Thread Scheduling in Multi-core Operating Systems

How to Understand, Improve and Fix your Scheduler

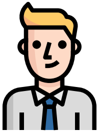
Redha Gouicem

Thesis defended on the 23rd of October 2020 before a jury composed of:

Mr. Pascal Felber, Full Professor, Université de Neuchâtel	*Reviewer*
Mr. Vivien Quéma, Full Professor, Grenoble INP (ENSIMAG)	*Reviewer*
Mr. Rachid Guerraoui, Full Professor, École Polytechnique Fédérale de Lausanne	*Examiner*
Ms. Karine Heydemann, Associate Professor (HDR), Sorbonne Université	*Examiner*
Mr. Etienne Rivière, Full Professor, Université Catholique de Louvain	*Examiner*
Mr. Gilles Muller, Senior Research Scientist, Inria	*Advisor*
Mr. Julien Sopena, Associate Professor, Sorbonne Université	*Advisor*

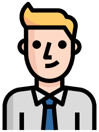
SORBONNE UNIVERSITÉ

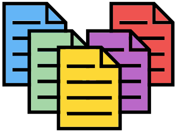
Office worker



Scheduling

Office worker
Task



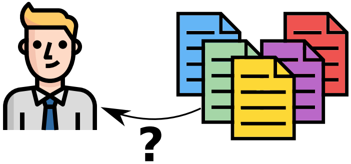


Scheduling

Office worker

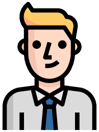
Task

Scheduling: *Choosing the order in which tasks are performed*



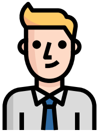
Scheduling

Office worker $\Rightarrow$ **CPU**
Task $\Rightarrow$ **Application**
Scheduling: *Choosing the order in which tasks are performed*



Office worker $\Rightarrow$ **CPU**
Task $\Rightarrow$ **Application**
Scheduling: *Choosing the order in which tasks are performed*

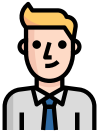
Before 1955: Human operators



Scheduling

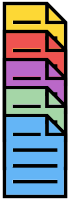
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

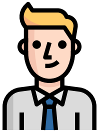
Before 1955: Human operators



Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators

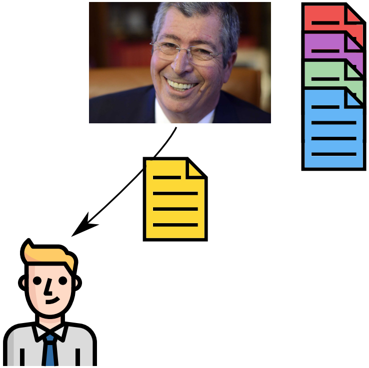




Scheduling

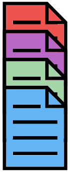
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

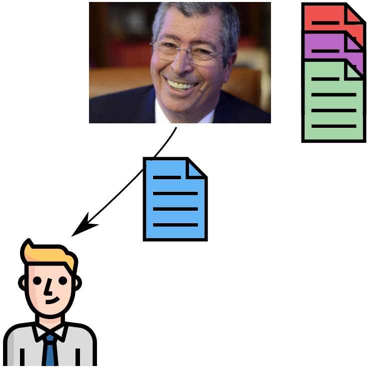


Before 1955: Human operators

Scheduling

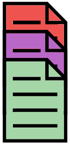
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators

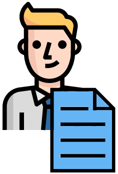


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



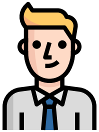
Before 1955: Human operators



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

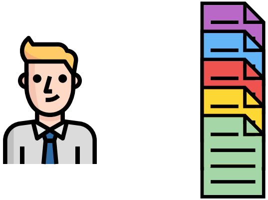
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

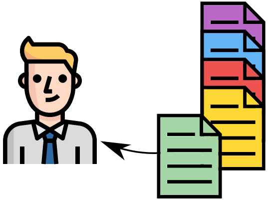
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

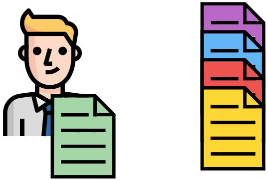
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

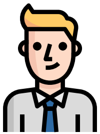
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

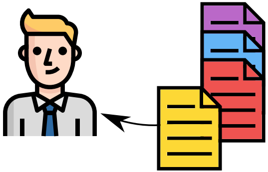
Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

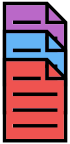
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

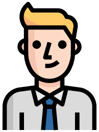
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

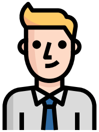
Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)



Scheduling

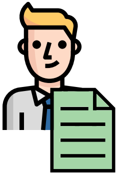
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)



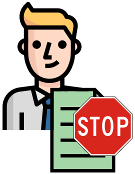
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)



Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)



Office worker ⇒ **CPU**

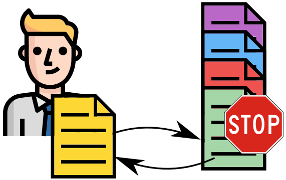
Task ⇒ **Application**

Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators

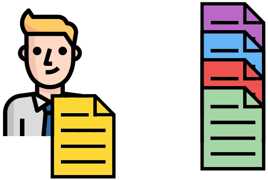
1955: 1^{st} OS with batch scheduler (GM-NAA)

1967: Multiprogramming (IBM OS/360 MFT/MVT)



Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

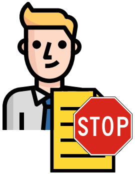
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)

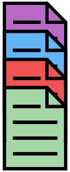


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)

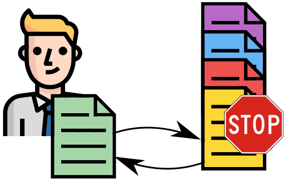




Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

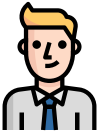
Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

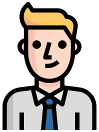
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)

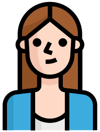


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)

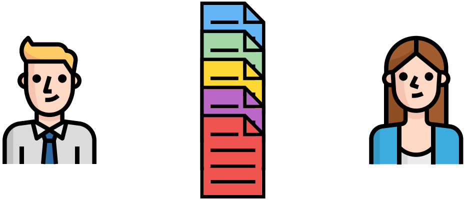




Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

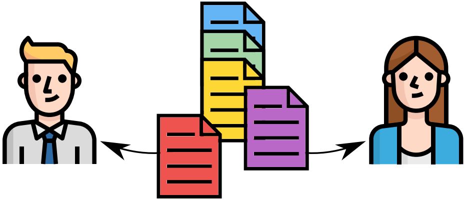
Before 1955: Human operators
1955: 1*st* OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)



Scheduling

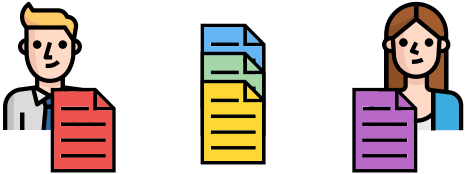
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)



Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

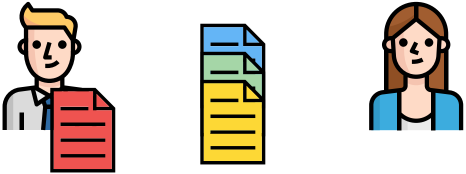
Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)



Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)

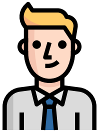


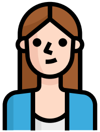
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)



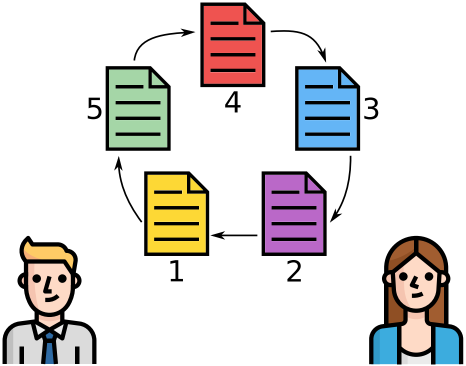


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

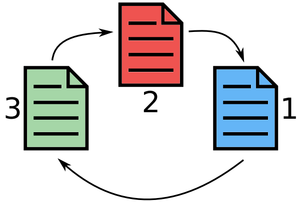
Before 1955: Human operators
1955: 1*st* OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

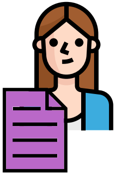


Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

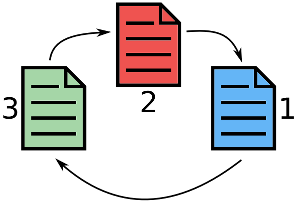


Before 1955: Human operators
1955: 1*st* OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)



Scheduling

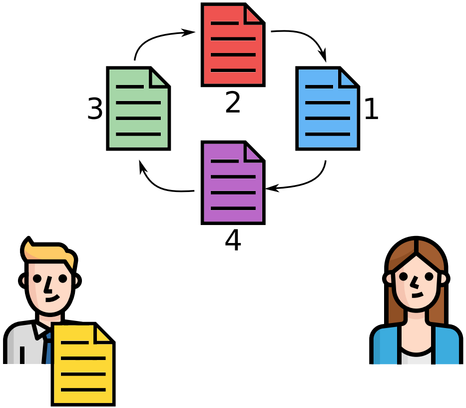
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

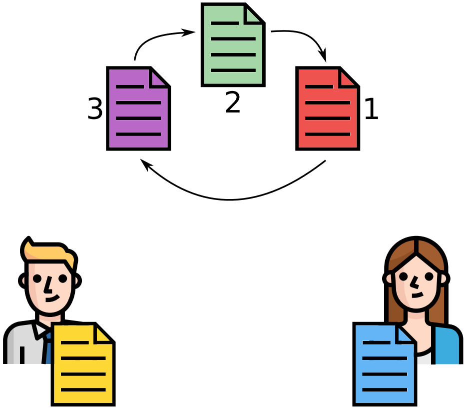
Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

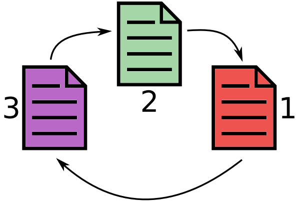
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

Scheduling

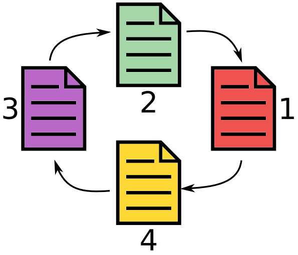
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



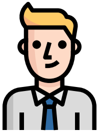
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

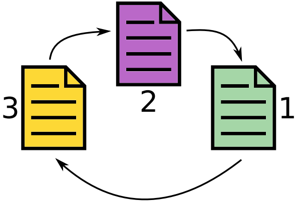


Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)



Scheduling

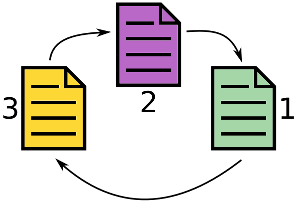
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



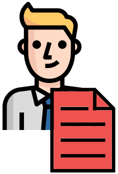
Before 1955: Human operators
1955: 1st OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)

Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

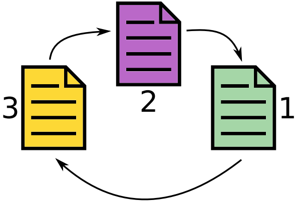


Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)
1990s: NUMA architectures

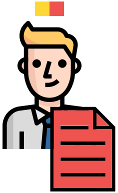


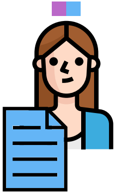
Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



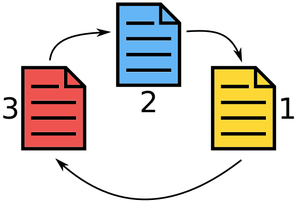
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)
1990s: NUMA architectures



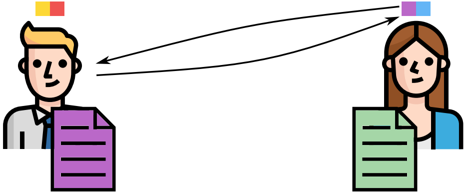


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

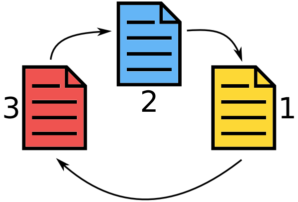


Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)
1990s: NUMA architectures

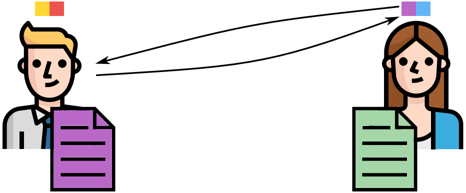


Scheduling

Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*



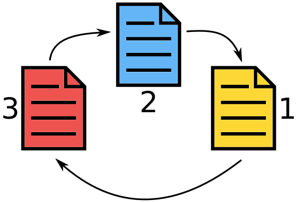
Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)
1990s: NUMA architectures

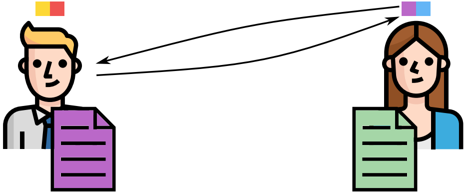


Scheduling

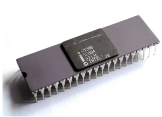
Office worker ⇒ **CPU**
Task ⇒ **Application**
Scheduling: *Choosing the order in which tasks are performed*

Before 1955: Human operators
1955: 1^{st} OS with batch scheduler (GM-NAA)
1967: Multiprogramming (IBM OS/360 MFT/MVT)
1968: Multiprocessors (IBM OS/360 M65MP)
1971: Time sharing (IBM OS/360)
1990s: NUMA architectures
2000s: Heterogeneous architectures, SMT, frequency scaling, . . .



