Ipanema:
Safe multicore scheduling in a Linux cluster environment

Jean-Pierre Lozi — *Université Nice Sophia-Antipolis*

with :

Gilles Muller, Julia Lawall — *UPMC/INRIA/LIP6 Paris*

Nicolas Palix — *Université Grenoble Alpes*

Baptiste Lepers, Willy Zwaenpoel — *EPFL*

Ipanema:
Safe multicore scheduling in a Linux cluster environ

Work in Progress!

Jean-Pierre Lozi — *Université Nice Sophia-Antipolis*

with :

Gilles Muller, Julia Lawall — *UPMC/INRIA/LIP6 Paris*

Nicolas Palix — *Université Grenoble Alpes*

Baptiste Lepers, Willy Zwaenpoel — *EPFL*

Context: cluster computing

- Focus on cluster computing

Context: cluster computing

- Focus on cluster computing
- Multicore servers with dozens of cores
  - High cost of infrastructure, high energy consumption

Context: cluster computing

- Focus on cluster computing
- Multicore servers with dozens of cores
  - High cost of infrastructure, high energy consumption
- Linux-based software stack
  - Low (license) cost, yet high reliability

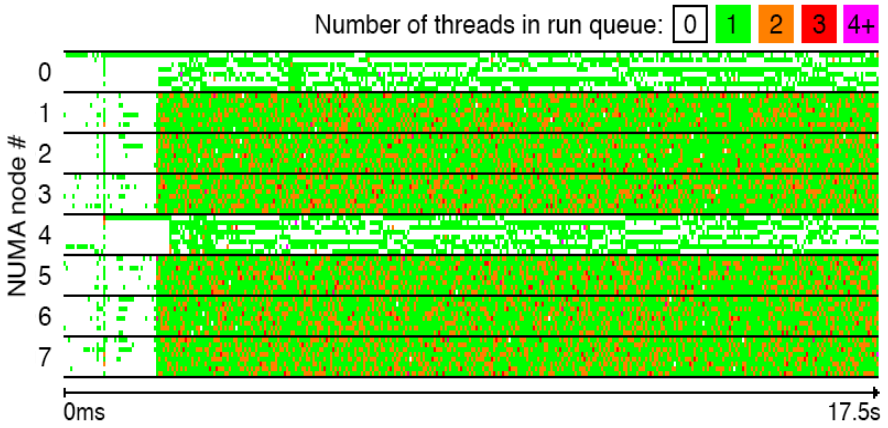
Context: cluster computing

- Focus on cluster computing
- Multicore servers with dozens of cores
  - 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

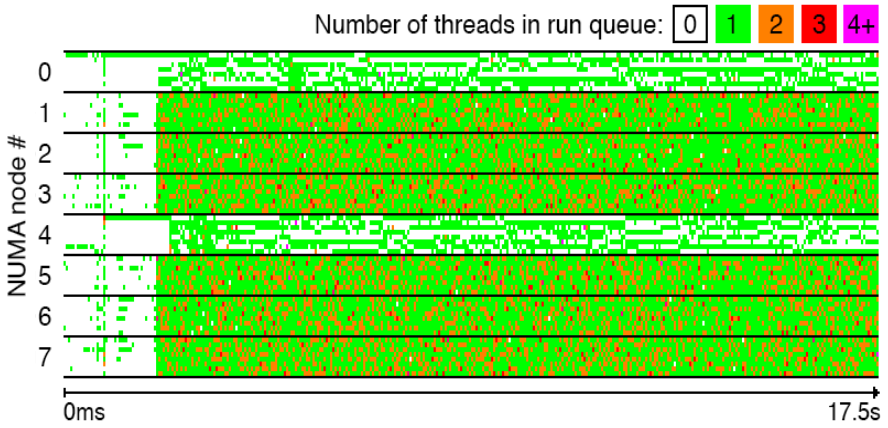
The problem: perf bugs in scheduler

- The Linux scheduler has performance bugs!
- Showed this last year @EuroSys
  « The Linux Scheduler: A Decade of Wasted Cores »



The problem: perf bugs in scheduler

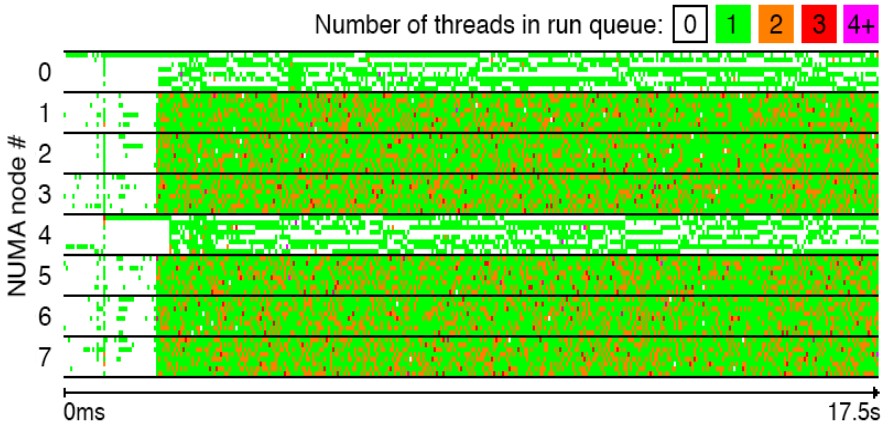
- The Linux scheduler has performance bugs!
- Showed this last year @EuroSys
  « The Linux Scheduler: A Decade of Wasted Cores »



- Work-conservation invariant not maintained:
  - Idle cores while several threads running on some cores
  - Situation lasts for a long time (several seconds, sometimes forever)

The problem: perf bugs in scheduler

- The Linux scheduler has performance bugs!
- Showed this last year @EuroSys
  « The Linux Scheduler: A Decade of Wasted Cores »



- Work-conservation invariant not maintained:
  - Idle cores while several threads running on some cores
  - Situation lasts for a long time (several seconds, sometimes forever)

- Consequences:
  - Wasted energy, infrastructure resources, lower bandwidth, higher latency...
  - Lack of predictability: harder to scale-out!

