- **Run two CPU-intensive processes in two terminals (e.g. R scripts):**
  
  ```
  R < script.R --nosave &
  R < script.R --nosave &
  ```

- Compile your kernel in a third terminal:
  ```
  make -j 62 kernel
  ```

- Here is what might happen:
  - Two NUMA nodes with many idle cores (white)
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)
- Run two CPU-intensive processes in two terminals (e.g. R scripts):
  R < script.R --nosave &
  R < script.R --nosave &
- Compile your kernel in a third terminal:
  make -j 62 kernel
- Here is what might happen:
  - Two NUMA nodes with many idle cores (white)
  - Other NUMA nodes with many overloaded cores (orange, red)
INTRODUCTION

- Take a machine with a lot of cores (64 in our case)
- Run two CPU-intensive processes in two terminals (e.g. R scripts):
  \[ R < \text{script.R} --nosave & \]
  \[ R < \text{script.R} --nosave & \]
- Compile your kernel in a third terminal:
  make -j 62 kernel
- Here is what might happen:
  - Two NUMA nodes with many idle cores (white)
  - Other NUMA nodes with many overloaded cores (orange, red)

Performance degradation: 14% for the make process!
INTRODUCTION

- General-purpose schedulers aim to be work-conserving on multicore architectures
INTRODUCTION

- General-purpose schedulers aim to be work-conserving on multicore architectures
- **Basic invariant:** no idle cores if some cores have several threads in their runqueues
  - Can actually happen, but only in transient situations!
INTRODUCTION

- General-purpose schedulers aim to be work-conserving on multicore architectures.
- **Basic invariant:** no idle cores if some cores have several threads in their runqueues.
- Can actually happen, but only in transient situations!

We found four major bugs that break this invariant in the Linux scheduler (CFS)!
INTRODUCTION

- General-purpose schedulers aim to be work-conserving on multicore architectures

- **Basic invariant:** no idle cores if some cores have several threads in their runqueues
  - Can actually happen, but only in transient situations!

  **We found four major bugs that break this invariant in the Linux scheduler (CFS)!**

- **This talk:** presentation of the CFS scheduler + issues we found + discussion
INTRODUCTION

- General-purpose schedulers aim to be work-conserving on multicore architectures.
- **Basic invariant:** no idle cores if some cores have several threads in their runqueues.
  - Can actually happen, but only in transient situations!
  - We found four major bugs that break this invariant in the Linux scheduler (CFS)!

- **This talk:** presentation of the CFS scheduler + issues we found + discussion.

---

**Disclaimer:** this is a motivation paper!
Don’t expect a solved problem 😊
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue, threads sorted by runtime
The Completely Fair Scheduler (CFS): Concept

One runqueue, threads sorted by runtime

When thread done running for its timeslice: enqueued again
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue, threads sorted by runtime

When thread done running for its timeslice: enqueued again

Lower niceness = longer timeslice (tasks allowed to run longer)
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue, threads sorted by *runtime*

Cores: next task from runqueue

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

Lower *niceness* = longer *timeslice* (tasks allowed to run longer)
One runqueue, threads sorted by runtime

Cores: next task from runqueue

In practice: cannot work with single runqueue because of contention!

When thread done running for its timeslice: enqueued again

Lower niceness = longer timeslice (tasks allowed to run longer)
CFS: IN PRACTICE

- One runqueue per core to avoid contention
CFS: IN PRACTICE

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

\[
\text{load(task)} = \text{weight}^1 \times \% \text{ cpu use}^2
\]
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 Lower niceness = higher 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. Lower niceness = higher weight
2. Prevent high-priority thread from taking whole CPU just to sleep
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. Lower niceness = higher weight
2. Prevent high-priority thread from taking whole CPU just to 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. Lower niceness = higher weight
2. Prevent high-priority thread from taking whole CPU just to sleep

- Since there can be many cores: hierarchical approach!
CFS: BALANCING THE LOAD

Core 0

Core 1

Core 2

Core 3

L=2000

L=3000

L=6000

L=1000
CFS: BALANCING THE LOAD

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

Core 0

L=2000

L=1000

L=1000

Core 1

L=3000

Core 2

L=6000

L=1000

L=1000

L=1000

L=1000

Balanced!