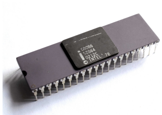


When Schedulers Answer to Hardware Evolution



Single core	Time management	batch processing preemption time sharing

When Schedulers Answer to Hardware Evolution

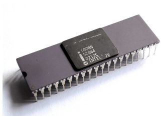


Single core | Time management { batch processing, preemption, time sharing

Multi-processors
Multi-core
NUMA

Placement management { shared resources, contention, latency

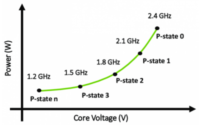
When Schedulers Answer to Hardware Evolution



Single core | Time management
- batch processing
- preemption
- time sharing

Multi-processors
Multi-core
NUMA

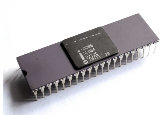
Placement management
- shared resources
- contention
- latency



Dynamic frequency
Heterogeneity

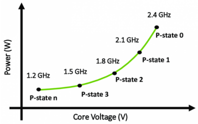
Feature management
- perf/energy ratio
- different capabilities

When Schedulers Answer to Hardware Evolution



Single core | Time management
- batch processing
- preemption
- time sharing

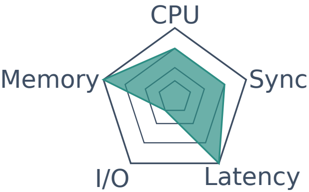
Multi-processors
Multi-core
NUMA
| Placement management
- shared resources
- contention
- latency



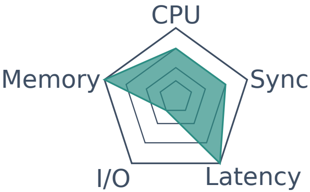
Dynamic frequency
Heterogeneity
| Feature management
- perf/energy ratio
- different capabilities

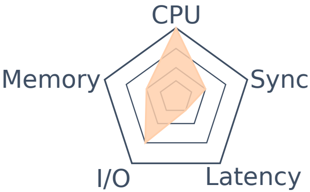
Resources are more and more complex to manage!

Application Requirements

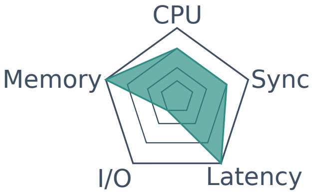


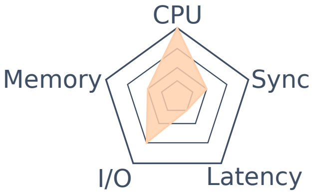
Application Requirements

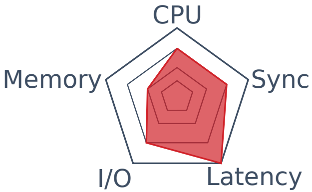




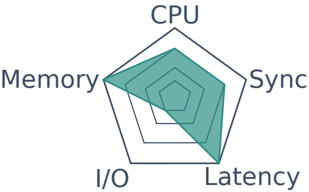
Application Requirements

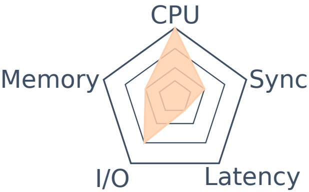


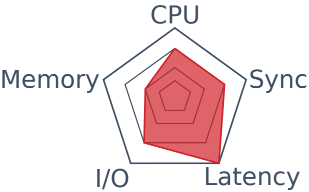


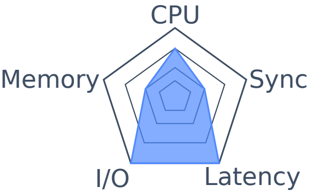


Application Requirements



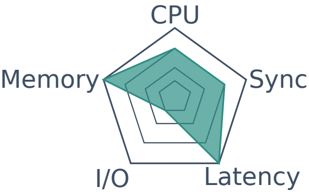


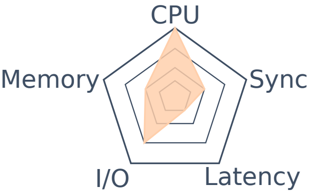




Application Requirements

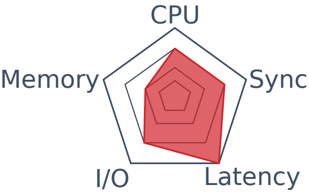
MEMCACHED

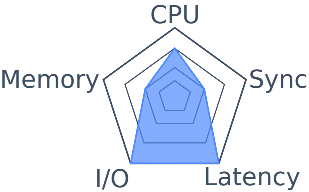




GCC

APACHE





NordVPN®

Requirements vary greatly from one application to another!

How do we satisfy these varying application requirements on all available hardware features?

Application-Specific	**General-Purpose**
✗ Impractical, requires lots of human power.	✔ Easier to maintain, ✗ but more and more complex.
We cannot make 1,000 schedulers for a 1,000 applications, but we can make 1 scheduler for a 1,000 applications. – S. Karamazov, probably	Most OSs implement a single scheduler (Linux, FreeBSD, Windows, OS X)

General Purpose Schedulers

How do we satisfy these varying application requirements on all available hardware features?

Application-Specific

✗ Impractical, requires lots of human power.

We cannot make 1,000 schedulers for a 1,000 applications, but we can make 1 scheduler for a 1,000 applications.

– S. Karamazov, probably

General-Purpose

✔ Easier to maintain,

✗ but more and more complex.

Most OSs implement a single scheduler (Linux, FreeBSD, Windows, OS X)

General Purpose Schedulers

How do we satisfy these varying application requirements on all available hardware features?

Application-Specific

✗ Impractical, requires lots of human power.

We cannot make 1,000 schedulers for a 1,000 applications, but we can make 1 scheduler for a 1,000 applications.

– S. Karamazov, probably

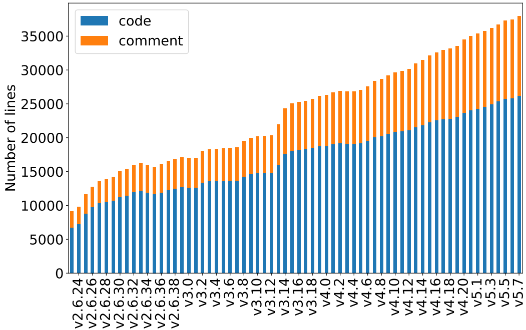
General-Purpose

✔ Easier to maintain,

✗ but more and more complex.

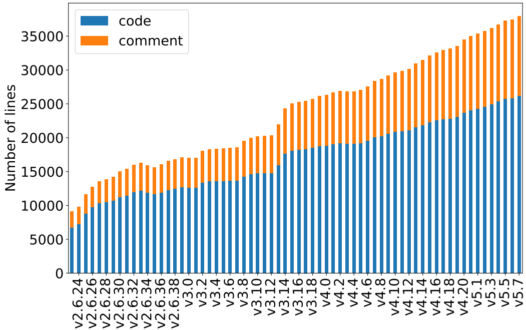
Most OSs implement a single scheduler (Linux, FreeBSD, Windows, OS X)

Limits of General-Purpose Schedulers: the CFS Example

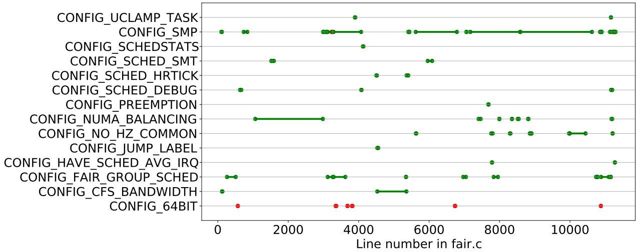


- From 6,706 to 26,213 lines of code in 13 years (×**3.9**)

Limits of General-Purpose Schedulers: the CFS Example

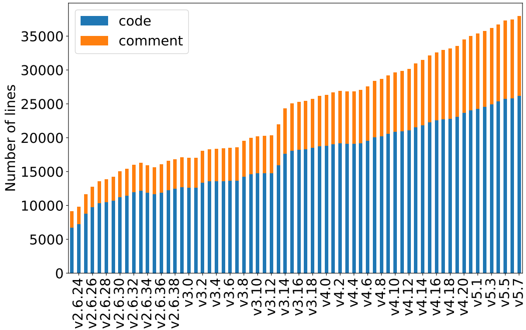


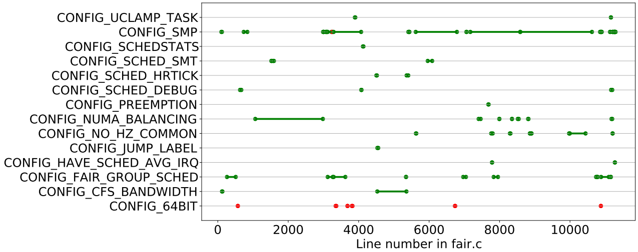
- From 6,706 to 26,213 lines of code in 13 years ($\times\mathbf{3.9}$)



- 14 configuration options $\rightarrow$ 16,384 combinations
- Features overlap and are intertwined

Limits of General-Purpose Schedulers: the CFS Example





- 14 configuration options $\rightarrow$ 16,384 combinations
- Features overlap and are intertwined

- From 6,706 to 26,213 lines of code in 13 years ($\times$**3.9**)

Maintenance and configuration are hard and impractical

What if we could build 1,000 schedulers for 1,000 applications?

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1	**Axis 2**	**Axis 3**
Scheduler Development	Performance Enhancement	Application-Specific Schedulers

What if we could build 1,000 schedulers for 1,000 applications?

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1	**Axis 2**	**Axis 3**
Scheduler Development	Performance Enhancement	Application-Specific Schedulers

What if we could build 1,000 schedulers for 1,000 applications?

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1	**Axis 2**	**Axis 3**
Scheduler Development	Performance Enhancement	Application-Specific Schedulers

What if we could build 1,000 schedulers
for 1,000 applications?

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1	**Axis 2**	**Axis 3**
Scheduler Development	Performance Enhancement	Application-Specific Schedulers

Axis 1

Scheduler Development

Writing Schedulers

Developing an efficient scheduler is a **daunting** task, with various skills needed:

- Knowledge of scheduling
- Knowledge of the underlying hardware
- Knowledge of application requirements
- Low-level programming skills

We need to ease this process
if we want new schedulers to be created!

Writing Schedulers

Developing an efficient scheduler is a **daunting** task, with various skills needed:

- Knowledge of scheduling
- Knowledge of the underlying hardware
- Knowledge of application requirements
- Low-level programming skills

We need to ease this process
if we want new schedulers to be created!

The Ipanema Tool Chain

We propose **Ipanema**, a **Domain Specific Language** for multi-core schedulers.

The **compiler** takes **Ipanema source code** and outputs two targets:

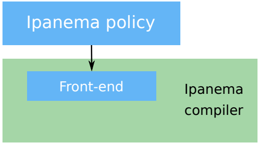
- a **C kernel module** usable in Linux,
- a **WhyML proof** used to formally verify scheduling properties.

The Ipanema Tool Chain

We propose **Ipanema**, a **Domain Specific Language** for multi-core schedulers.

The **compiler** takes **Ipanema source code** and outputs two targets:

- a **C kernel module** usable in Linux,
- a **WhyML proof** used to formally verify scheduling properties.

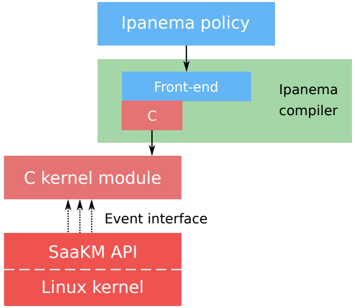


The Ipanema Tool Chain

We propose **Ipanema**, a **Domain Specific Language** for multi-core schedulers.

The **compiler** takes **Ipanema source code** and outputs two targets:

- a **C kernel module** usable in Linux,
- a **WhyML proof** used to formally verify scheduling properties.

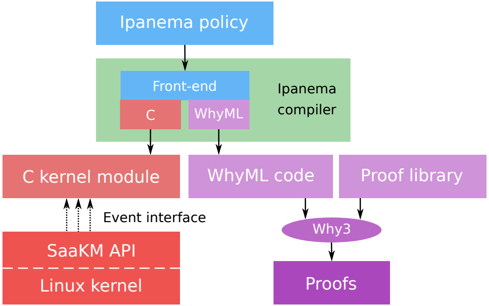


The Ipanema Tool Chain

We propose **Ipanema**, a **Domain Specific Language** for multi-core schedulers.

The **compiler** takes **Ipanema source code** and outputs two targets:

- a **C kernel module** usable in Linux,
- a **WhyML proof** used to formally verify scheduling properties.



Overview of the Ipanema DSL

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

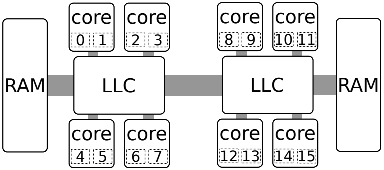
How do we account for the hardware topology?

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

How do we account for the hardware topology?



- SMT: sharing computing hardware
- LLC: sharing cache
- NUMA: memory access times may not be uniform

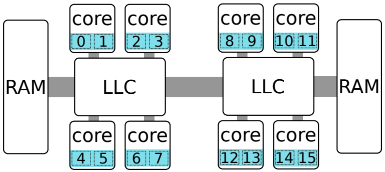
Overview of the Ipanema DSL

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

How do we account for the hardware topology?



- **SMT**: sharing computing hardware
- LLC: sharing cache
- NUMA: memory access times may not be uniform

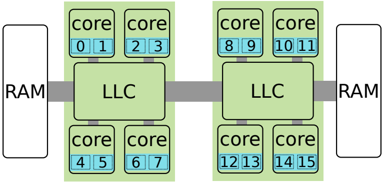
Overview of the Ipanema DSL

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

How do we account for the hardware topology?



- **SMT**: sharing computing hardware
- **LLC**: sharing cache
- **NUMA**: memory access times may not be uniform

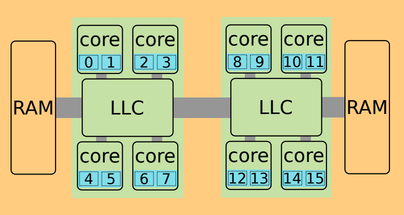
Overview of the Ipanema DSL

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

How do we account for the hardware topology?