The problem: perf bugs in scheduler

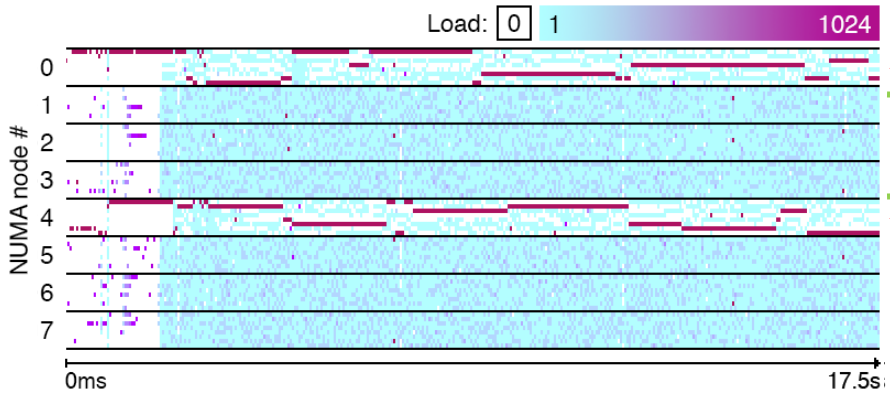
- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 1: problem with the way load is calculated

The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 1: problem with the way load is calculated
- **Idea:** the scheduler thinks the load is balanced if nodes have same average load

The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 1: problem with the way load is calculated
- **Idea:** the scheduler thinks the load is balanced if nodes have same average load
- **Not necessarily the case!**



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

Load 2 = avg(many make threads with low load)

Load 1 = Load 2: the scheduler thinks the load is balanced!

The problem: perf bugs in scheduler

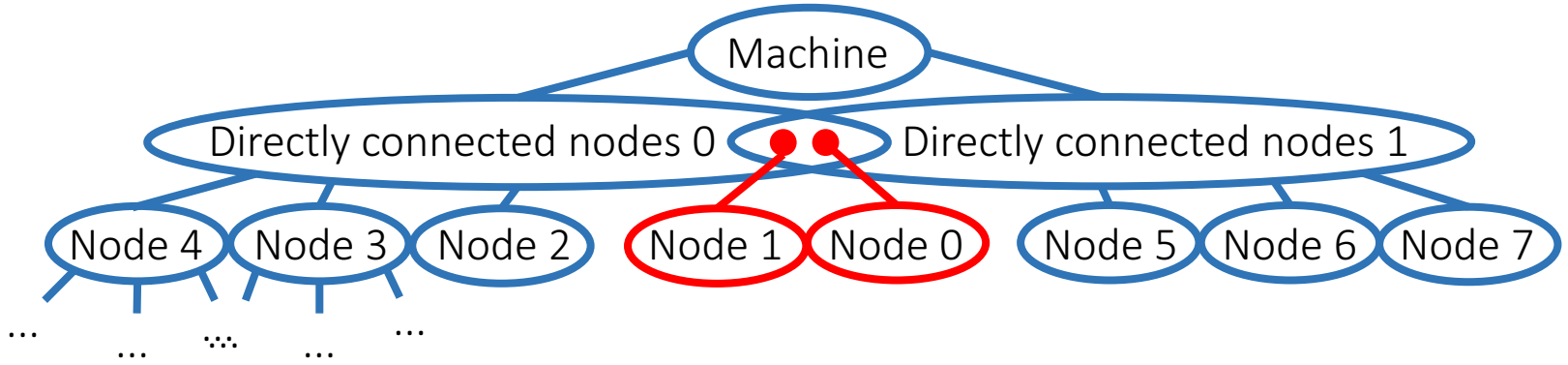
- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bugs 2 & 3: problem with the way the hierarchy is built

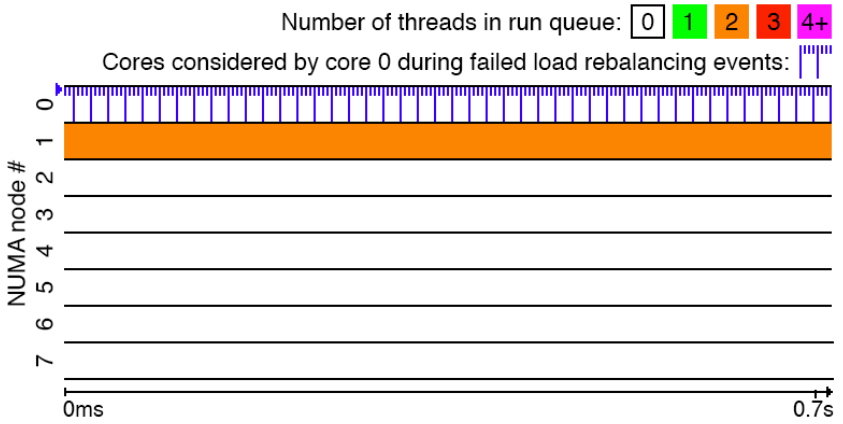
The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bugs 2 & 3: problem with the way the hierarchy is built
- **E.g., idea of bug 2:** at the last level (connected nodes), one node in both groups

The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bugs 2 & 3: problem with the way the hierarchy is built
- **E.g., idea of bug 2:** at the last level (connected nodes), one node in both groups
- **Threads on that core never balanced :** load of both groups equal





The problem: perf bugs in scheduler

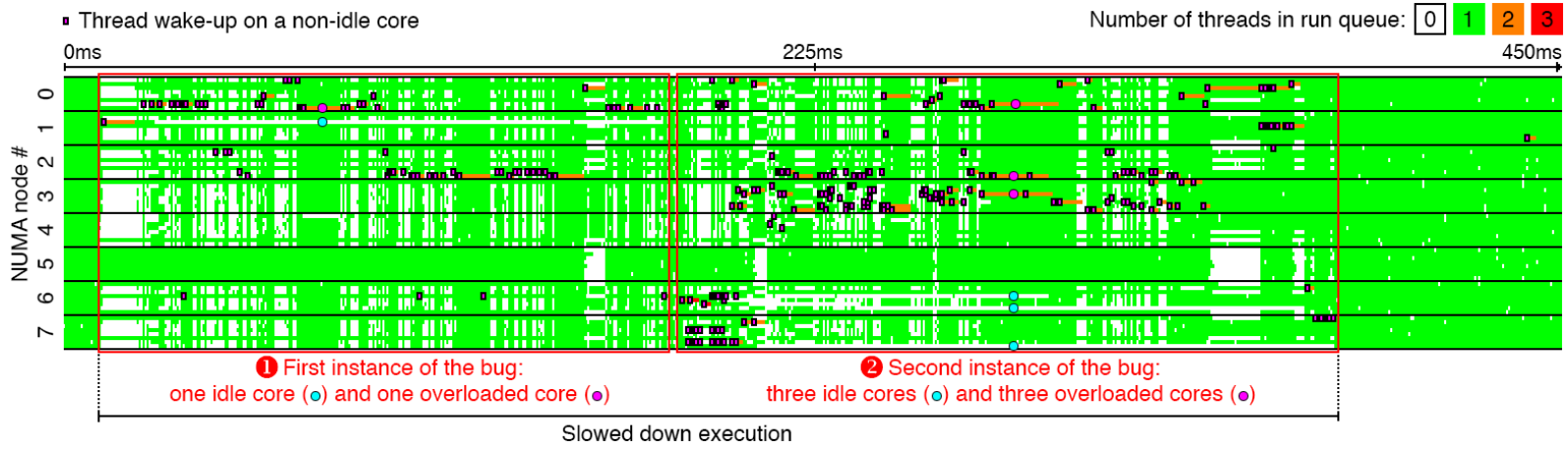
- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 4: problem with « smart » wakeups