Core 3

L=1000

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

The Linux Scheduler: A Decade of Wasted Cores
CFS: BALANCING THE LOAD

Balanced!

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

AVG(L) = 2500

L = 2000

L = 3000

AVG(L) = 3500

L = 4000

L = 3000

Core 0

Core 1

Core 2

Core 3
CFS: BALANCING THE LOAD

\[ \text{AVG}(L) = 3000 \]

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

AVG(L)=3000

Balanced!

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
  - **Idea:** ensure a `tty` cannot eat up all resources by spawning many threads
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics.

- One of them aims to increase fairness between “sessions”
  - Idea: ensure a tty cannot eat up all resources by spawning many threads.

```
L=1000
```

Session (tty) 1

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

Session (tty) 2

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

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”
- Idea: ensure a tty cannot eat up all resources by spawning many threads.
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
  - Idea: ensure a tty cannot eat up all resources by spawning many threads

50% of a core 😠

150% 😎
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between “sessions”.
  - Idea: ensure a tty cannot eat up all resources by spawning many threads.

Session (tty) 1

50% of a core

Session (tty) 2

150%

Unfair!
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- **One of them aims to increase fairness between “sessions”**
  - **Solution:** divide the load of a task by the number of threads in its tty!
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
  - Solution: divide the load of a task by the number of threads in its tty!

<table>
<thead>
<tr>
<th>Session (tty) 1</th>
<th>L=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=250</td>
<td></td>
</tr>
<tr>
<td>L=250</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Session (tty) 2</th>
<th>L=250</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=250</td>
<td></td>
</tr>
<tr>
<td>L=250</td>
<td></td>
</tr>
</tbody>
</table>
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
- **Solution**: divide the load of a task by the number of threads in its tty!
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics
- One of them aims to increase fairness between “sessions”
- Solution: divide the load of a task by the number of threads in its tty!
CFS: BALANCING THE LOAD

- Load calculations are actually more complicated, use more heuristics.
- One of them aims to increase fairness between "sessions".
- **Solution:** divide the load of a task by the number of threads in its tty!

100% of a core 😊

Wait, does that work?
CFS: BALANCING THE LOAD: BUG #1

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD: BUG #1

The Linux Scheduler: A Decade of Wasted Cores
CFS: BALANCING THE LOAD: BUG #1

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
CFS: BALANCING THE LOAD: **BUG #1**

![Diagram showing balanced load distribution across cores](image-url)

**THE LINUX SCHEDULER: A DECADE OF WASTED CORES**
CFS: BALANCING THE LOAD: **BUG #1**

**Core 0**

**Core 1**

**Core 2**

**Core 3**

Balanced!
CFS: BALANCING THE LOAD: BUG #1

The Linux Scheduler: A Decade of Wasted Cores
CFS: BALANCING THE LOAD: BUG #1

AVG(L) = 500

L = 0  Balanced!  L = 1000

L = 1000

L = 0  Balanced!  L = 500

L = 500

L = 250

L = 250

L = 250

L = 250

Core 0

Core 1

Core 2

Core 3
CFS: BALANCING THE LOAD: BUG #1

AVG(L) = 500

Balanced!

L = 0  L = 1000  L = 500  L = 500

Core 0  Core 1  Core 2  Core 3

L = 1000  L = 250  L = 250  L = 250

Balanced!
CFS: BALANCING THE LOAD: **BUG #1**

AVG(L)=500  Balanced!  AVG(L)=500

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

Core 0  L=1000  Core 1  L=250  L=250  Core 2  L=250  L=250  Core 3
CFS: BALANCING THE LOAD: **BUG #1**

**The Linux Scheduler: A Decade of Wasted Cores**

**Fundamental issue with the load metric...**
CFS: BALANCING THE LOAD: **BUG #1**

- This was our bug!

Number of threads in run queue: 0 1 2 3 4+
CFS: BALANCING THE LOAD: **BUG #1**

- This was our bug!

