Jean-Pierre Lozi
jplozi@unice.fr

Baptiste Lepers
baptiste.lepers@epfl.ch

Fabien Gaud
me@fabiengaud.net

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
IS THE SCHEDULER OF YOUR MACHINE WORKING?
IS THE SCHEDULER OF YOUR MACHINE WORKING?

- 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 OF YOUR MACHINE WORKING?

- 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.
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 OF YOUR MACHINE WORKING?

- 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 OF YOUR MACHINE WORKING?

- 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 OF YOUR MACHINE WORKING?

- 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?
THIS TALK

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

Contestion 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...
THIS TALK

- 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, analyzed and fixed

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

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

After fixing 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?
THIS TALK

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...
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by runtime

Core 0
Core 1
Core 2
Core 3

R = 103
R = 82
R = 24
R = 18
R = 12
One runqueue where threads are globally sorted by runtime.

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

Core 0
Core 1
Core 2
Core 3
THE COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by runtime

Threads 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 COMPLETELY FAIR SCHEDULER (CFS): CONCEPT

One runqueue where threads are globally sorted by *runtime*

Threads 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)
CFS: IN PRACTICE

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

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

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

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

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

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

  \(^1\)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!
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

L=2000

L=3000

L=6000

L=1000

Core 0

Core 1

Core 2

Core 3
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

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

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

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

The Linux Scheduler: A Decade of Wasted Cores
CFS IN PRACTICE: HIERARCHICAL LOAD BALANCING

AVG(L) = 2500

L = 2000

L = 3000

L = 4000

L = 3000

L = 1000
L = 1000
L = 1000
L = 1000
L = 1000
L = 1000

Core 0

Core 1

Core 2

Core 3

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

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

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

AVG(L) = 3000

L = 3000

Balanced!

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
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=100</td>
<td>L=100</td>
<td>L=100</td>
</tr>
<tr>
<td>L=6000</td>
<td>L=1000</td>
<td>L=1000</td>
<td>L=1000</td>
</tr>
<tr>
<td></td>
<td>L=3000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AVG(L)=3000

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
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.

![Diagram showing load balancing](image)
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:

```
L=3000
L=1000
L=1000
L=1000
L=3000
```

Core 0
Core 1
Core 2
Core 3

Balanced!

AVG(L)=3000
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”.

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
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”.

Session (tty) 1

L=1000  L=1000
L=1000  L=1000

Session (tty) 2

L=1000  L=1000
L=1000  L=1000
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

Session (tty) 2
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
L=1000
L=1000
L=1000
L=1000