- **SMT**: sharing computing hardware
- **LLC**: sharing cache
- **NUMA**: memory access times may not be uniform

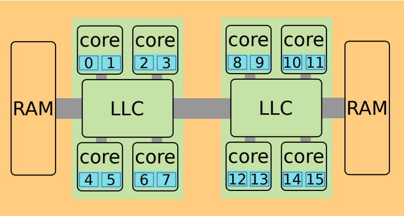
Overview of the Ipanema DSL

The **Ipanema DSL** abstracts **multi-core scheduling** for developers.

Two main features:

- **scheduling**, based on the Bossa DSL that targets single core machines [1]
- **load balancing** to even the load between cores: *(needs to be done!)*

How do we account for the hardware topology?



- **SMT**: sharing computing hardware
- **LLC**: sharing cache
- **NUMA**: memory access times may not be uniform

We need a hierarchical load balancer!

Load Balancing in Ipanema

Balancing is abstracted into **3 phases**, most of the code is **generated** by the compiler.

```
steal_for(dst):
    stealable_cores = {}
    foreach c in all_cores
        if can_steal_core(c, dst)
            stealable_cores.add(c)
```
PHASE 1

Phase 1: Finding stealable cores (lockless).
can_steal_core() is user-defined

Phase 2: Selecting the target core (lockless)
select_core() and **stop_steal** are user-defined

Phase 3: Stealing threads from target (**with locks**)
steal_thread() and **stop_steal_core** are user-defined

Minimal locking for performance and
easy to reason on for formal verification

Load Balancing in Ipanema

Balancing is abstracted into **3 phases**, most of the code is **generated** by the compiler.

```
steal_for(dst):
    stealable_cores = {}                                        PHASE 1
    foreach c in all_cores
        if can_steal_core(c, dst)
            stealable_cores.add(c)
    while !empty(stealable_cores) && !stop_steal                PHASE 2
        src = select_core(stealable_cores)
        stealable_cores.del(src)
        tmp_rq = {}
```

Phase 1: Finding stealable cores (lockless).
can_steal_core() is user-defined

Phase 2: Selecting the target core (lockless)
select_core() and **stop_steal** are user-defined

Phase 3: Stealing threads from target (**with locks**)
steal_thread() and **stop_steal_core** are user-defined

Minimal locking for performance and
easy to reason on for formal verification

Load Balancing in Ipanema

Balancing is abstracted into **3 phases**, most of the code is **generated** by the compiler.

```
steal_for(dst):
    stealable_cores = {}                                   PHASE 1
    foreach c in all_cores
        if can_steal_core(c, dst)
            stealable_cores.add(c)
    while !empty(stealable_cores) && !stop_steal           PHASE 2
        src = select_core(stealable_cores)
        stealable_cores.del(src)
        tmp_rq = {}
        foreach t in src.runqueue                  ┐        PHASE 3
            if steal_thread(t, src, dst)           │
                src.runqueue.del(t)                │ src
                tmp_rq.add(t)                      │
            if stop_steal_core                     │
                break                              ┘
        foreach t in tmp_rq                        ┐ dst
            dst.runqueue.add(t)                    ┘
```

Phase 1: Finding stealable cores (lockless).
can_steal_core() is user-defined

Phase 2: Selecting the target core (lockless)
select_core() and **stop_steal** are user-defined

Phase 3: Stealing threads from target (**with locks**)
steal_thread() and **stop_steal_core** are user-defined

Minimal locking for performance and
easy to reason on for formal verification

Load Balancing in Ipanema

Balancing is abstracted into **3 phases**, most of the code is **generated** by the compiler.

```
steal_for(dst):
    stealable_cores = {}                                   PHASE 1
    foreach c in all_cores
        if can_steal_core(c, dst)
            stealable_cores.add(c)
    while !empty(stealable_cores) && !stop_steal           PHASE 2
        src = select_core(stealable_cores)
        stealable_cores.del(src)
        tmp_rq = {}
        foreach t in src.runqueue          ┐               PHASE 3
            if steal_thread(t, src, dst)   │
                src.runqueue.del(t)        │ lock src
                tmp_rq.add(t)              │
            if stop_steal_core             │
                break                      ┘
        foreach t in tmp_rq                ┐ lock dst
            dst.runqueue.add(t)            ┘
```

Phase 1: Finding stealable cores (lockless).
can_steal_core() is user-defined

Phase 2: Selecting the target core (lockless)
select_core() and **stop_steal**
are user-defined

Phase 3: Stealing threads from target (**with locks**)
steal_thread() and **stop_steal_core**
are user-defined

Minimal locking for performance and
easy to reason on for formal verification

The Ipanema Language

Easy to learn **C-like syntax**, **7 handlers** to write (thread transitions) + balancing.

```
On unblock {
    core c = first(active_cores order = { lowest nr_tasks } );
    e.target => c.ready;
}
```

Ipanema policies are:

- **small** in Ipanema
- **smaller than CFS** in generated C code
- **standard library** with data structures and helpers (SaaKM: 1,527 lines of code)

Policy	Ipanema	C
CFS (vanilla, baseline)		5,712
CFS-CWC	360	1,006
CFS-CWC-FLAT	242	791
ULE	272	851
ULE-CWC	245	898

The Ipanema Language

Easy to learn **C-like syntax**, **7 handlers** to write (thread transitions) + balancing.

```
On unblock {
  core c = first(active_cores order = { lowest nr_tasks } );
  e.target => c.ready;
}
```

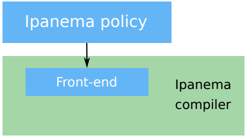
Ipanema policies are:

- **small** in Ipanema
- **smaller than CFS** in generated C code
- **standard library** with data structures and helpers (SaaKM: 1,527 lines of code)

Policy	**Ipanema**	**C**
CFS (vanilla, baseline)		5,712
CFS-CWC	360	1,006
CFS-CWC-FLAT	242	791
ULE	272	851
ULE-CWC	245	898

The **Ipanema compiler** takes **Ipanema policies** and compiles them into a **C kernel module**.

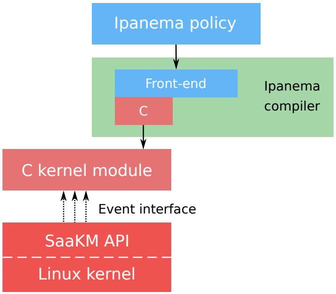
This module is inserted in a modified Linux kernel featuring **SaaKM**, the Scheduler as a Kernel Module interface.



The Ipanema Runtime System

The **Ipanema compiler** takes **Ipanema policies** and compiles them into a **C kernel module**.

This module is inserted in a modified Linux kernel featuring **SaaKM**, the Scheduler as a Kernel Module interface.



Scheduling Class VS SaaKM

Linux scheduling class API

- ✗ Designed with CFS in mind
  → **no genericity**
- ✗ Schedulers are **built-in** the kernel binary
  → hard to distribute 1,000s of schedulers
  → statically enabled
- ✗ **Poorly documented and specified**
  → hard to use

Scheduler as a Kernel Module (SaaKM)

- ✔ Mirrors basic scheduling concepts
  → close to Ipanema events
- ✔ Scheduler modules can be distributed separately
- ✔ Modules can be loaded and unloaded dynamically
- ✔ Clear specification

Scheduling Class VS SaaKM

Linux scheduling class API

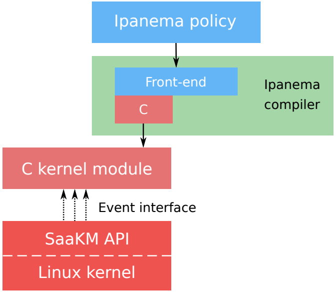
- ✗ Designed with CFS in mind
  → **no genericity**
- ✗ Schedulers are **built-in** the kernel binary
  → hard to distribute 1,000s of schedulers
  → statically enabled
- ✗ **Poorly documented and specified**
  → hard to use

Scheduler as a Kernel Module (SaaKM)

- ✔ Mirrors basic scheduling concepts
  → close to Ipanema events
- ✔ Scheduler modules can be distributed separately
- ✔ Modules can be loaded and unloaded dynamically
- ✔ Clear specification

The **Ipanema compiler** takes **Ipanema policies** and also compiles them into **WhyML code**. This code is used with **proof skeletons** and passed to the **Why3** program verification platform.

We verify **concurrent work conservation** (CWC), a weaker property than work conservation that does not require excessive locking.

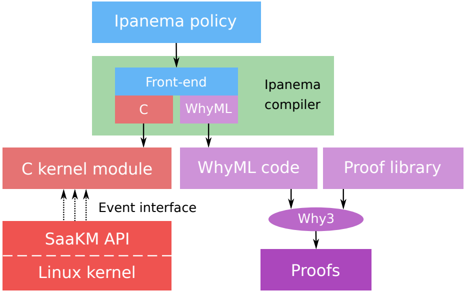


This work is the result of collaborations.

The Property Verification System

The **Ipanema compiler** takes **Ipanema policies** and also compiles them into **WhyML code**. This code is used with **proof skeletons** and passed to the **Why3** program verification platform.

We verify **concurrent work conservation** (CWC), a weaker property than work conservation that does not require excessive locking.

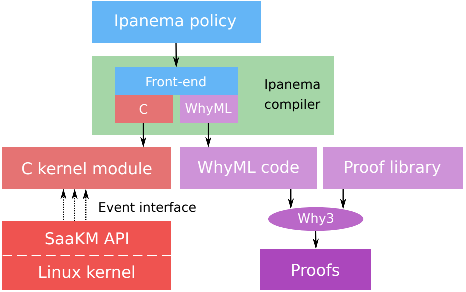


This work is the result of collaborations.

The Property Verification System

The **Ipanema compiler** takes **Ipanema policies** and also compiles them into **WhyML code**. This code is used with **proof skeletons** and passed to the **Why3** program verification platform.

We verify **concurrent work conservation** (CWC), a weaker property than work conservation that does not require excessive locking.

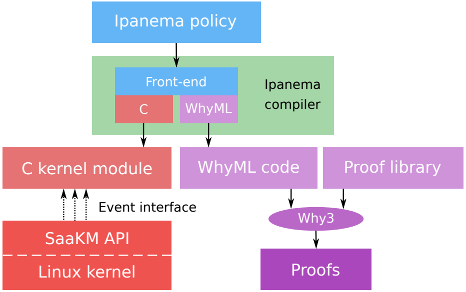


This work is the result of collaborations.

The Property Verification System

The **Ipanema compiler** takes **Ipanema policies** and also compiles them into **WhyML code**. This code is used with **proof skeletons** and passed to the **Why3** program verification platform.

We verify **concurrent work conservation** (CWC), a weaker property than work conservation that does not require excessive locking.



This work is the result of collaborations.

Evaluating the Ipanema System

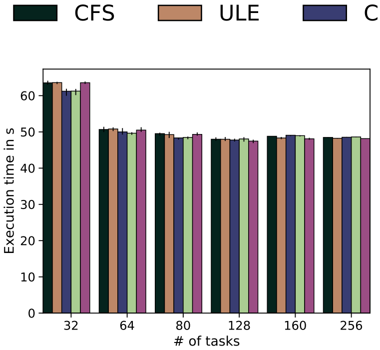
Experimental setup:

- Intel Xeon E7-8870 v4 (4 sockets, 160 cores with SMT enabled)
- 512 GiB of RAM
- OS: Debian Buster
- Kernel: Linux 4.19 with the SaaKM interface
- Applications: NAS benchmark, kernel build, OLTP with MySQL and MongoDB

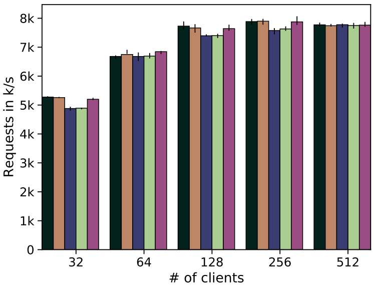
Scheduling policies:

- `CFS`: vanilla 4.19 scheduler, used as a baseline
- `CFS-CWC`: simplified and work-conserving version of CFS written in Ipanema
- `CFS-CWC-FLAT`: same as `CFS-CWC` with a flat topology
- `ULE` and `ULE-CWC`: simplified versions of the FreeBSD scheduler written in Ipanema

Evaluating the Ipanema System: Kernel Build and Database



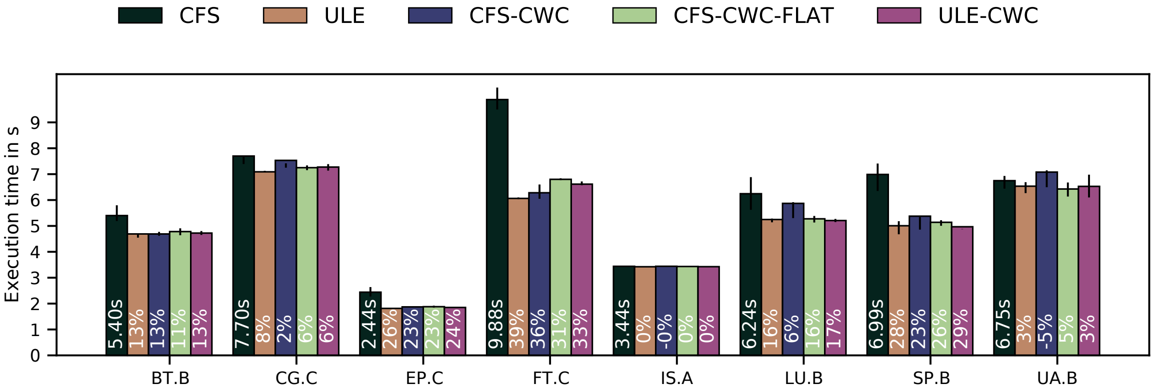
Kernel build



sysbench OLTP with MongoDB (throughput)

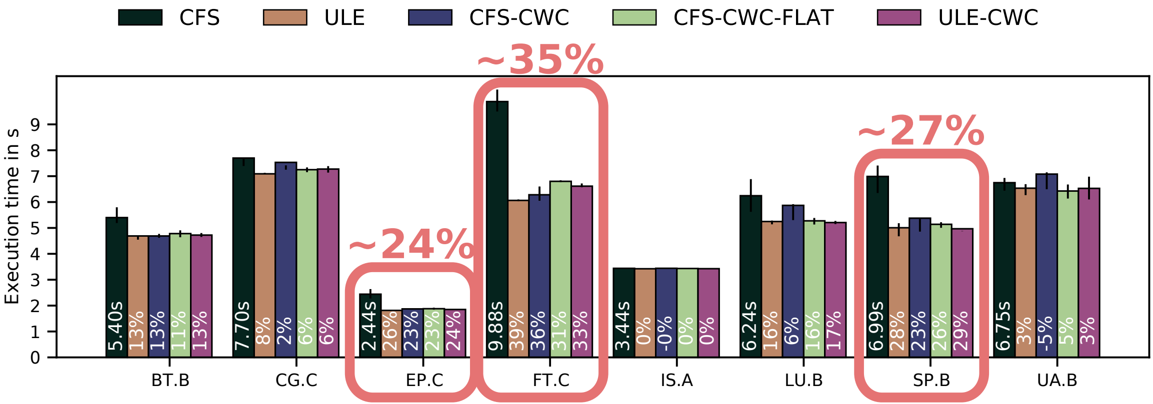
Ipanema policies are on par with CFS on these applications.