The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 4: problem with « smart » wakeups
- **Idea of bug 4:** periodic load balancing global, « smart » wakeups on local node

The problem: perf bugs in scheduler

- **Work-conservation invariant not maintained:** four bugs described in the paper *« The Linux Scheduler: A Decade of Wasted Cores »*
- Bug 4: problem with « smart » wakeups
- **Idea of bug 4:** periodic load balancing global, « smart » wakeups on local node
- **One makes mistakes the other can’t fix!**



Analysis: Linux scheduler too complex!

- **Linux used for many classes of applications**
  - Cloud hosting, database, n-tier services, HPC...
  - Interactive applications

Analysis: Linux scheduler too complex!

- Linux used for many classes of applications
  - Cloud hosting, database, n-tier services, HPC...
  - Interactive applications
- Multicore architectures increasingly diverse and complex!

Analysis: Linux scheduler too complex!

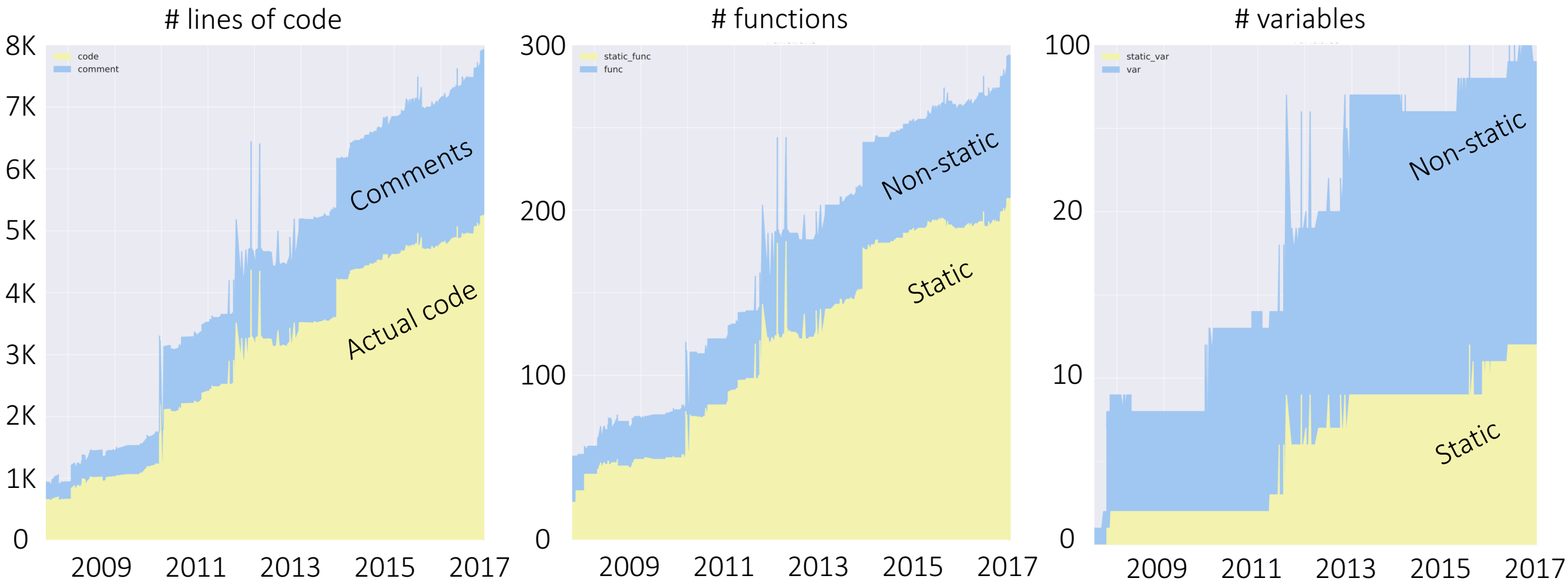
- Linux used for many classes of applications
  - Cloud hosting, database, n-tier services, HPC...
  - Interactive applications
- Multicore architectures increasingly diverse and complex!
- Result: a very complex monolithic scheduler, supposed to work in all situations!
  - Many heuristics interact in complex, unpredictable ways
  - Some features greatly complexify, e.g., load balancing (tasksets, cgroups/autogroups...)

Analysis: Linux scheduler too complex!

- Linux used for many classes of applications
  - Cloud hosting, database, n-tier services, HPC...
  - Interactive applications
- Multicore architectures increasingly diverse and complex!
- Result: a 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!

Analysis: Linux scheduler too complex!

For instance, `fair.c`:



Solution?

- A solution: prove scheduler implementation correct?
  - Way too much code for current technology
  - We'd need to detect high-level abstractions from low-level C: a challenge!

Solution?

- **A solution: prove scheduler implementation correct?**
  - Way too much code for current technology
  - We'd need to detect high-level abstractions from low-level C: a challenge!

- **Supposing we managed this feat through hard work...**
  - How do we keep up with updates?
  - The code keeps evolving with new architectures and application needs...

Solution?

- **A solution: prove scheduler implementation correct?**
  - Way too much code for current technology
  - We'd need to detect high-level abstractions from low-level C: a challenge!