Session (tty) 2
L=1000
L=1000
L=1000
L=1000

50% of a CPU 😞
150% 😊
```
CFS IN PRACTICE: MORE HEURISTICS

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

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...
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?

100% of a CPU 😊

100% of a CPU 😊

Session (tty) 1

Session (tty) 2

L=1000

L=250

L=250

L=250

L=250

L=250
BUG 1/4: GROUP IMBALANCE

Session (tty) 1

Session (tty) 2
BUG 1/4: GROUP IMBALANCE

Load(thread) = \%cpu × weight / #threads

Session (tty) 1

Load(thread) = 100 × 10 / 1
= 1000

Session (tty) 2

Load(thread) = 100 × 10 / 8
= 125
BUG 1/4: GROUP IMBALANCE

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

Session (tty) 1

\[ L = \frac{100 \times 10}{1} = 1000 \]

Session (tty) 2

\[ L = \frac{100 \times 10}{8} = 125 \]
BUG 1/4: GROUP IMBALANCE

L=0

L=1000

L=500

L=500

Core 0

Core 1

Core 2

Core 3
BUG 1/4: GROUP IMBALANCE

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
BUG 1/4: GROUP IMBALANCE

L=0  L=1000

Core 0  Core 1

L=500  L=500

Core 2  Core 3

L=125  L=125  L=125  L=125
L=125  L=125  L=125  L=125
L=125  L=125  L=125  L=125
BUG 1/4: GROUP IMBALANCE

THE LINUX SCHEDULER: A DECADE OF WASTED CORES

L=0
Balanced!

L=1000

L=500
L=500

L=125
L=125
L=125
L=125

Core 0
Core 1
Core 2
Core 3
BUG 1/4: GROUP IMBALANCE

L=0  L=1000  Balanced!

Core 0  Core 1

L=500  L=500

Core 2  Core 3

L=125  L=125  L=125  L=125  L=125  L=125  L=125  L=125
BUG 1/4: GROUP IMBALANCE

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
BUG 1/4: GROUP IMBALANCE

AVG(L)=500

L=0

Balanced!

L=1000

L=1000

L=0

AVG(L)=500

L=125

L=125

L=125

L=125

L=125

L=125

L=125

L=500

Balanced!

L=500

L=500

L=500

L=125

L=125

L=125

L=125

L=125

L=125

L=125

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
**BUG 1/4: GROUP IMBALANCE**

The Linux Scheduler: A Decade of Wasted Cores

AVG(L) = 500

- **Core 0**
  - L = 0
  - Balanced!

- **Core 1**
  - L = 1000
  - Balanced!

- **Core 2**
  - L = 500
  - Balanced!

- **Core 3**
  - L = 500
  - Balanced!
BUG 1/4: GROUP IMBALANCE

AVG(L)=500

Balanced!

L=0

L=1000

Balanced!

L=125
L=125
L=125
L=125

AVG(L)=500

Balanced!

L=125
L=125
L=125
L=125

Core 0

Core 1

Core 2

Core 3
**Bug 1/4: Group Imbalance**

The Linux Scheduler: A Decade of Wasted Cores

---

**Fundamental issue with the load metric...**
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.
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.

![Diagram showing load distribution across cores](image-url)
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.
## BUG 1/4: GROUP IMBALANCE

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

---

**Diagram:**

- **Core 0:**
  - Load: 0
  - Balanced!

- **Core 1:**
  - Load: 1000

- **Core 2:**
  - Load: 500
  - Balanced!

- **Core 3:**
  - Load: 500

---

**THE LINUX SCHEDULER: A DECADE OF WASTED CORES**
**BUG 1/4: GROUP IMBALANCE**

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

---

1. **L = 0**
   - MIN(L) = 0
   - Core 0

2. **L = 1000**
   - MIN(L) = 1000
   - Core 1

3. **L = 500**
   - MIN(L) = 500
   - Core 2

4. **L = 500**
   - MIN(L) = 500
   - Core 3

---

**THE LINUX SCHEDULER: A DECADE OF WASTED CORES**
A simple solution: balance the minimum load of groups instead of the average load.
A simple solution: balance the minimum load of groups instead of the average.

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

Balanced!
BUG 1/4: GROUP IMBALANCE

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

![Diagram showing load balancing]

L = 1000
MIN(L) = 250
L = 500
MIN(L) = 250
A simple solution: balance the minimum load of groups instead of the average

**THE LINUX SCHEDULER: A DECADE OF WASTED CORES**

**BUG 1/4: GROUP IMBALANCE**

- MIN(L) = 250
- L = 250
- L = 1000
- L = 125
- L = 125
- L = 125
- L = 125
- Core 0
- Core 1

- MIN(L) = 250
- L = 250
- L = 500
- L = 125
- L = 125
- Core 2
- Core 3
A simple solution: balance the *minimum load* of groups instead of the average.

**MIN(L) = 250**

- L = 1000
- L = 250
- Balanced!

**MIN(L) = 250**

- L = 500
- L = 250

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*.

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

\[
\text{MIN}(L) = 325
\]

---

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
BUG 1/4: GROUP IMBALANCE

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

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

**L = 1000**

**L = 125**

**L = 125**

**L = 125**

Balanced!

**L = 250**

**L = 325**

**L = 325**

**L = 325**

Balanced!

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
BUG 1/4: GROUP IMBALANCE

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

![Diagram showing balanced load distribution across cores]

The Linux Scheduler: A Decade of Wasted Cores
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.
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.
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:
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).
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.)
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.).
**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.)
  - At level 4, the *whole machine* (s.d.) contains *group of directly connected CPUs* (s.g.)
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 2nd level, the 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.
And then stop, because all CPUs are in a group.

Wait, does that work?
Suppose we taskset an application on these two nodes, two hops apart.
And threads are created on this core
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

Load gets correctly balanced on the pair of cores
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

Load gets correctly balanced on the CPU (8 threads)
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

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 is “balanced”: nothing happens

\[ \text{load(red)} = 16 \times \text{load(thread)} \]

\[ \text{load(green)} = 16 \times \text{load(thread)} \]
BUG 2/4: SCHEDULING GROUP CONSTRUCTION

\[
\text{load(red)} = 16 \times \text{load(thread)}
\]

\[
\text{load(green)} = 16 \times \text{load(thread)}
\]

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!

THE LINUX SCHEDULER: A DECADE OF WASTED CORES
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...
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)
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
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
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)
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!
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>
BUG 4/4: OVERLOAD-ON-WAKEUP

- Until now, we analyzed the behavior of the periodic, (buggy) hierarchical load balancing that uses (buggy) scheduling domains.
BUG 4/4: OVERLOAD-ON-WAKEUP

- 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)
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.
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...
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!*
**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!**

![Diagram showing overload and wake-up issues](image)

**Number of threads in run queue:** 0 1 2 3

- Overloaded core (#15)
- Idle core (#13)
- Extra thread moves across cores (from periodic or idle rebalancing)
- Extra thread back on idle core

**Slowed down execution**
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!

![Diagram showing thread distribution and core utilization]

- 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
- Slowed down execution

What is happening?
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...
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...

“Solution”: wake up on core idle for the longest time (not great for energy)
• 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>
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 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,

↑ Fundamental issue here
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,
  - Fundamental issue here
  - to periodically balance threads between scheduling domains.
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*,

↑ **Fundamental issue here**
- to periodically balance threads between *scheduling domains*.

↑ **Fundamental issue here**
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*.
  - Fundamental issue here
  - to periodically balance threads between *scheduling domains*.
  - 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 \textit{load},
  - to periodically balance threads between \textit{scheduling domains}.
  - In addition to this, threads balance load by \textit{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.

↑ Fundamental issue here

Wait, does anything work 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?

- A pragmatic “solution”: can’t prevent bugs, let’s detect them with a *sanity checker*
WHERE DO WE GO FROM HERE?

- **A pragmatic “solution”:** can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom:** some idle cores while others overloaded
WHERE DO WE GO FROM HERE?

- **A pragmatic “solution”:** can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom:** some idle cores while others overloaded