Evaluating the Ipanema System: the NAS Benchmark



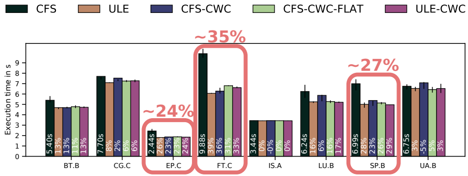
Ipanema policies outperform CFS, mostly due to work conservation.

Evaluating the Ipanema System: the NAS Benchmark



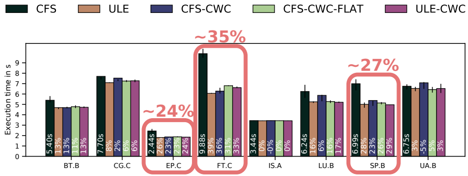
Ipanema policies outperform CFS, mostly due to work conservation.

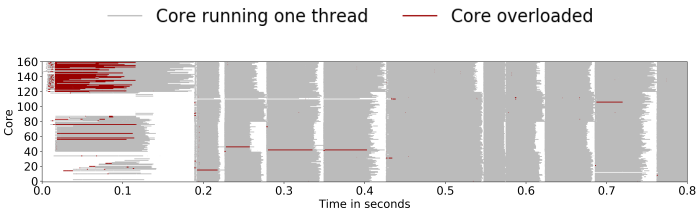
Evaluating the Ipanema System: the NAS Benchmark



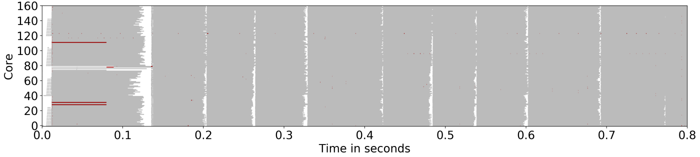
Ipanema policies outperform CFS, mostly due to work conservation.

Evaluating the Ipanema System: the NAS Benchmark





CFS



CFS-CWC

Ipanema policies outperform CFS, mostly due to work conservation.

Scheduler Development

1. **Ipanema DSL**
    - Abstracts scheduling concepts
    - Easy development, no low-level C code skills required
    - Allows the production of small efficient schedulers
2. **SaaKM interface**
    - Easy to use event-based interface
    - Scheduler hot plugging
    - Syscall and cgroup user interfaces
3. **Property verification** (collaboration)
    - The Ipanema DSL is tailored to help produce WhyML proofs
    - Proof of work conservation as an example

Perspectives

- Extend the standard library (new data structures)
- Enable the development of meta-schedulers

Axis 1: Contributions

Scheduler Development

1. **Ipanema DSL**
   - Abstracts scheduling concepts
   - Easy development, no low-level C code skills required
   - Allows the production of small efficient schedulers
2. **SaaKM interface**
   - Easy to use event-based interface
   - Scheduler hot plugging
   - Syscall and cgroup user interfaces
3. **Property verification** (collaboration)
   - The Ipanema DSL is tailored to help produce WhyML proofs
   - Proof of work conservation as an example

Perspectives

- Extend the standard library (new data structures)
- Enable the development of meta-schedulers

Scheduler Development

1. **Ipanema DSL**
   - Abstracts scheduling concepts
   - Easy development, no low-level C code skills required
   - Allows the production of small efficient schedulers
2. **SaaKM interface**
   - Easy to use event-based interface
   - Scheduler hot plugging
   - Syscall and cgroup user interfaces
3. **Property verification** (collaboration)
   - The Ipanema DSL is tailored to help produce WhyML proofs
   - Proof of work conservation as an example

Perspectives

- Extend the standard library (new data structures)
- Enable the development of meta-schedulers

Scheduler Development

1. **Ipanema DSL**
   - Abstracts scheduling concepts
   - Easy development, no low-level C code skills required
   - Allows the production of small efficient schedulers
2. **SaaKM interface**
   - Easy to use event-based interface
   - Scheduler hot plugging
   - Syscall and cgroup user interfaces
3. **Property verification** (collaboration)
   - The Ipanema DSL is tailored to help produce WhyML proofs
   - Proof of work conservation as an example

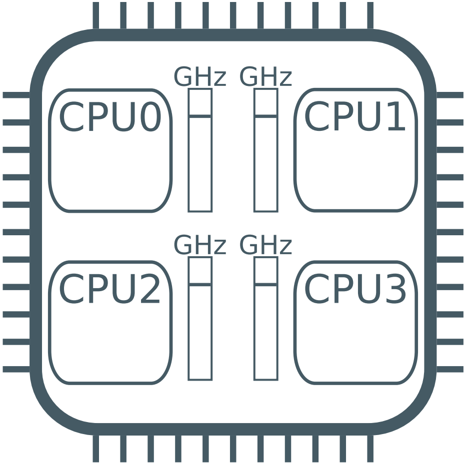
Perspectives

- Extend the standard library (new data structures)
- Enable the development of meta-schedulers

Axis 2

Performance Enhancement

Dynamic Frequency Scaling

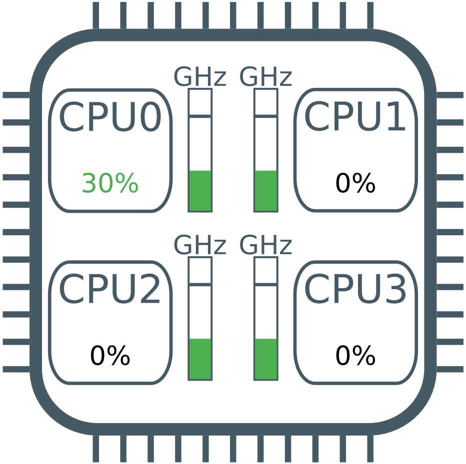


The frequency of a CPU:

- depends on the load
- is managed at the chip level
  $\Rightarrow$ the load of **one** core impacts the frequency of **all** cores on the chip

When all CPUs are fully loaded, **nominal frequency** is guaranteed.

Dynamic Frequency Scaling

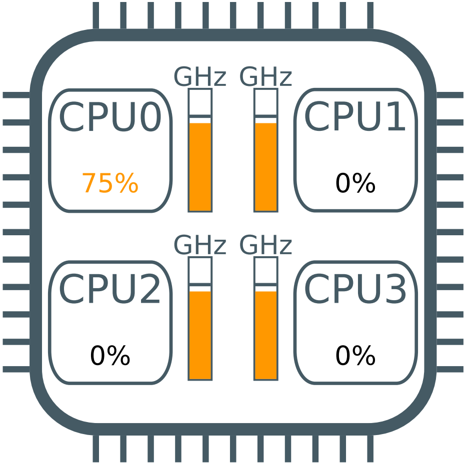


The frequency of a CPU:

- depends on the load
- is managed at the chip level
  ⇒ the load of **one** core impacts the frequency of **all** cores on the chip

When all CPUs are fully loaded, **nominal frequency** is guaranteed.

Dynamic Frequency Scaling

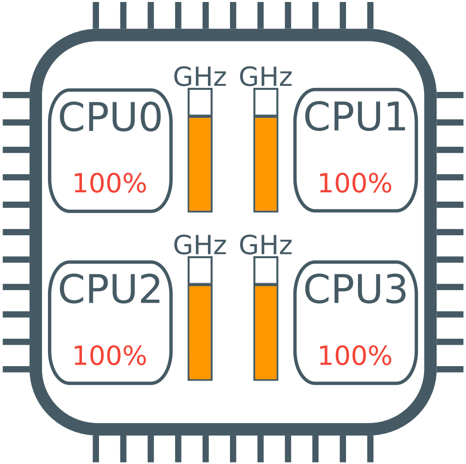


The frequency of a CPU:

- depends on the load
- is managed at the chip level
  $\Rightarrow$ the load of **one** core impacts the frequency of **all** cores on the chip

When all CPUs are fully loaded, **nominal frequency** is guaranteed.

Dynamic Frequency Scaling

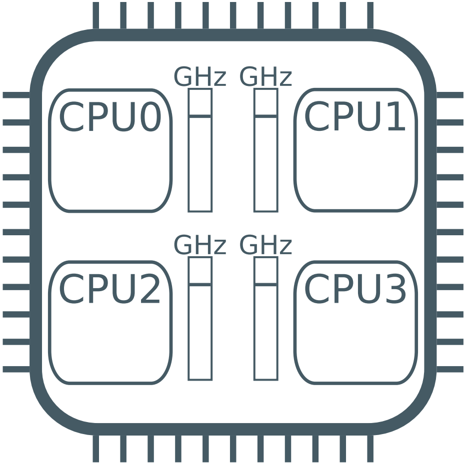


The frequency of a CPU:

- depends on the load
- is managed at the chip level
  $\Rightarrow$ the load of **one** core impacts the frequency of **all** cores on the chip

When all CPUs are fully loaded, **nominal frequency** is guaranteed.

Dynamic Frequency Scaling on Modern Processors



On modern chips, frequency is managed **per core**:

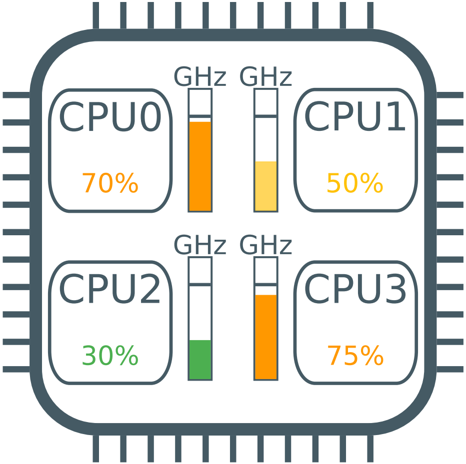
- Intel Cascade Lake (2019)
- AMD Ryzen (2019)

Each core sets **its** frequency to match **its** load. **Idle cores** run at the **minimal frequency** while busy cores use higher frequencies.

⇒ Energy savings

Turbo mode: when some cores not active, busy cores can use even higher frequencies

Dynamic Frequency Scaling on Modern Processors



On modern chips, frequency is managed **per core**:

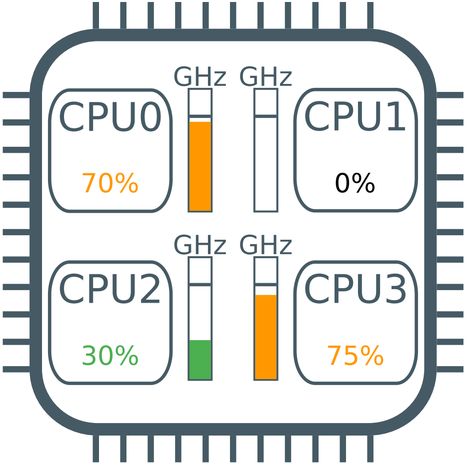
- Intel Cascade Lake (2019)
- AMD Ryzen (2019)

Each core sets **its** frequency to match **its** load.
Idle cores run at the **minimal frequency** while busy cores use higher frequencies.

$\Rightarrow$ Energy savings

Turbo mode: when some cores not active, busy cores can use even higher frequencies

Dynamic Frequency Scaling on Modern Processors



On modern chips, frequency is managed **per core**:

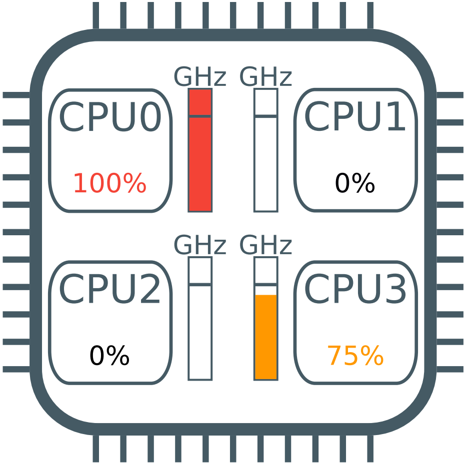
- Intel Cascade Lake (2019)
- AMD Ryzen (2019)

Each core sets **its** frequency to match **its** load. **Idle cores** run at the **minimal frequency** while busy cores use higher frequencies.

$\Rightarrow$ Energy savings

Turbo mode: when some cores not active, busy cores can use even higher frequencies

Dynamic Frequency Scaling on Modern Processors



On modern chips, frequency is managed **per core**:

- Intel Cascade Lake (2019)
- AMD Ryzen (2019)

Each core sets **its** frequency to match **its** load. **Idle cores** run at the **minimal frequency** while busy cores use higher frequencies.