- **Supposing we managed this feat through hard work…**
  - How do we keep up with updates?
  - The code keeps evolving with new architectures and application needs…

- Not doable! We need another approach…

Our solution: Ipanema

- A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion
    - *E.g., more preemption for more interactive applications…*
  - Specific load balancing criterion
    - *EDF for real-time apps, locality-aware balancing…*
  - Event-based state machine (new, block, unblock, terminate, tick, balance)…

Our solution: Ipanema

- A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion
    - *E.g., more preemption for more interactive applications…*
  - Specific load balancing criterion
    - *EDF for real-time apps, locality-aware balancing…*
  - Event-based state machine (new, block, unblock, terminate, tick, balance)…

- Machine partitioned into sets of cores that run ≠ schedulers

Our solution: Ipanema

- A scheduler is tailored to a (class of) parallel application(s)
  - Specific thread election criterion
    - *E.g., more preemption for more interactive applications…*
  - Specific load balancing criterion
    - *EDF for real-time apps, locality-aware balancing…*
  - Event-based state machine (new, block, unblock, terminate, tick, balance)…

- Machine partitioned into sets of cores that run ≠ schedulers
- Scheduler deployed together with an application on a partition

Scientific challenges

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)

Scientific challenges

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be **proven safe** so that they do not hang or crash the kernel

Scientific challenges

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be proven safe so that they do not hang or crash the kernel
3. Scheduling policies must be proven free of the recently identified performance bugs

Scientific challenges

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be proven safe so that they do not hang or crash the kernel
3. Scheduling policies must be proven free of the recently identified performance bugs
4. Scheduling policies must capture the diversity of modern multicore architectures

Scientific challenges

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be proven safe so that they do not hang or crash the kernel
3. Scheduling policies must be proven free of the recently identified performance bugs
4. Scheduling policies must capture the diversity of modern multicore architectures
5. The approach should not introduce a performance penalty

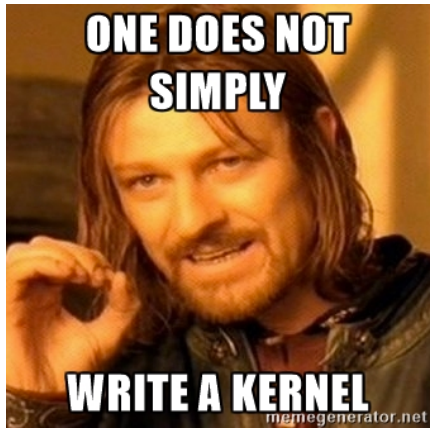
Challenge 1: ease of implementation

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)

Challenge 1: ease of implementation

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)

Problem: kernel development is (still) a nightmare, error-prone!

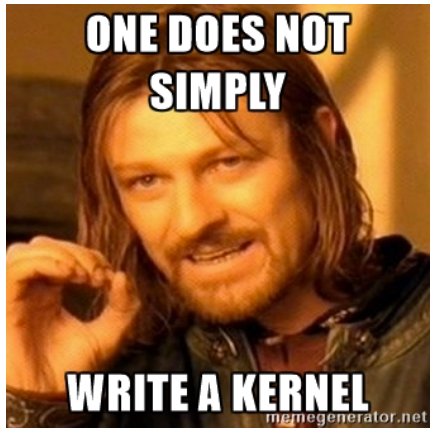


Challenge 1: ease of implementation

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)

Problem: kernel development is (still) a nightmare, error-prone!

-Low-level C code ⇒ little help from the compiler!

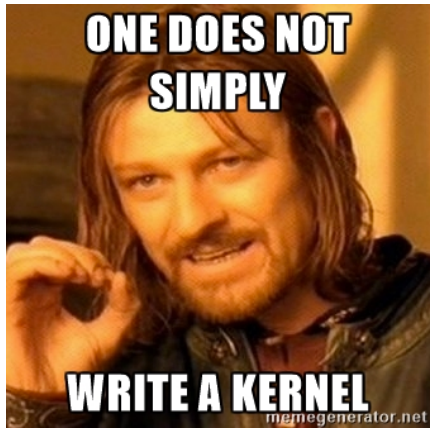


Challenge 1: ease of implementation

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)

Problem: kernel development is (still) a nightmare, error-prone!

- Low-level C code ⇒ little help from the compiler!
- Likely to crash/hang the OS!
  - Testing/debugging time-consuming, tedious!
  - Not all stack trace info easily available...

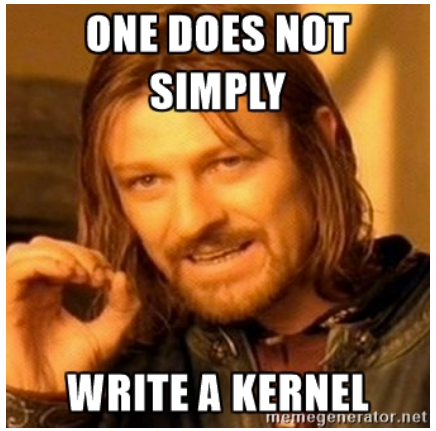


Challenge 1: ease of implementation

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)

Problem: kernel development is (still) a nightmare, error-prone!

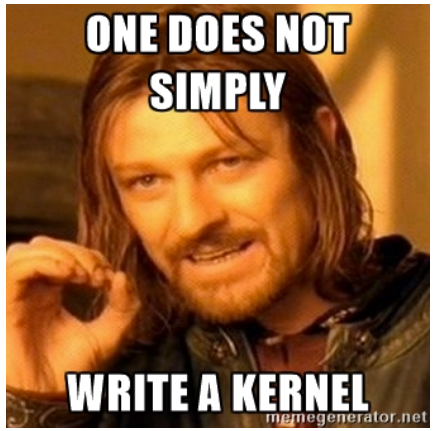
- Low-level C code ⇒ little help from the compiler!
- Likely to crash/hang the OS!
  - Testing/debugging time-consuming, tedious!
  - Not all stack trace info easily available...
- No clear framework for writing schedulers ⇒ unclear interactions, synchro. issues!



Challenge 1: ease of implementation

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)

Problem: kernel development is (still) a nightmare, error-prone!



- Low-level C code ⇒ little help from the compiler!
- Likely to crash/hang the OS!
  - Testing/debugging time-consuming, tedious!
  - Not all stack trace info easily available...
- No clear framework for writing schedulers ⇒ unclear interactions, synchro. issues!
- More issues, e.g., optimizations hinder code maintenance
  - Target-specific implementation of mechanisms ⇒ policy obfuscated!

Challenge 1: solution

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)

Solution: capture kernel expertise into a Domain-Specific Language (DSL)!

Challenge 1: solution

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)

Solution: capture kernel expertise into a Domain-Specific Language (DSL)!

DSL: A programming language dedicated to a family of programs that offers specific abstractions and notations.

Challenge 1: solution

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)

Solution: capture kernel expertise into a Domain-Specific Language (DSL)!