![Diagram showing load distribution across NUMA nodes]
CFS: BALANCING THE LOAD: **BUG #1**

- This was our bug!

Load 1 = \( \text{avg}(R \text{ thread with high load} + \text{a few make threads with low load}) \)
CFS: BALANCING THE LOAD: **BUG #1**

- This was our bug!

Load 1 = \( \text{avg}(\text{R thread with high load} + \text{a few make threads with low load}) \)

Load 2 = \( \text{avg}(\text{many make threads with low load}) \)
This was our bug!

Load 1 = \text{avg}(R \text{ thread with high load + a few make threads with low load})

Load 2 = \text{avg}(\text{many make threads with low load})

Load 1 = \text{Load 2} : \text{the scheduler thinks the load is balanced!}
MORE BUGS: THE HIERARCHY

- We saw load balancing hierarchical: cores, pairs of cores, dies, CPUs, NUMA nodes...
MORE BUGS: THE HIERARCHY

- We saw load balancing hierarchical: cores, pairs of cores, dies, CPUs, NUMA nodes...
- Bug #2: on complex machines, hierarchy built incorrectly!
MORE BUGS: THE HIERARCHY

- We saw load balancing hierarchical: cores, pairs of cores, dies, CPUs, NUMA nodes...
- Bug #2: on complex machines, hierarchy built incorrectly!
- Intuition: at the last level, groups in the hierarchy “not disjoint”
MORE BUGS: THE HIERARCHY

- We saw load balancing hierarchical: cores, pairs of cores, dies, CPUs, NUMA nodes...
- Bug #2: on complex machines, hierarchy built incorrectly!

- Intuition: at the last level, groups in the hierarchy “not disjoint”

- Can break load balancing: whole application running on a single node!
MORE BUGS: THE HIERARCHY

- We saw load balancing hierarchical: cores, pairs of cores, dies, CPUs, NUMA nodes...
- Bug #2: on complex machines, hierarchy built incorrectly!
  - Intuition: at the last level, groups in the hierarchy “not disjoint”
  - Can break load balancing: whole application running on a single node!
- Bug #3: disabling/reenabling a core breaks the hierarchy completely
MORE BUGS: WAKEUPS

- Bug #4: slow phases with idle cores with popular commercial database + TPC-H

- Thread wake-up on a non-idle core

Bug: many idle cores!
MORE BUGS: WAKEUPS

- Bug #4: slow phases with idle cores with popular commercial database + TPC-H
- In addition to periodic load balancing, threads pick where they wake up

Thread wake-up on a non-idle core

Number of threads in run queue: 0 1 2 3

Bug: many idle cores!

Slowed down execution
MORE BUGS: WAKEUPS

- **Bug #4**: slow phases with idle cores with popular commercial database + TPC-H
  - In addition to periodic load balancing, threads pick where they wake up
  - Only local CPU cores considered for wakeup due to locality “optimization”

![](image)
MORE BUGS: WAKEUPS

- **Bug #4**: slow phases with idle cores with popular commercial database + TPC-H
  - In addition to periodic load balancing, threads pick where they wake up
  - Only local CPU cores considered for wakeup due to locality “optimization”
- **Intuition**: periodic load balancing global, wakeup balancing local

- Thread wake-up on a non-idle core

![Diagram showing many idle cores]

**Bug: many idle cores!**

Slowed down execution
MORE BUGS: WAKEUPS

- **Bug #4**: slow phases with idle cores with popular commercial database + TPC-H
- In addition to periodic load balancing, threads pick where they wake up
- Only local CPU cores considered for wakeup due to locality “optimization”
- **Intuition**: periodic load balancing global, wakeup balancing local
- **One makes mistakes the other cannot fix!**

Thread wake-up on a non-idle core

Number of threads in run queue: 0 1 2 3

Bug: many idle cores!
MORE BUGS: WAKEUPS

- Bug #4: slow phases with idle cores with popular commercial database + TPC-H
  - In addition to periodic load balancing, threads pick where they wake up
  - Only local CPU cores considered for wakeup due to locality “optimization”
- Intuition: periodic load balancing global, wakeup balancing local
- One makes mistakes the other cannot fix!