$\Rightarrow$ Energy savings

Turbo mode: when some cores not active, busy cores can use even higher frequencies

Frequency and Scheduling

Change the frequency to match the load

- Linux *scaling governors* (ondemand, schedutil)
- hardware frequency scaling (e.g. Intel HWP)

Frequency scaling used to

- maximize the instructions per joule metric (Weiser'94 [2])
- reduce contention on shared hardware (Merkel'10 [3], Zhang'10 [4])
- reduce energy usage (Bianchini'03 [5])

Recent work by the Linux scheduler community

- TurboSched: placing small jitter tasks on Turbo cores
- support for heterogeneous architectures (ARM big.LITTLE, Intel Hybrid)

Frequency and Scheduling

Change the frequency to match the load

- Linux *scaling governors* (ondemand, schedutil)
- hardware frequency scaling (e.g. Intel HWP)

Frequency scaling used to

- maximize the instructions per joule metric (Weiser'94 [2])
- reduce contention on shared hardware (Merkel'10 [3], Zhang'10 [4])
- reduce energy usage (Bianchini'03 [5])

Recent work by the Linux scheduler community

- TurboSched: placing small jitter tasks on Turbo cores
- support for heterogeneous architectures (ARM big.LITTLE, Intel Hybrid)

Frequency and Scheduling

Change the frequency to match the load

- Linux *scaling governors* (ondemand, schedutil)
- hardware frequency scaling (e.g. Intel HWP)

Frequency scaling used to

- maximize the instructions per joule metric (Weiser'94 [2])
- reduce contention on shared hardware (Merkel'10 [3], Zhang'10 [4])
- reduce energy usage (Bianchini'03 [5])

Recent work by the Linux scheduler community

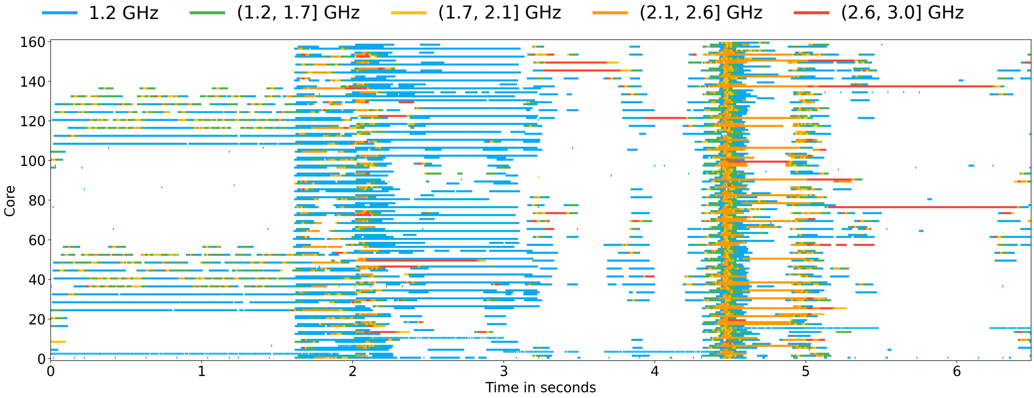
- TurboSched: placing small jitter tasks on Turbo cores
- support for heterogeneous architectures (ARM big.LITTLE, Intel Hybrid)

Case Study: Compiling the Linux Scheduler

We develop **high-resolution monitoring tools** for the scheduler. Example: **frequency**.

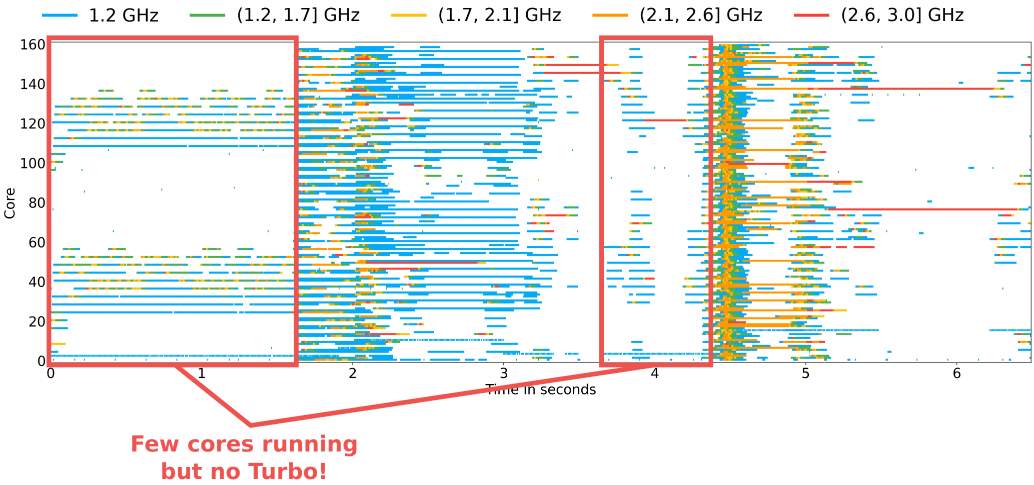
Case Study: Compiling the Linux Scheduler

We develop **high-resolution monitoring tools** for the scheduler. Example: **frequency**.



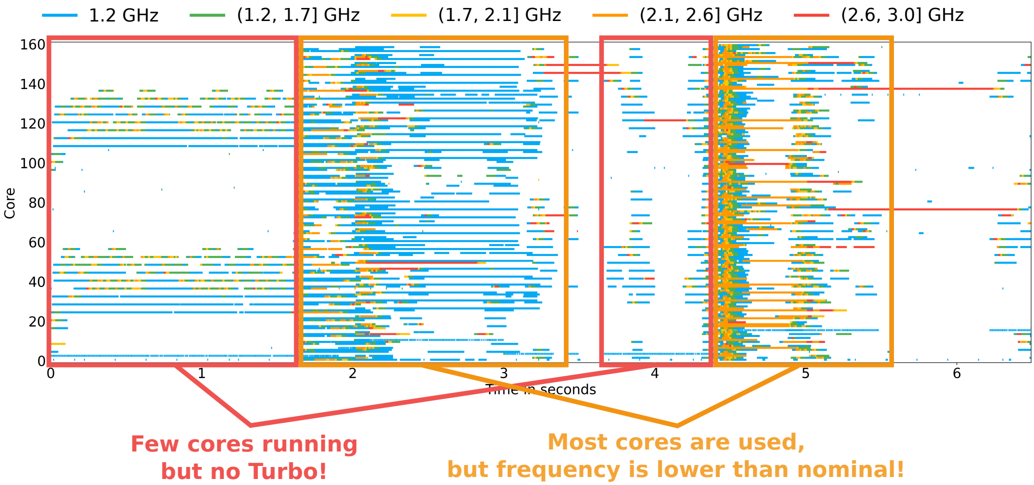
Case Study: Compiling the Linux Scheduler

We develop **high-resolution monitoring tools** for the scheduler. Example: **frequency**.

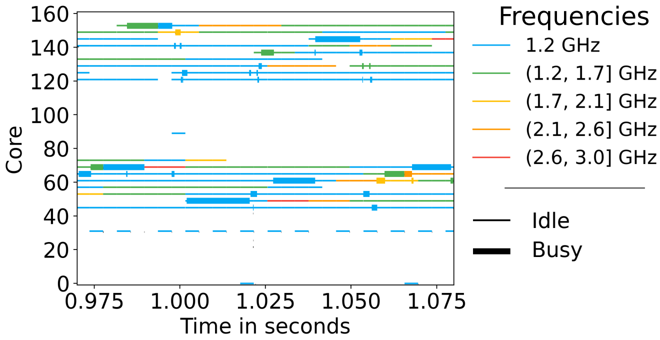


Case Study: Compiling the Linux Scheduler

We develop **high-resolution monitoring tools** for the scheduler. Example: **frequency**.

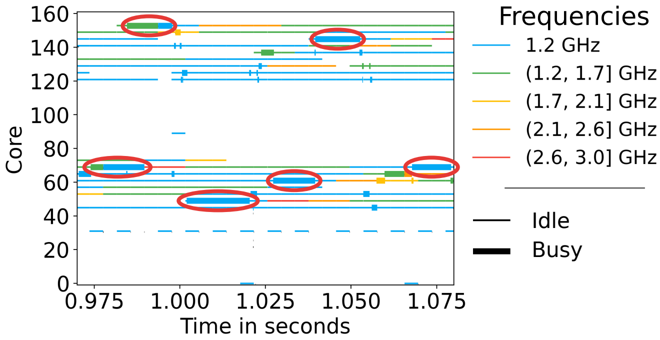


Case Study: Zooming in the Trace



Case Study: Zooming in the Trace

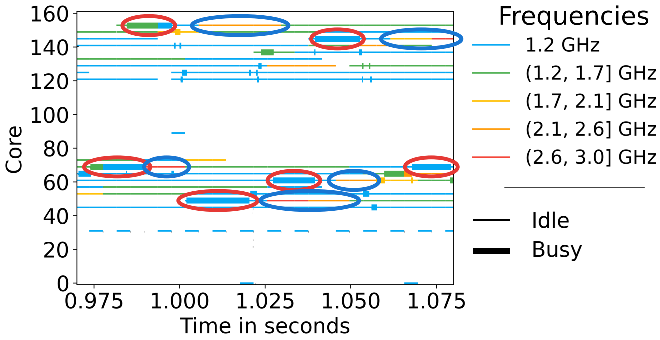
Busy at low frequency



Case Study: Zooming in the Trace

Busy at low frequency

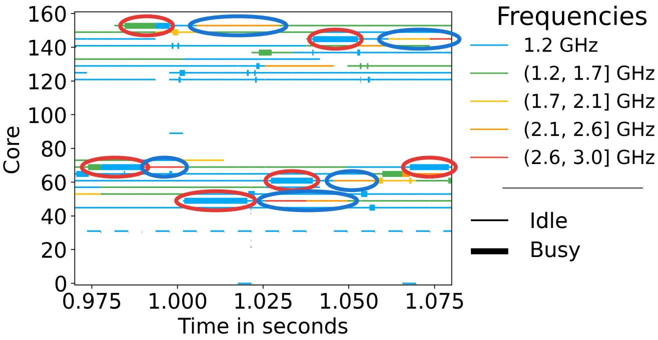
Idle at high frequency



Case Study: Zooming in the Trace

Busy at low frequency
Idle at high frequency

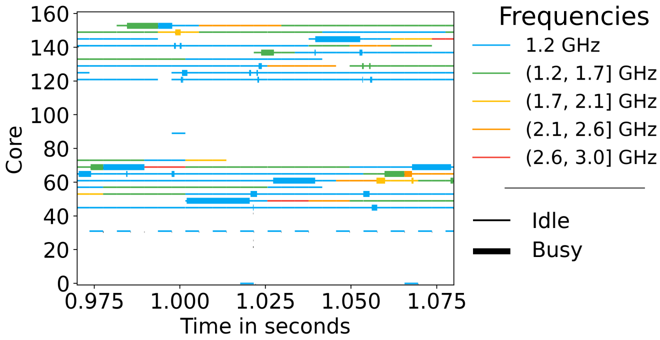
Frequency scaling is too late to be effective!



Case Study: Zooming in the Trace

Busy at low frequency
Idle at high frequency

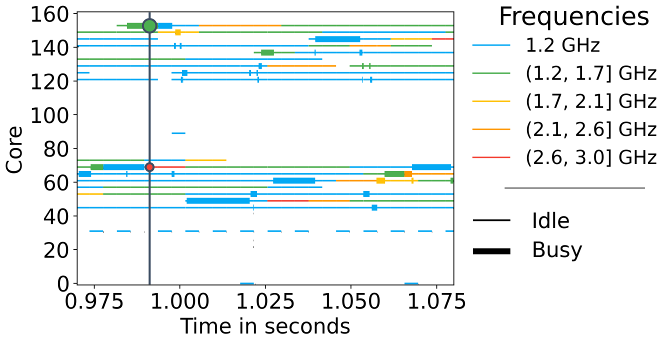
Frequency scaling is too late to be effective!



Case Study: Zooming in the Trace

Busy at low frequency
Idle at high frequency

Frequency scaling is too late to be effective!



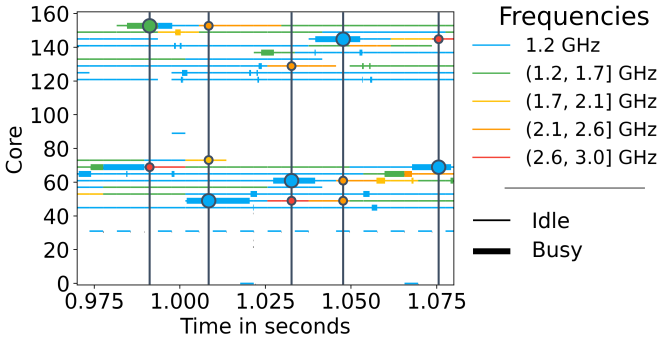
Case Study: Zooming in the Trace

Busy at low frequency
Idle at high frequency

Frequency scaling is too late to be effective!

Busy, low frequency
Idle, high frequency

Better suited cores are available!



Frequency Transition Latency

We develop a **tool** to measure the **Frequency Transition Latency (FTL)**:

Latency between a change of load and the corresponding change of frequency.

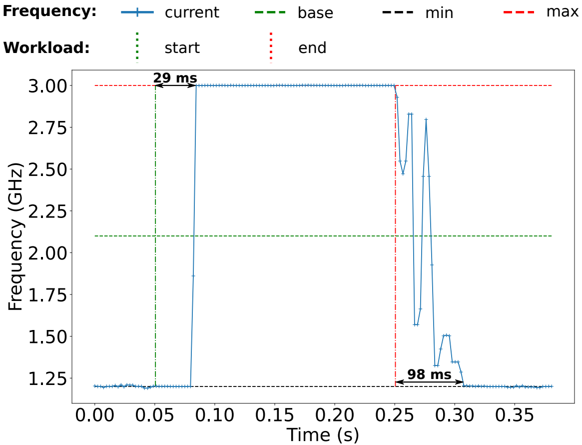
Frequency Transition Latency

We develop a **tool** to measure the **Frequency Transition Latency (FTL)**:

Latency between a change of load and the corresponding change of frequency.

Infinite loop on a single core,
from idleness to 100% load, resp.
from min to max frequency.

0%	→	100%	:	**29 ms**
100%	→	0%	:	**98 ms**



Frequency Transition Latency

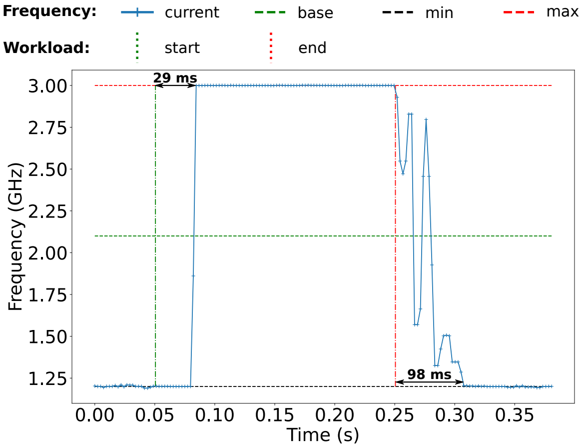
We develop a **tool** to measure the **Frequency Transition Latency (FTL)**:

Latency between a change of load and the corresponding change of frequency.