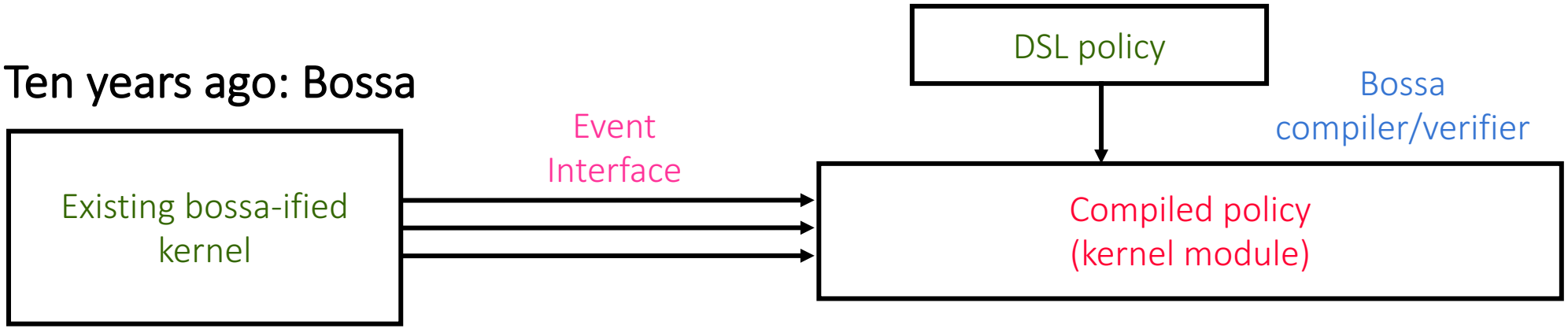
DSL: A programming language dedicated to a family of programs that offers specific abstractions and notations.

-Trade expressiveness for expertise/knowledge:

- Productivity: easier and safer programming
- Robustness: (static) verification of properties
- Performance: efficient compilation

Bossa

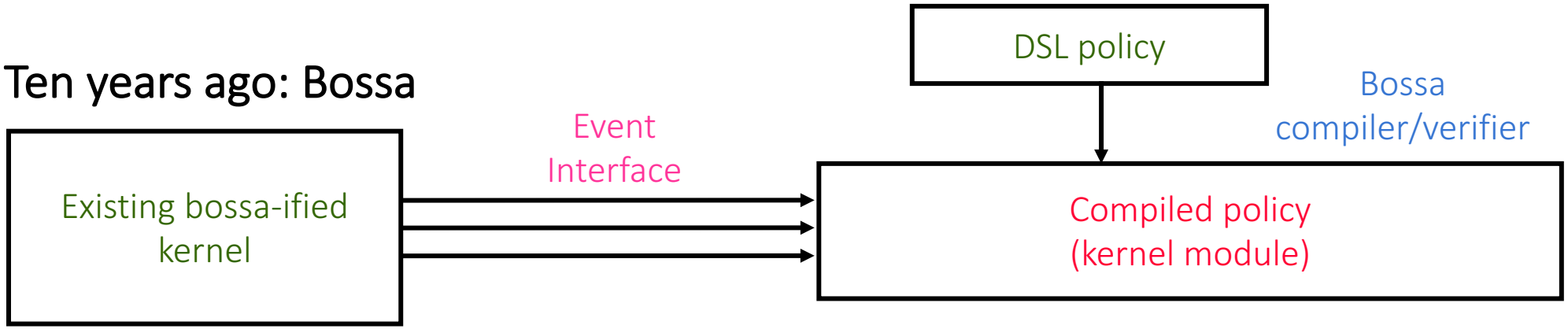
- **Ten years ago: Bossa**



- **Idea:** enrich an existing kernel with a scheduling-specific event interface
- Framework and rules for developing a scheduler

Bossa

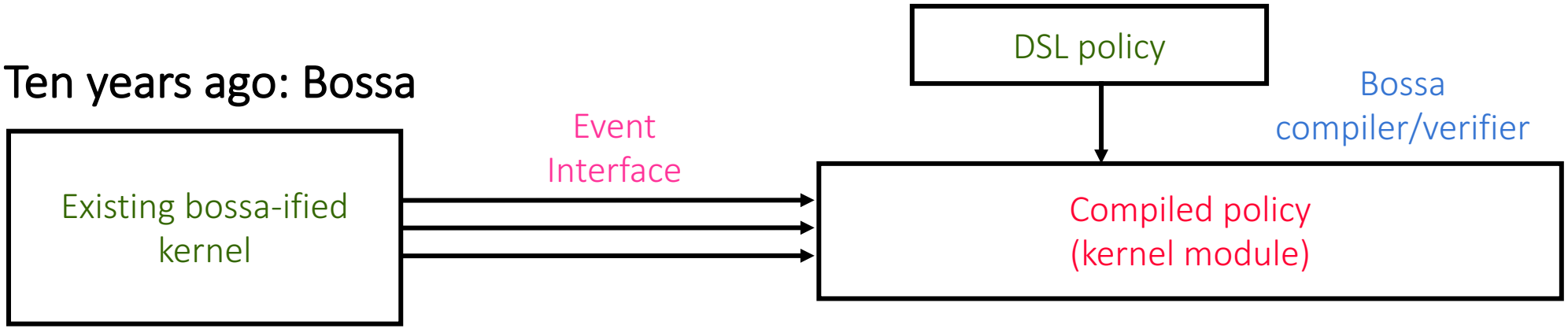
- **Ten years ago: Bossa**



- **Idea:** enrich an existing kernel with a scheduling-specific event interface
- Framework and rules for developing a scheduler
- Used for teaching scheduling

Bossa

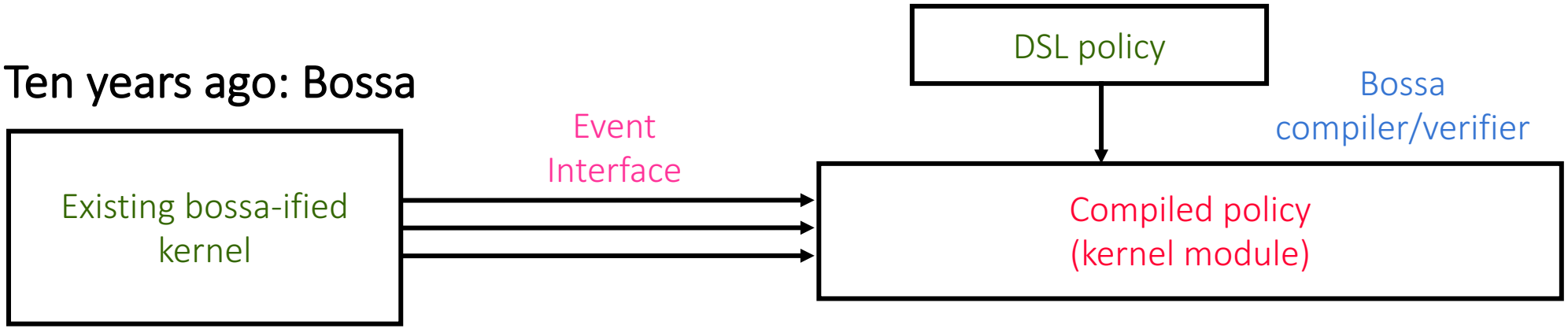
- **Ten years ago: Bossa**



- **Idea:** enrich an existing kernel with a scheduling-specific event interface
- Framework and rules for developing a scheduler
- Used for teaching scheduling
- Related publications [ASE 2003, EW 2004, HASE 2006]

Bossa

- Ten years ago: Bossa



- **Idea:** enrich an existing kernel with a scheduling-specific event interface
- Framework and rules for developing a scheduler
- Used for teaching scheduling
- Related publications [ASE 2003, EW 2004, HASE 2006]
- Target: single-core systems only!