Performance degradation: 13-24%!

Bug: many idle cores!
DISCUSSION: HOW DID WE COME TO THIS?

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named *load*. 
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named *load*.
  - Threads among groups of cores in a *hierarchy*.
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named \textit{load},
  - threads among groups of cores in a \textit{hierarchy}.
  - In addition to this, threads \textit{balance the load by selecting core where to wake up}. 
DISCUSSION: HOW DID WE COME TO THIS?

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

- **To recap, on Linux, CFS works like this:**
  - It periodically balances, using a metric named `load`,
  - **Fundamental issue here...** appeared with `tty`-balancing heuristic for multithreaded apps.
  - threads among groups of cores in a `hierarchy`.

- In addition to this, threads **balance the load by selecting core where to wake up.**
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named load,
  - Fundamental issue here... appeared with tty-balancing heuristic for multithreaded apps
  - threads among groups of cores in a hierarchy.
  - Fundamental issue here... added with support of complex NUMA hierarchies
  - In addition to this, threads balance the load by selecting core where to wake up.
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named **load**,
  - **Fundamental issue here...** appeared with `tty`-balancing heuristic for multithreaded apps
  - threads among groups of cores in a **hierarchy**.
  - **Fundamental issue here...** added with support of complex NUMA hierarchies
  - In addition to this, threads **balance the load by selecting core where to wake up.**
  - **Fundamental issue here...** added with locality optimization for multicore architectures
DISCUSSION: HOW DID WE COME TO THIS?

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

- To recap, on Linux, CFS works like this:
  - It periodically balances, using a metric named load,
  - Fundamental issue here... appeared with tty-balancing heuristic for multithreaded apps
  - threads among groups of cores in a hierarchy.
  - Fundamental issue here... added with support of complex NUMA hierarchies
  - In addition to this, threads balance the load by selecting core where to wake up.
  - Fundamental issue here... added with locality optimization for multicore architectures

CFS was simple...

then became complex/broken when needed to support new hardware/uses!
DISCUSSION: WHERE DO WE GO FROM HERE?

- Linux scheduler keeps evolving, different algorithms, new heuristics...
- Hardware evolves fast, won’t get any better!
DISCUSSION: WHERE DO WE GO FROM HERE?

- Linux scheduler keeps evolving, different algorithms, new heuristics...
- Hardware evolves fast, won’t get any better!

We *need* a *safe* way to keep up with future hardware/uses!
DISCUSSION: WHERE DO WE GO FROM HERE?

- Linux scheduler keeps evolving, different algorithms, new heuristics...
  - Hardware evolves fast, won’t get any better!

We *need* a *safe* way to keep up with future hardware/uses!

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

- Linux scheduler keeps evolving, different algorithms, new heuristics...
  - Hardware evolves fast, won’t get any better!

We *need* a *safe* way to keep up with future hardware/uses!

- 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
DISCUSSION: WHERE DO WE GO FROM HERE?

- Linux scheduler keeps evolving, different algorithms, new heuristics...
  - Hardware evolves fast, won’t get any better!

We *need* a *safe* way to keep up with future hardware/uses!

- 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...
**DISCUSSION: WHERE DO WE GO FROM HERE?**

- **What worked for us:** *sanity checker* detects invariant violations to find bugs.
**DISCUSSION: WHERE DO WE GO FROM HERE?**

- **What worked for us:** *sanity checker* detects invariant violations to find bugs
- **Idea:** detect suspicious situations, monitor them and produce report if they last
DISCUSSION: WHERE DO WE GO FROM HERE?

- **What worked for us:** *sanity checker* detects invariant violations to find bugs
- **Idea:** detect suspicious situations, monitor them and produce report if they last
- **All bugs presented here detected with sanity checker!**
DISCUSSION: WHERE DO WE GO FROM HERE?