Infinite loop on a single core,
from idleness to 100% load, resp.
from min to max frequency.

0%	$\rightarrow$	100%	:	**29 ms**
100%	$\rightarrow$	0%	:	**98 ms**

Changing frequency is not instantaneous!

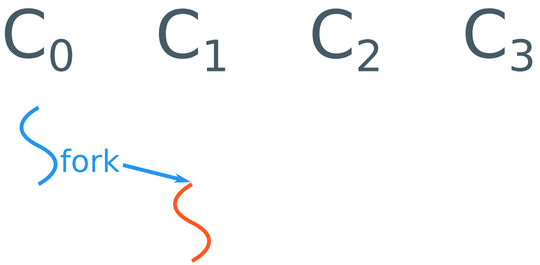


CFS and the Fork/Wait Pattern

C_0 C_1 C_2 C_3

CFS and the Fork/Wait Pattern

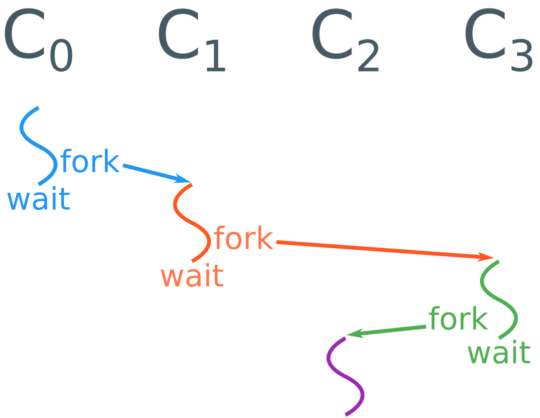
CFS tries to be **work conserving**
→ new and waking threads are placed on **idle** cores if available



CFS and the Fork/Wait Pattern

CFS tries to be **work conserving**
→ new and waking threads are placed on **idle** cores if available

In our case study, we have a repeated **fork/wait** pattern.

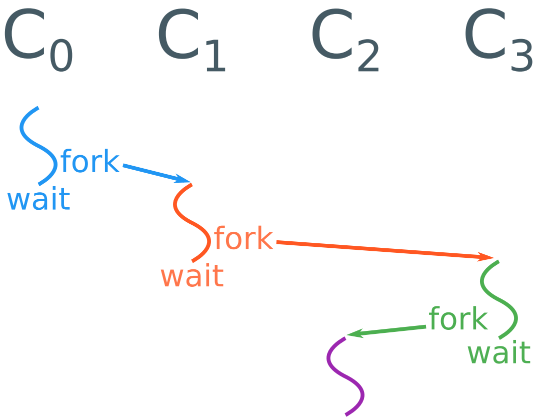


CFS and the Fork/Wait Pattern

CFS tries to be **work conserving**
→ new and waking threads are placed on **idle** cores if available

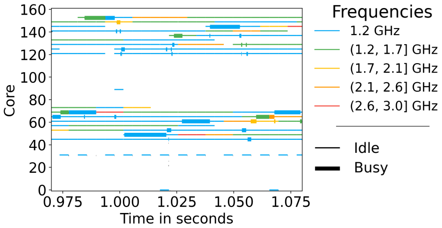
In our case study, we have a repeated **fork/wait** pattern.

A **sequential** workload uses **multiple CPUs**!



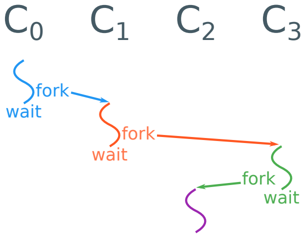
Problem: Frequency Inversion

Long FTLs



+

Work conserving scheduler



=

The frequencies at which two cores operate are inverted as compared to their load

Solution: Delayed Thread Migration

We propose $\mathbf{S_{move}}$: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

When the timer is **triggered** 50 μs later, we migrate the **child thread** to C_1.

We only lose 50 μs compared to CFS.
Without the timer, periodic load balancing would have fixed this situation in tens of milliseconds.

C_0 C_1

Solution: Delayed Thread Migration

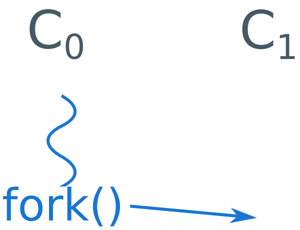
We propose **S_{move}**: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

When the timer is **triggered** 50 μs later, we migrate the **child thread** to C_1.

We only lose 50 μs compared to CFS.
Without the timer, periodic load balancing would have fixed this situation in tens of milliseconds.



Solution: Delayed Thread Migration

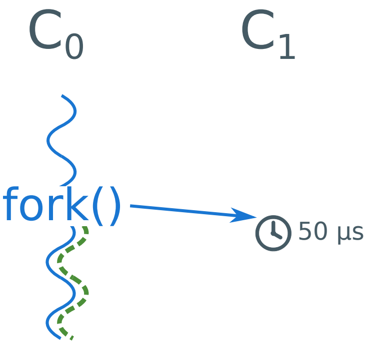
We propose $\mathbf{S_{move}}$: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

When the timer is **triggered** 50 μs later, we migrate the **child thread** to C_1.

We only lose 50 μs compared to CFS.
Without the timer, periodic load balancing would have fixed this situation in tens of milliseconds.



Solution: Delayed Thread Migration

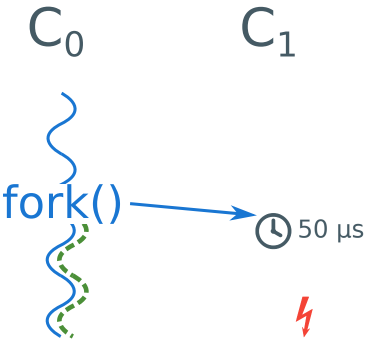
We propose **S_{move}**: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

When the timer is **triggered** 50 μs later, we migrate the **child thread** to C_1.

We only lose 50 μs compared to CFS.
Without the timer, periodic load balancing would have fixed this situation in tens of milliseconds.



Solution: Delayed Thread Migration

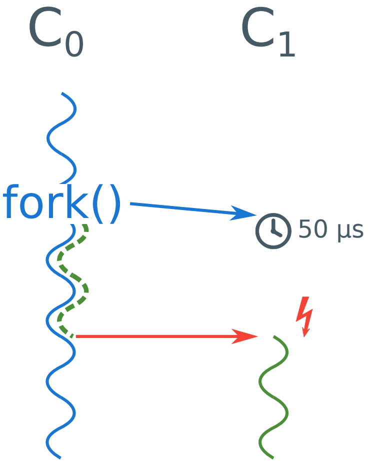
We propose $\mathbf{S_{move}}$: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

When the timer is **triggered** 50 μs later, we migrate the **child thread** to C_1.

We only lose 50 μs compared to CFS.
Without the timer, periodic load balancing would have fixed this situation in tens of milliseconds.



Solution: Delayed Thread Migration (2)

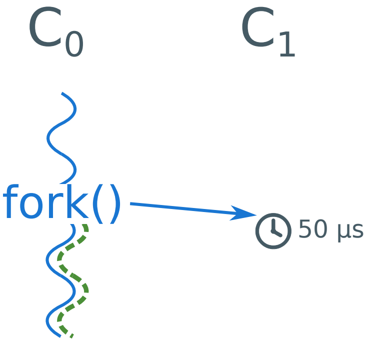
We propose $\mathbf{S_{move}}$: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

Parent thread calls the `wait` syscall, the **child thread** is scheduled on C_0, the timer is **canceled**.

The sequential program uses a single core, running at a **high frequency** and C_1 stays **idle**.



Solution: Delayed Thread Migration (2)

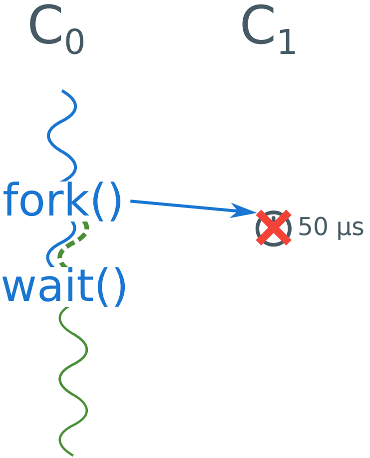
We propose **S_{move}**: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

Parent thread calls the `wait` syscall, the **child thread** is scheduled on C_0, the timer is **canceled**.

The sequential program uses a single core, running at a **high frequency** and C_1 stays **idle**.



Solution: Delayed Thread Migration (2)

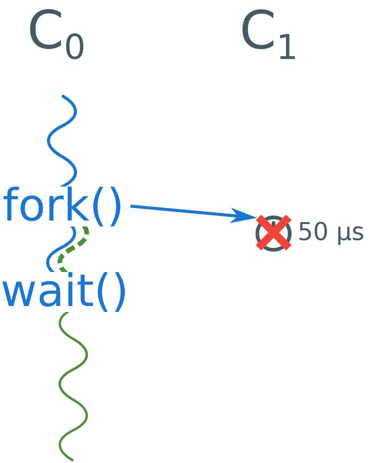
We propose $\mathbf{S_{move}}$: delaying thread migrations on fork/wakeup.

Parent thread runs on C_0, calls the `fork()` syscall. CFS decides to place **child thread** on C_1.

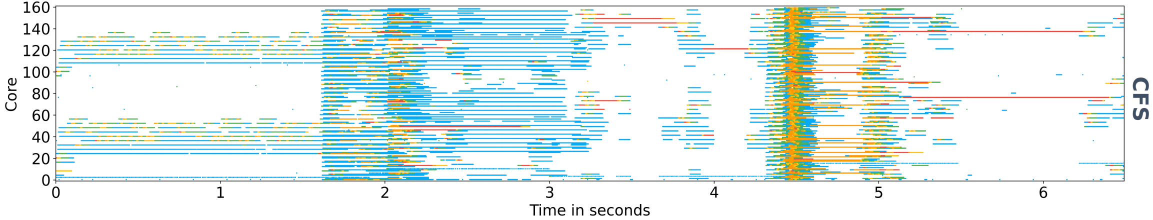
If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50 μs and place the **child thread** on C_0.

Parent thread calls the `wait` syscall, the **child thread** is scheduled on C_0, the timer is **canceled**.

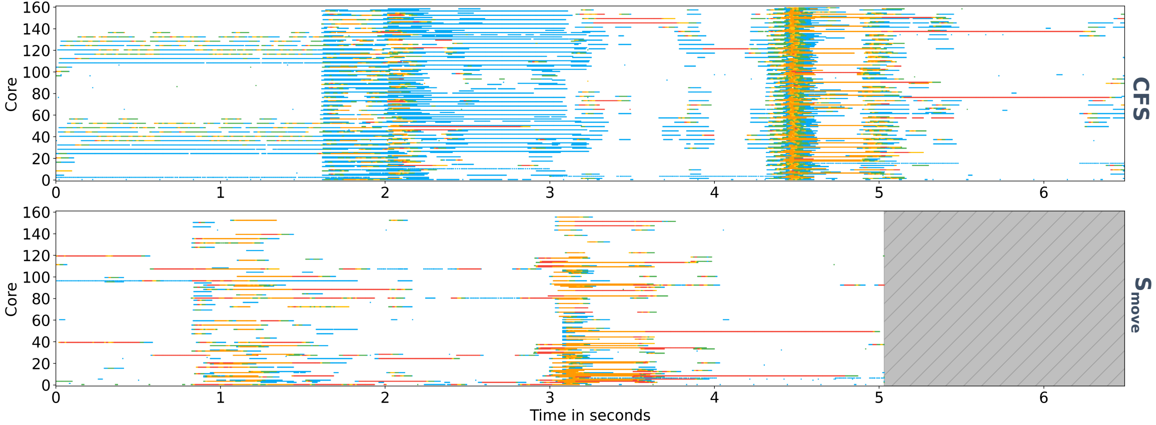
The sequential program uses a single core, running at a **high frequency** and C_1 stays **idle**.