Bossa

Bossa provides: 1, 2, and 5

1. Implementing scheduling policies must be simple enough to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be proven safe so that they do not hang or crash the kernel
3. Scheduling policies must be proven free of the recently identified performance bugs
4. Scheduling policies must capture the diversity of modern multicore architectures
5. The approach should not introduce a performance penalty

The Ipanema DSL

- Abstractions inherited from the Bossa DSL

The Ipanema DSL

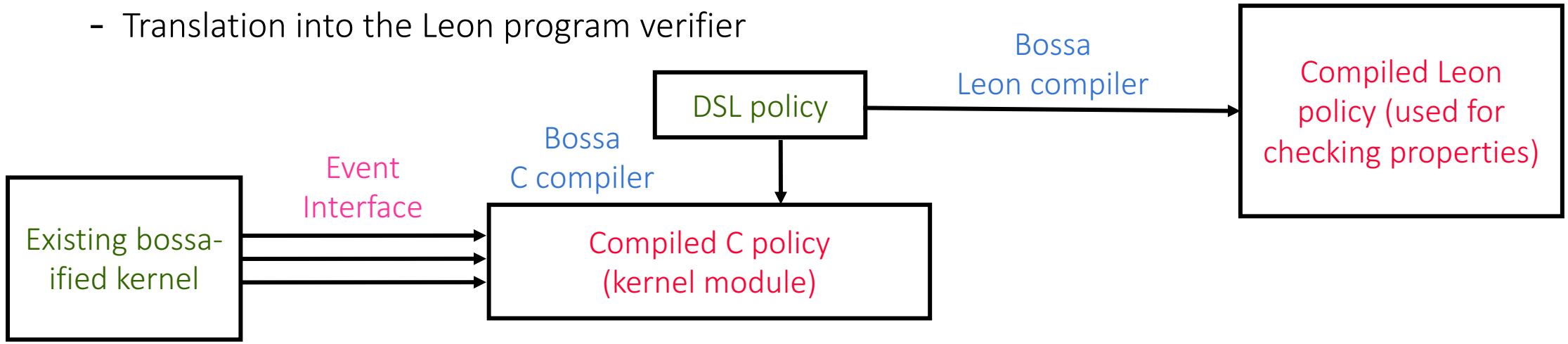
- Abstractions inherited from the Bossa DSL
- Abstractions dedicated to multicore architectures
  - Objective: no explicit synchronization

The Ipanema DSL

- Abstractions inherited from the Bossa DSL
- Abstractions dedicated to multicore architectures
  - Objective: no explicit synchronization
- Verification of properties
  - Co-design of the proofs with the design of the DSL abstractions
  - Translation into the Leon program verifier

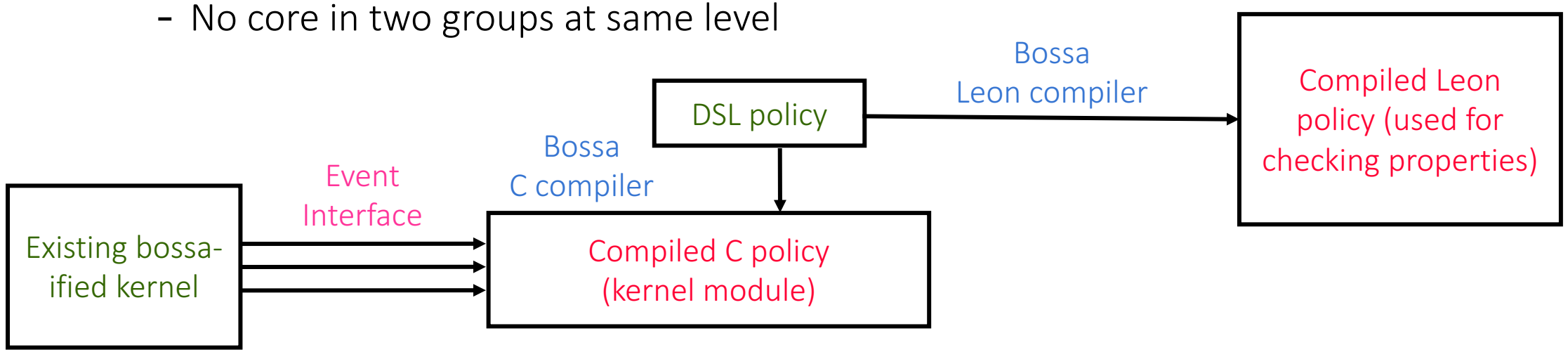
The Ipanema DSL

- Abstractions inherited from the Bossa DSL
- Abstractions dedicated to multicore architectures
  - Objective: no explicit synchronization
- Verification of properties
  - Co-design of the proofs with the design of the DSL abstractions
  - Translation into the Leon program verifier



The Ipanema DSL

- Properties checked with Leon:
  - Load-balancing is work-conserving (can ensure it on « reasonable » policies)
  - Load is balanced in finite number of rounds of load-balancing (assuming « stable » system)
  - Load-balancing hierarchy is valid:
    - Top level contains all cores
    - No core in two groups at same level



Scientific challenges

Ipanema also provides: 3 and 4

1. Implementing **scheduling policies must be simple enough** to be doable by an application developer (not a Linux kernel expert)
2. Scheduling policies must be **proven safe** so that they do not hang or crash the kernel
3. Scheduling policies must be proven free of the recently identified performance bugs
4. Scheduling policies must capture the diversity of modern multicore architectures
5. The approach **should not introduce a performance penalty**

The Ipanema DSL

What's inherited from Bossa?

The Ipanema DSL

What's inherited from Bossa?

- **Abstractions:**
  - Thread attributes
  - Ordering criteria
  - Thread states
  - Event handlers
  - A few more things (interface functions...)

The Ipanema DSL

What's inherited from Bossa?