```
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?

- **A pragmatic “solution”:** can’t prevent bugs, let’s detect them with a *sanity checker*
- **Always same symptom: some idle cores while others overloaded**

```
100ms

Idle core while a core is overloaded?

Yes

Monitor thread migrations, creations, destructions

Imbalance not fixed

Report a bug
```

Not an assertion/watchdog: might not be a bug

Every second
WHERE DO WE GO FROM HERE?

- **A pragmatic “solution”:** can’t prevent bugs, let’s detect them with a **sanity checker**
- **Always same symptom:** some idle cores while others overloaded

---

**Diagram:**

1. **Idle core while a core is overloaded?**
   - **Yes**
   - **Every second**
   - **100ms**
   - **Monitor thread migrations, creations, destructions**
   - **Imbalance not fixed**
   - **Not an assertion/watchdog:** might not be a bug
   - **situation has to last for a long time**

2. **Report a bug**

---

*THE LINUX SCHEDULER: A DECADE OF WASTED CORES*
WHERE DO WE GO FROM HERE?

- We might miss some bugs. Not an issue, bugs that impact performance happen often.
WHERE DO WE GO FROM HERE?

- We might miss some bugs. Not an issue, bugs that impact performance happen often.
- We’ll eventually catch them.
WHERE DO WE GO FROM HERE?

- We might miss some bugs. Not an issue, bugs that impact performance happen often
- We’ll eventually catch them
- Low overhead, possible to reduce period (will just take longer to detect bugs)
WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We’ll eventually catch them**

- Low overhead, possible to reduce period (will just take longer to detect bugs)

- All bugs presented here detected with sanity checker
WHERE DO WE GO FROM HERE?

- **We might miss some bugs.** Not an issue, bugs that impact performance happen often
  - **We’ll eventually catch them**

- Low overhead, possible to reduce period (will just take longer to detect bugs)

- All bugs presented here detected with sanity checker

- Possible to replay bugs, and produce graphical traces to understand them better
WHERE DO WE GO FROM HERE?

- We might miss some bugs. Not an issue, bugs that impact performance happen often
- We’ll eventually catch them
- Low overhead, possible to reduce period (will just take longer to detect bugs)
- All bugs presented here detected with sanity checker
- Possible to replay bugs, and produce graphical traces to understand them better
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 theads) 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
- **Proposed pragmatic detection approach** ("sanity checker"): helpful
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
- **Proposed pragmatic detection approach ("sanity checker"):** helpful
- **Code testing, performance regression, model checking / proofs:** can’t work for now.
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
- **Proposed pragmatic detection approach (“sanity checker”):** helpful
- Code testing, performance regression, model checking / proofs: can’t work for now.

**Our takeaway:** *more research must be directed towards implementing an efficient and reliable scheduler 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
- **Proposed pragmatic detection approach** ("sanity checker"): helpful
- Code testing, performance regression, model checking / proofs: can’t work for now.

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