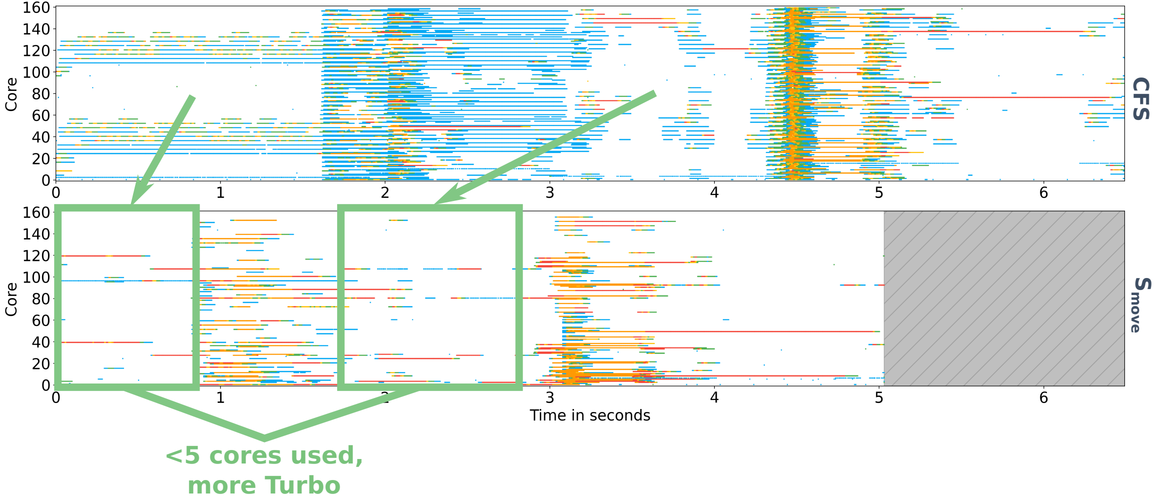
Solution: Delayed Thread Migration Evaluation



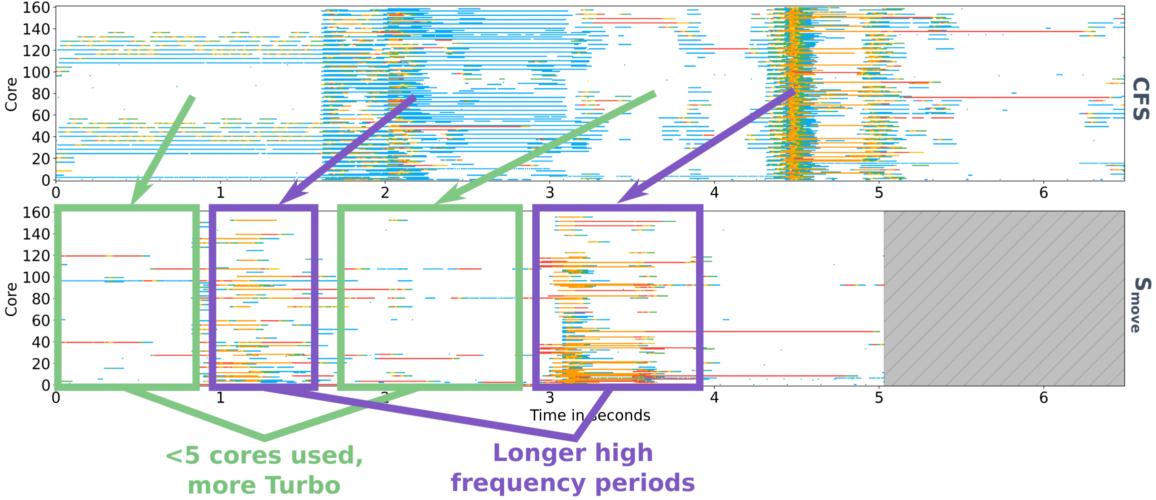
Solution: Delayed Thread Migration Evaluation



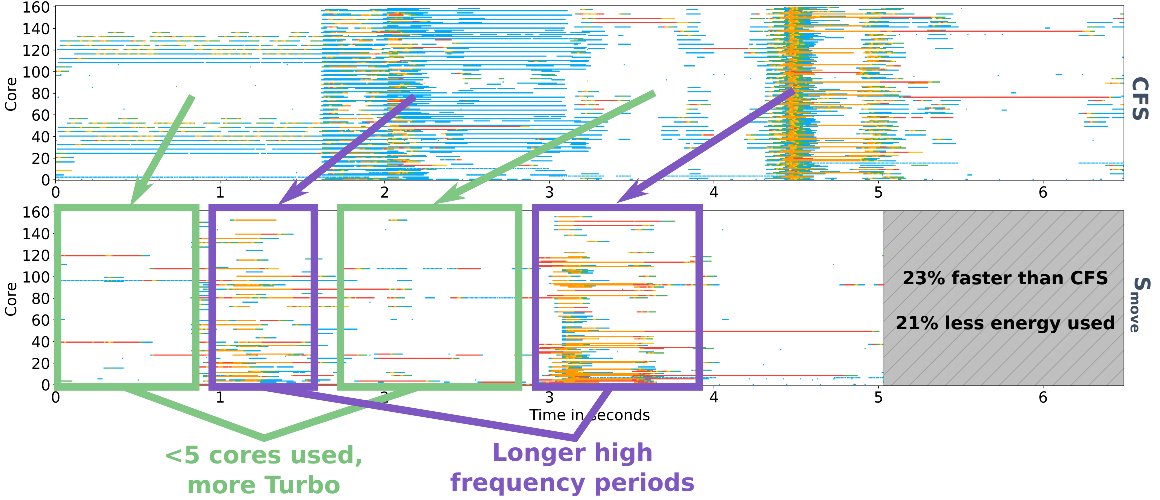
Solution: Delayed Thread Migration Evaluation



Solution: Delayed Thread Migration Evaluation



Solution: Delayed Thread Migration Evaluation



Performance and Energy Evaluation

Hardware:

- **Server**: 80-core Intel Xeon E7-8870 v4 (160 HW threads)
- **Desktop**: 4-core AMD Ryzen 5 3400G (8 HW threads)

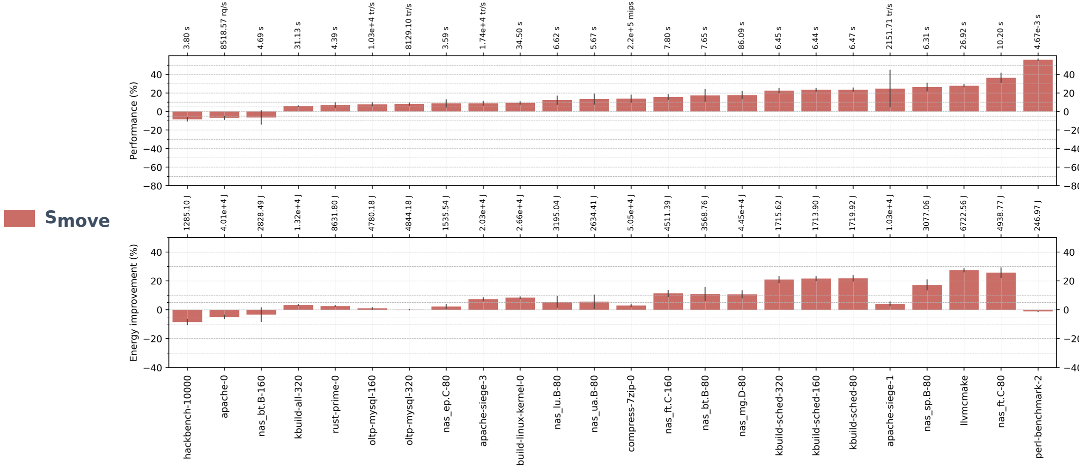
Benchmarks: 60 applications from

- NAS: HPC applications
- Phoronix: web servers, compilations, DNN libs, compression, databases, ...
- hackbench and sysbench OLTP

Frequency scaling governors:

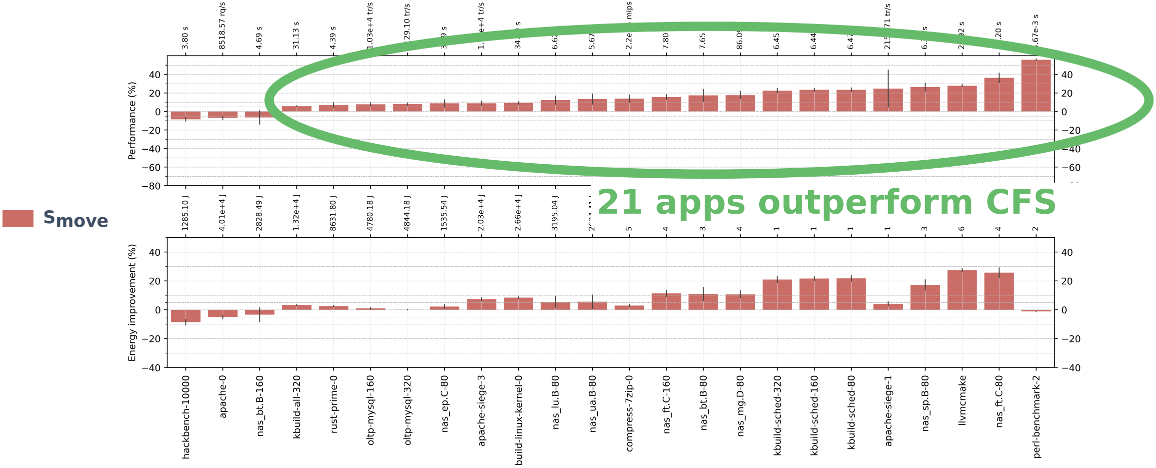
- powersave
- schedutil

Compared to CFS, server machine, powersave governor, higher is better

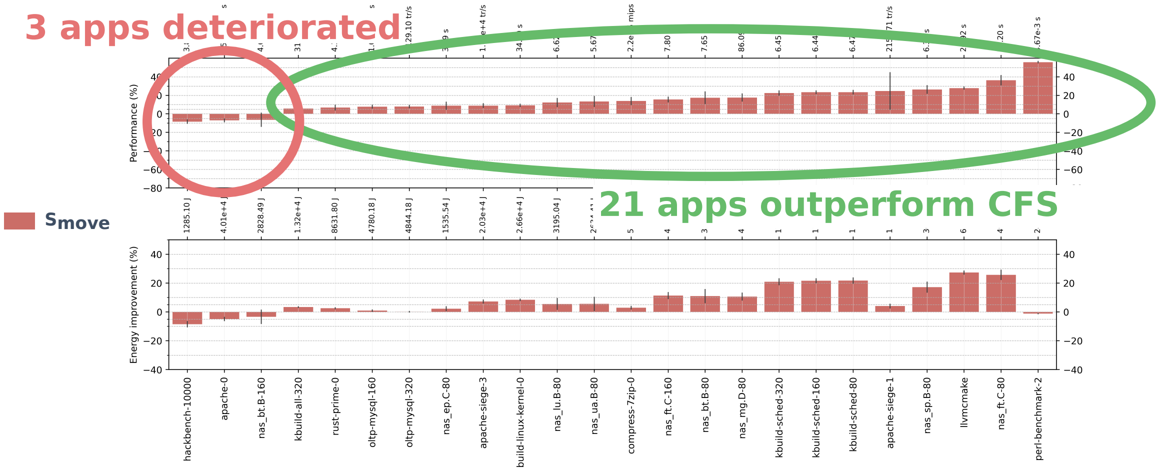


Performance and Energy Evaluation (2)

Compared to CFS, server machine, powersave governor, higher is better

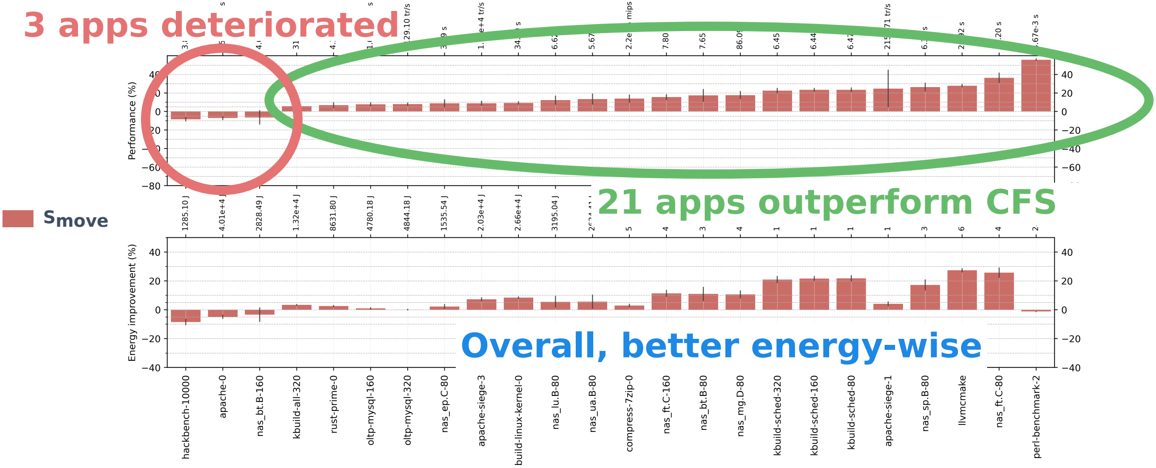


Compared to CFS, server machine, powersave governor, higher is better



Performance and Energy Evaluation (2)

Compared to CFS, server machine, powersave governor, higher is better



Performance Enhancement

1. **Monitoring tools** for the scheduler subsystem
2. Discovery of the **frequency inversion problem**
   - Long FTLs + work conserving scheduler
   - New problem with per-core dynamic frequency scaling
3. **Two solutions implemented in Linux**
   - S_{local}: simple, aggressive, relies on load balancing
   - S_{move}: frequency-aware, efficient. Submitted to the Linux kernel community

Perspectives

- Fully frequency-aware scheduler
- Modeling the frequency behavior of CPUs (active cores, temperature, instruction set, . . .)
- Shortening the FTL with faster reconfiguration (hardware, scaling governor)

Performance Enhancement

1. **Monitoring tools** for the scheduler subsystem
2. Discovery of the **frequency inversion problem**
   - Long FTLs + work conserving scheduler
   - New problem with per-core dynamic frequency scaling
3. **Two solutions implemented in Linux**
   - S_{local}: simple, aggressive, relies on load balancing
   - S_{move}: frequency-aware, efficient. Submitted to the Linux kernel community

Perspectives

- Fully frequency-aware scheduler
- Modeling the frequency behavior of CPUs (active cores, temperature, instruction set, ...)
- Shortening the FTL with faster reconfiguration (hardware, scaling governor)

Axis 2: Contributions

Performance Enhancement

1. **Monitoring tools** for the scheduler subsystem
2. Discovery of the **frequency inversion problem**
   - Long FTLs + work conserving scheduler
   - New problem with per-core dynamic frequency scaling
3. **Two solutions implemented in Linux**
   - S_{local}: simple, aggressive, relies on load balancing
   - S_{move}: frequency-aware, efficient. Submitted to the Linux kernel community

Perspectives

- Fully frequency-aware scheduler
- Modeling the frequency behavior of CPUs (active cores, temperature, instruction set, ...)
- Shortening the FTL with faster reconfiguration (hardware, scaling governor)

Performance Enhancement

1. **Monitoring tools** for the scheduler subsystem
2. Discovery of the **frequency inversion problem**
   - Long FTLs + work conserving scheduler
   - New problem with per-core dynamic frequency scaling
3. **Two solutions implemented in Linux**
   - S_{local}: simple, aggressive, relies on load balancing
   - S_{move}: frequency-aware, efficient. Submitted to the Linux kernel community

Perspectives

- Fully frequency-aware scheduler
- Modeling the frequency behavior of CPUs (active cores, temperature, instruction set, ...)
- Shortening the FTL with faster reconfiguration (hardware, scaling governor)

Axis 3

Application-Specific Schedulers

Fifty Shades of Scheduling

Scheduling comes in various flavors:

- fair (CFS, ULE)
- enforce real-time deadlines (EDF)
- optimize data locality
- reduce contention on caches, memory, disks, . . .
- and so on . . .