- Abstractions:
  - Thread attributes
  - Ordering criteria
  - Thread states
  - Event handlers
  - A few more things (interface functions...)
- Properties (mandatory):
  - Termination of events, bounded loops
  - Valid state transitions
  - No loss of a thread

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

Process/thread-local variables.

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

Process/thread-local variables.

Number of quanta the process has been running for.

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

Process/thread-local variables.

Number of quanta the process has been running for.

Core-local, process-related variables.

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

Process/thread-local variables.

Number of quanta the process has been running for.

Core-local, process-related variables.

Process currently running on the core.

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

Process/thread-local variables.

Number of quanta the process has been running for.

Core-local, process-related variables.

Process currently running on the core.

List of processes, ordered by quantum (lazy evaluation), can be accessed by other processes (**shared** keyword).

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

Process/thread-local variables.

Number of quanta the process has been running for.

Core-local, process-related variables.

Process currently running on the core.

List of processes, ordered by quantum (lazy evaluation), can be accessed by other processes (**`shared`** keyword).

List of blocked processes (on an I/O, a lock).

The Ipanema DSL: a very basic example

Process/thread and core-local abstractions:

```
process = {
   int quanta;
   int load;
}

core = {
   processes = {
      RUNNING process current;
      shared READY set<process> ready : order = {lowest quanta};

      BLOCKED set<process> blocked;
      TERMINATED terminated;
   }
...
}
```

Process/thread-local variables.

Number of quanta the process has been running for.

Core-local, process-related variables.

Process currently running on the core.

List of processes, ordered by quantum (lazy evaluation), can be accessed by other processes (**shared** keyword).

List of blocked processes (on an I/O, a lock).

No reference kept (pseudo-state).

The Ipanema DSL: a very basic example

Process events:

```
handler (process_event e) {
   on tick {
      e.target.quanta++;
      if (e.target.quanta % 5 == 0) {
         e.target => ready;
      }
   }
   on yield {
      e.target => ready;
   }
   on block {
      e.target => blocked;
   }
   on unblock {
      e.target => ready;
   }
   on schedule {
      first(ready) => current;
   }
}
```

The Ipanema DSL: a very basic example

Process events:

```
handler (process_event e) {
   on tick {
      e.target.quanta++;
      if (e.target.quanta % 5 == 0) {
         e.target => ready;
      }
   }
   on yield {
      e.target => ready;
   }
   on block {
      e.target => blocked;
   }
   on unblock {
      e.target => ready;
   }
   on schedule {
      first(ready) => current;
   }
}
```

Handlers for all events regarding a process (or thread).

The Ipanema DSL: a very basic example

Process events:

```
handler (process_event e) {
   on tick {
      e.target.quanta++;
      if (e.target.quanta % 5 == 0) {
         e.target => ready;
      }
   }
   on yield {
      e.target => ready;
   }
   on block {
      e.target => blocked;
   }
   on unblock {
      e.target => ready;
   }
   on schedule {
      first(ready) => current;
   }
}
```

Handlers for all events regarding a process (or thread).

Context switch (will trigger schedule). Implicit list management.

The Ipanema DSL: a very basic example

Process events:

```
handler (process_event e) {
   on tick {
      e.target.quanta++;
      if (e.target.quanta % 5 == 0) {
         e.target => ready;
      }
   }
   on yield {
      e.target => ready;
   }
   on block {
      e.target => blocked;
   }
   on unblock {
      e.target => ready;
   }
   on schedule {
      first(ready) => current;
   }
}
```

Handlers for all events regarding a process (or thread).

Context switch (will trigger schedule). Implicit list management.

Uses **`ready`**'s ordering criterion.

The Ipanema DSL: a very basic example

Process events:

```
handler (process_event e) {
    on tick {
        e.target.quanta++;
        if (e.target.quanta % 5 == 0) {
            e.target => ready;
        }
    }
    on yield {
        e.target => ready;
    }
    on block {
        e.target => blocked;
    }
    on unblock {
        e.target => ready;
    }
    on schedule {
        first(ready) => current;
    }
}
```

Handlers for all events regarding a process (or thread).

Context switch (will trigger schedule). Implicit list management.

Valid state transitions checked at compile-time.

Uses **`ready`**'s ordering criterion.

The Ipanema DSL

What's new? Mostly multicore stuff.

The Ipanema DSL

What's new? Mostly multicore stuff.

- Abstractions:
  - Core attributes
  - Load criteria
  - Groups of cores
  - Core handlers
  - Load balancing functions

The Ipanema DSL

What's new? Mostly multicore stuff.

- Abstractions:
  - Core attributes
  - Load criteria
  - Groups of cores
  - Core handlers
  - Load balancing functions
- Performance/synchronization properties:
  - Locking/sychronization handled by the framework
  - Mostly trylocks: if unable to lock a runqueue, give up on stealing thread (best effort)
  - Ensure no performance bugs

The Ipanema DSL: a very basic example

Multicore abstractions:

```
domain = {
   set<group> groups;
}

group = {
   set<core> cores;
   lazy int load = sum(cores.load);
   int capacity = count(cores);

   lazy bool is_stealable = or(cores.is_stealable);
}
```

The Ipanema DSL: a very basic example

Multicore abstractions:

```
domain = {
   set<group> groups;
}

group = {
   set<core> cores;
   lazy int load = sum(cores.load);
   int capacity = count(cores);

   lazy bool is_stealable = or(cores.is_stealable);
}
```

Scheduling hierarchy: works like in Linux, i.e. tree where at each level a domain contains groups, themselves being domains of lower level.

The Ipanema DSL: a very basic example

Multicore abstractions:

```
domain = {
    set<group> groups;
}

group = {
    set<core> cores;
    lazy int load = sum(cores.load);
    int capacity = count(cores);

    lazy bool is_stealable = or(cores.is_stealable);
}
```

Scheduling hierarchy: works like in Linux, i.e. tree where at each level a domain contains groups, themselves being domains of lower level.

Evaluated when value is read (`lazy`).

The Ipanema DSL: a very basic example

Multicore abstractions:

```
domain = {
    set<group> groups;
}

group = {
    set<core> cores;
    lazy int load = sum(cores.load);
    int capacity = count(cores);

    lazy bool is_stealable = or(cores.is_stealable);
}
```

Scheduling hierarchy: works like in Linux, i.e. tree where at each level a domain contains groups, themselves being domains of lower level.

Evaluated when value is read (`lazy`).

Stealing from this group won't cause load conservation issues.

The Ipanema DSL: a very basic example

Multicore abstractions:

```
domain = {
    set<group> groups;
}

group = {
    set<core> cores;
    lazy int load = sum(cores.load);
    int capacity = count(cores);

    lazy bool is_stealable = or(cores.is_stealable);
}
```

Scheduling hierarchy: works like in Linux, i.e. tree where at each level a domain contains groups, themselves being domains of lower level.