- **What worked for us:** *sanity checker* detects invariant violations to find bugs
- **Idea:** detect suspicious situations, monitor them and produce report if they last
- **All bugs presented here detected with sanity checker!**
- **Our experience:** exact traces are *necessary* to understand complex scheduling problems
DISCUSSION: WHERE DO WE GO FROM HERE?

- **What worked for us:** *sanity checker* detects invariant violations to find bugs
- **Idea:** detect suspicious situations, monitor them and produce report if they last
- **All bugs presented here detected with sanity checker!**
- **Our experience:** exact traces are *necessary* to understand complex scheduling problems
- Custom visual tool show all scheduling events / migrations / considered cores / load...

THE LINUX SCHEDULER: A DECADE OF WASTED CORES 14/16
DISCUSSION: FIXING THE SCHEDULER POSSIBLE?

- Basic fixes for the bugs we analyzed:
  - **Bug #1:** minimum load instead of average *(may be less stable!)*
  - **Bugs #2-#3:** building the hierarchy differently *(seems to always work!)*
  - **Bug #4:** wake up on cores idle for longest time *(may be bad for energy!)*
DISCUSSION: FIXING THE SCHEDULER POSSIBLE?

- Basic fixes for the bugs we analyzed:
  - **Bug #1**: minimum load instead of average *(may be less stable!)*
  - **Bugs #2-#3**: building the hierarchy differently *(seems to always work!)*
  - **Bug #4**: wake up on cores idle for longest time *(may be bad for energy!)*

- **Fixes not perfect, hard to ensure they never worsen performance**
  - Linux scheduler too complex, many competing heuristics added empirically!
  - Hard to guess the effect of one change...
DISCUSSION: FIXING THE SCHEDULER POSSIBLE?

- Basic fixes for the bugs we analyzed:
  - Bug #1: minimum load instead of average (may be less stable!)
  - Bugs #2-#3: building the hierarchy differently (seems to always work!)
  - Bug #4: wake up on cores idle for longest time (may be bad for energy!)

- Fixes not perfect, hard to ensure they never worsen performance
  - Linux scheduler too complex, many competing heuristics added empirically!
  - Hard to guess the effect of one change...

- Efficient redesign of the scheduler possible?
  - We envision scheduler with *isolated* modules each trying to optimize one variable...
DISCUSSION: FIXING THE SCHEDULER POSSIBLE?

- Basic fixes for the bugs we analyzed:
  - Bug #1: minimum load instead of average (may be less stable!)
  - Bugs #2-#3: building the hierarchy differently (seems to always work!)
  - Bug #4: wake up on cores idle for longest time (may be bad for energy!)

- Fixes not perfect, hard to ensure they never worsen performance
  - Linux scheduler too complex, many competing heuristics added empirically!
  - Hard to guess the effect of one change...

- Efficient redesign of the scheduler possible?
  - We envision scheduler with *isolated* modules each trying to optimize one variable...
  - How do you make them all work together? Complex, open problem!
CONCLUSION

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

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
- **Analysis:** fundamental issues (added incrementally), even basic invariant violated!
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
- **Analysis:** fundamental issues (added incrementally), even basic invariant violated!
- **Proposed pragmatic detection approach** *(sanity checker + traces)*: helpful
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
- **Analysis:** fundamental issues (added incrementally), even basic invariant violated!
- **Proposed pragmatic detection approach** *(sanity checker + traces):* helpful
- **Proposed fixes:** not always satisfactory
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
- Analysis: fundamental issues (added incrementally), even basic invariant violated!
- Proposed pragmatic detection approach (sanity checker + traces): helpful
- Proposed fixes: not always satisfactory

Open problem: how do we ensure the scheduler works/evolves correctly?

New design? New techniques involving testing/performance regression/proofs/...?
CONCLUSION

- Scheduling (as in dividing CPU cycles among threads) often thought to be a solved problem
- **Analysis:** fundamental issues (added incrementally), even basic invariant violated!
- **Proposed pragmatic detection approach** (**sanity checker** + traces): helpful
- **Proposed fixes:** not always satisfactory

**Open problem:** how do we ensure the scheduler works/evolves correctly ?

New design? New techniques involving testing/performance regression/proofs/...?

Your next paper 😊