Evaluated when value is read (`lazy`).

Stealing from this group won't cause load conservation issues.

Group stealable iff one of its cores is.

The Ipanema DSL: a very basic example

Core abstractions:

```
core = {

   ...

   system int id;
   lazy int load = sum(current.load, ready.load);
   lazy bool is_stealable = count(current, ready) > 1;
   set<domain> scheduling_domains;

   domains (core self) to scheduling_domains = {
      foreach (dist in distances starting_at 1) {
         domain (c | distance(c, self) <= dist) to groups = {
            group (c1,c2 | distance(c1, c2) <= dist - 1) to cores;
         }
      }
   }
}
```

The Ipanema DSL: a very basic example

Core abstractions:

```
core = {

   ...

   system int id;
   lazy int load = sum(current.load, ready.load);
   lazy bool is_stealable = count(current, ready) > 1;
   set<domain> scheduling_domains;

   domains (core self) to scheduling_domains = {
      foreach (dist in distances starting_at 1) {
         domain (c | distance(c, self) <= dist) to groups = {
            group (c1,c2 | distance(c1, c2) <= dist - 1) to cores;
         }
      }
   }
}
```

Obtained from the kernel.

The Ipanema DSL: a very basic example

Core abstractions:

```
core = {

   ...

   system int id;
   lazy int load = sum(current.load, ready.load);
   lazy bool is_stealable = count(current, ready) > 1;
   set<domain> scheduling_domains;

   domains (core self) to scheduling_domains = {
      foreach (dist in distances starting_at 1) {
         domain (c | distance(c, self) <= dist) to groups = {
            group (c1,c2 | distance(c1, c2) <= dist - 1) to cores;
         }
      }
   }
}
```

Obtained from the kernel.

Be work-conserving (basic).

The Ipanema DSL: a very basic example

Core abstractions:

```
core = {

    ...

    system int id;
    lazy int load = sum(current.load, ready.load);
    lazy bool is_stealable = count(current, ready) > 1;
    set<domain> scheduling_domains;

    domains (core self) to scheduling_domains = {
        foreach (dist in distances starting_at 1) {
            domain (c | distance(c, self) <= dist) to groups = {
                group (c1,c2 | distance(c1, c2) <= dist - 1) to cores;
            }
        }
    }
}
```

Obtained from the kernel.

Be work-conserving (basic).

Hierarchy-building functions co-designed with proofs: Leon code checks good properties (top domain contains all cores, no core in two groups at the same level...).

The Ipanema DSL: a very basic example

Load balancing: who steals whom?

```
handler (core_event e) {

   on balancing_select {

      foreach (sd in e.target.scheduling_domains) {
         group busiest = max(sd.groups order = { highest load / capacity } filter = { is_stealable });

         if (valid(busiest)) {
            core busiest_core = max(busiest.core order = { highest load } filter = { ready.size >= 1 });
            balancing_steal(e.target, busiest_core);
         }
      }

   }

}
```

The Ipanema DSL: a very basic example

Load balancing: who steals whom?

Load balancing event.

```
handler (core_event e) {

   on balancing_select {

      foreach (sd in e.target.scheduling_domains) {
         group busiest = max(sd.groups order = { highest load / capacity } filter = { is_stealable });

         if (valid(busiest)) {
            core busiest_core = max(busiest.core order = { highest load } filter = { ready.size >= 1 });
            balancing_steal(e.target, busiest_core);
         }
      }

   }

}
```

The Ipanema DSL: a very basic example

Load balancing: who steals whom?

Load balancing event.

```
handler (core_event e) {

   on balancing_select {

      foreach (sd in e.target.scheduling_domains) {
         group busiest = max(sd.groups order = { highest load / capacity } filter = { is_stealable });

         if (valid(busiest)) {
            core busiest_core = max(busiest.core order = { highest load } filter = { ready.size >= 1 });
            balancing_steal(e.target, busiest_core);
         }
      }

   }

}
```

Load-balancing logic similar to Linux (simplified).

The Ipanema DSL: a very basic example

Load balancing: stealing processes

```
try void balancing_steal(core self, core busiest) {

      int imbalance = (busiest.load - self.load) / 2;
      if (imbalance <= 0)
        return;

      foreach (p in busiest.ready) {
         if (imbalance < p.load)
            continue;

          p => self.ready;

          imbalance -= p.load;
          if (imbalance <= 0)
             break;
      }
}
```

The Ipanema DSL: a very basic example

Load balancing: stealing processes

Acquires locks automatically and may quietly fail (best effort).

```
try void balancing_steal(core self, core busiest) {

      int imbalance = (busiest.load - self.load) / 2;
      if (imbalance <= 0)
        return;

      foreach (p in busiest.ready) {
         if (imbalance < p.load)
            continue;

          p => self.ready;

          imbalance -= p.load;
          if (imbalance <= 0)
             break;
      }
}
```

Ipanema: to conclude

- Makes programming multicore scheduling policies possible for non-kernel experts

Ipanema: to conclude

- Makes programming multicore scheduling policies possible for non-kernel experts
- Ensures safety and performance properties:
  - Valid state transitions, bounded loops, terminating events, no loss of process
  - Work-conservation, eventual balancing, valid hierarchy

Ipanema: to conclude

- Makes programming multicore scheduling policies possible for non-kernel experts
- Ensures safety and performance properties:
  - Valid state transitions, bounded loops, terminating events, no loss of process
  - Work-conservation, eventual balancing, valid hierarchy
- Useful for research, teaching, and real-world scenarios

Ipanema: to conclude

- Makes programming multicore scheduling policies possible for non-kernel experts
- Ensures safety and performance properties:
  - Valid state transitions, bounded loops, terminating events, no loss of process
  - Work-conservation, eventual balancing, valid hierarchy
- Useful for research, teaching, and real-world scenarios
- Current status:
  - DSL nearly completed, verification of static properties
  - Basic versions of the Ipanema runtime and compiler
  - Manual verifications of multicore properties with Leon

Ipanema: to conclude

- Makes programming multicore scheduling policies possible for non-kernel experts
- Ensures safety and performance properties:
  - Valid state transitions, bounded loops, terminating events, no loss of process
  - Work-conservation, eventual balancing, valid hierarchy
- Useful for research, teaching, and real-world scenarios
- Current status:
  - DSL nearly completed, verification of static properties
  - Basic versions of the Ipanema runtime and compiler
  - Manual verifications of multicore properties with Leon
- Everything about the old Bossa DSL: http://bossa.lip6.fr