One Scheduler Cannot Rule Them All . . .

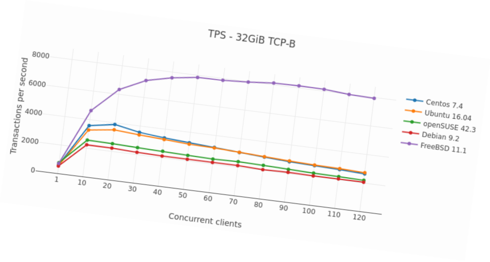
Existing studies show varying levels of performance depending on the application.

- blog posts from users (e.g. PostgreSQL)
- comparisons from the benchmarking community (e.g. Phoronix)
- academic results [Bouron'18]

One Scheduler Cannot Rule Them All . . .

Existing studies show varying levels of performance depending on the application.

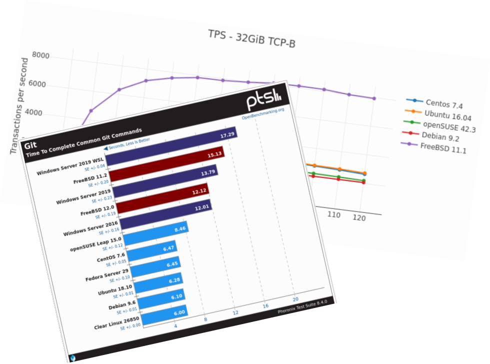
- blog posts from users (e.g. PostgreSQL)
- comparisons from the benchmarking community (e.g. Phoronix)
- academic results [Bouron'18]



One Scheduler Cannot Rule Them All . . .

Existing studies show varying levels of performance depending on the application.

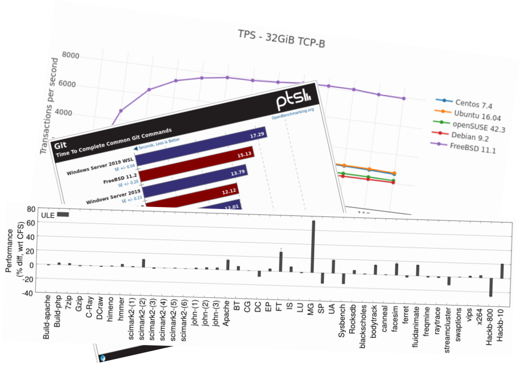
- blog posts from users (e.g. PostgreSQL)
- comparisons from the benchmarking community (e.g. Phoronix)
- academic results [Bouron'18]



One Scheduler Cannot Rule Them All . . .

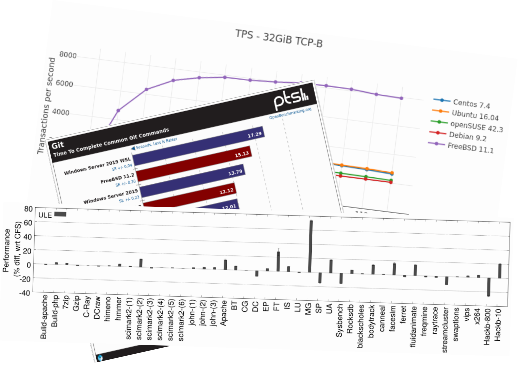
Existing studies show varying levels of performance depending on the application.

- blog posts from users (e.g. PostgreSQL)
- comparisons from the benchmarking community (e.g. Phoronix)
- academic results [Bouron'18]



Existing studies show varying levels of performance depending on the application.

- blog posts from users (e.g. PostgreSQL)
- comparisons from the benchmarking community (e.g. Phoronix)
- academic results [Bouron'18]



There is no silver bullet in scheduling!

Building Application-Specific Schedulers

Could we leverage Ipanema and SaaKM to write application-specific schedulers?

We propose the following approach:

- Develop a **feature-oriented model** of schedulers
- Implement a **library of features** to build modular schedulers
- Propose an **evaluation methodology** for these produced schedulers
- Develop techniques to **automatically build the best application-specific scheduler**

Building Application-Specific Schedulers

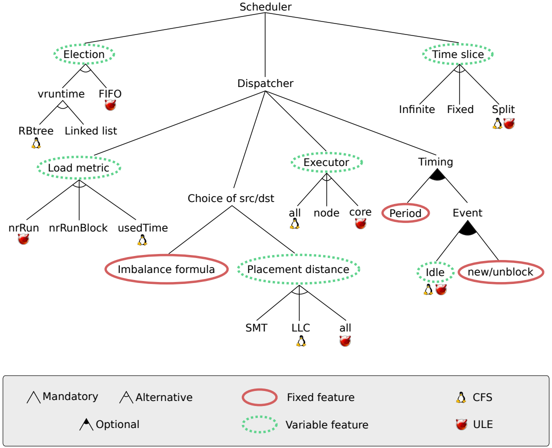
Could we leverage Ipanema and SaaKM to write application-specific schedulers?

We propose the following approach:

- Develop a **feature-oriented model** of schedulers
- Implement a **library of features** to build modular schedulers
- Propose an **evaluation methodology** for these produced schedulers
- Develop techniques to **automatically build the best application-specific scheduler**

Scheduler Feature Model

Model



Implementation

Implemented as a **kernel library**.

SaaKM compliant.

Features are **independent** from each other.

16 features in current model
→ **486 combinations** can be generated.

Experimental Setup

Hardware:

- CPU: Intel Xeon E5645 (12 cores, 24 HW threads, 2 sockets)
- RAM: 64 GiB
- OS: Debian 8 with Linux 4.19 kernel

Applications:

- 7 *PARSEC* applications
- 7 *Phoronix* applications
- 2 *HiBench* applications
- 3 *sysbench* applications
- *hackbench* from the Linux Test Project

Each application is run **10 times** with each scheduler

Total experiment time of **1,925 hours**, distributed on 8 identical machines

Experimental Setup

Hardware:

- CPU: Intel Xeon E5645 (12 cores, 24 HW threads, 2 sockets)
- RAM: 64 GiB
- OS: Debian 8 with Linux 4.19 kernel

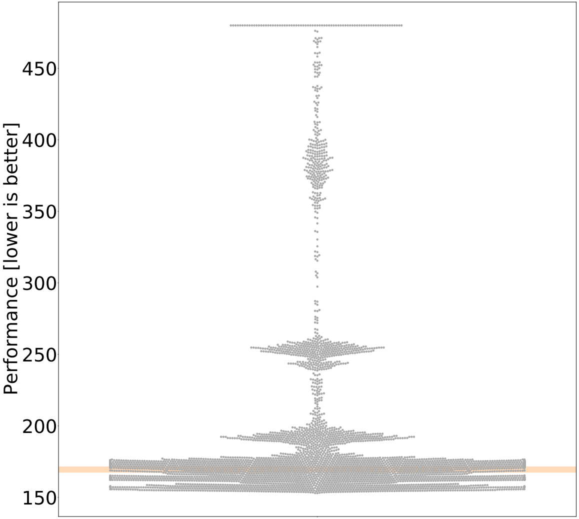
Applications:

- 7 *PARSEC* applications
- 7 *Phoronix* applications
- 2 *HiBench* applications
- 3 *sysbench* applications
- *hackbench* from the Linux Test Project

Each application is run **10 times** with each scheduler

Total experiment time of **1,925 hours**, distributed on 8 identical machines

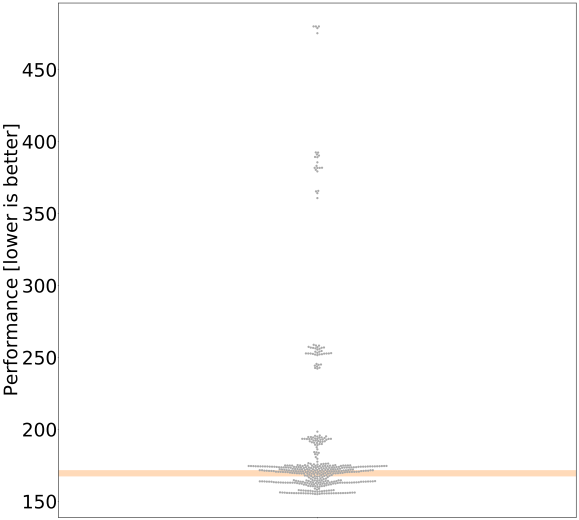
Feature Evaluation: the `facesim` Example



Initial state: (*facesim application*)

- 10 runs × 486 schedulers = **4,860 points**
- **CFS** as a baseline, in the background.

Feature Evaluation: the `facesim` Example



Initial state: (*facesim application*)

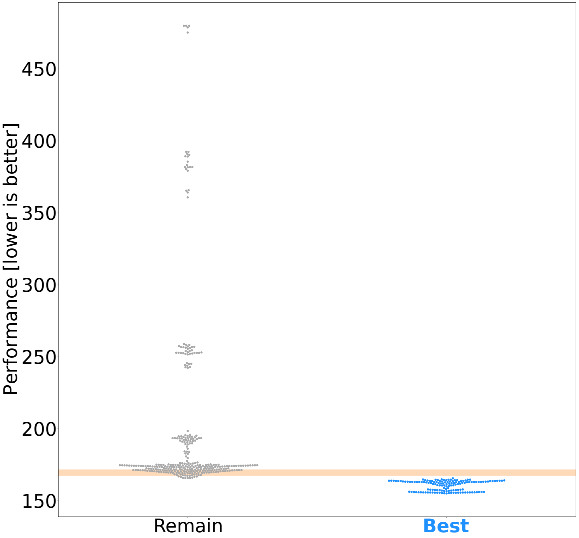
10 runs × 486 schedulers = **4,860 points**

CFS as a baseline, in the background.

Reducing data:

1. Discard unstable schedulers (*stddev*)
2. Reduce to mean value

Feature Evaluation: the `facesim` Example



Initial state: (*facesim application*)

10 runs × 486 schedulers = **4,860 points**

CFS as a baseline, in the background.

Reducing data:

1. Discard unstable schedulers (*stddev*)
2. Reduce to mean value

Finding the best:

Isolate the schedulers at most 10% from the best one

⇒ Set of **Best** schedulers

Preliminary Results

Raw results confirm the **value of building application-specific schedulers**:

Out of 20 applications:

- For **17 applications (85%)**, we build simple schedulers **on par** with CFS
- For **7 applications (35%)**, we build simple schedulers **better** than CFS

In terms of stability, CFS is **less stable** than most generated schedulers on **5 applications**.
$\Rightarrow$ 1 scheduler (CFS) is not a baseline to determine if an application is stable or not.

Preliminary Results

Raw results confirm the **value of building application-specific schedulers**:

Out of 20 applications:

- For **17 applications (85%)**, we build simple schedulers **on par** with CFS
- For **7 applications (35%)**, we build simple schedulers **better** than CFS

In terms of stability, CFS is **less stable** than most generated schedulers on **5 applications**.

$\Rightarrow$ 1 scheduler (CFS) is not a baseline to determine if an application is stable or not.

Preliminary Results

Raw results confirm the **value of building application-specific schedulers**:

Out of 20 applications:

- For **17 applications (85%)**, we build simple schedulers **on par** with CFS
- For **7 applications (35%)**, we build simple schedulers **better** than CFS

In terms of stability, CFS is **less stable** than most generated schedulers on **5 applications**.
$\Rightarrow$ 1 scheduler (CFS) is not a baseline to determine if an application is stable or not.

Finding the Best Application-Specific Scheduler

Brute-force approach: Run all generated schedulers and keep the best one.

+ Gives **the best** scheduler for the application
– Impractical (for 10 runs and 486 schedulers, 1,925 hours for all tested applications)
– Does not scale realistically with the number of features and applications

We need a more practical way of finding the best scheduler for an application!

The new approach should:

- Find a good scheduler without testing all schedulers
- Be able to use one application's results for other applications

Finding the Best Application-Specific Scheduler

Brute-force approach: Run all generated schedulers and keep the best one.

+ Gives **the best** scheduler for the application
– Impractical (for 10 runs and 486 schedulers, 1,925 hours for all tested applications)
– Does not scale realistically with the number of features and applications

We need a more practical way of finding the best scheduler for an application!

The new approach should:

- Find a good scheduler without testing all schedulers
- Be able to use one application's results for other applications

Finding the Best Application-Specific Scheduler

Brute-force approach: Run all generated schedulers and keep the best one.

+ Gives **the best** scheduler for the application
– Impractical (for 10 runs and 486 schedulers, 1,925 hours for all tested applications)
– Does not scale realistically with the number of features and applications

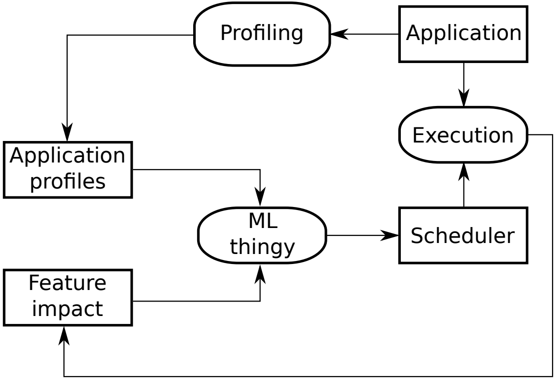
We need a more practical way of finding the best scheduler for an application!

The new approach should:

- Find a good scheduler without testing all schedulers
- Be able to use one application's results for other applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

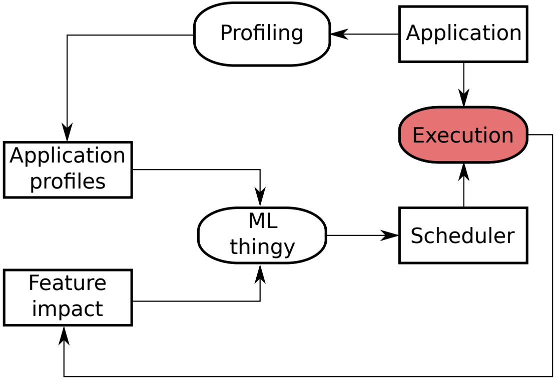
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

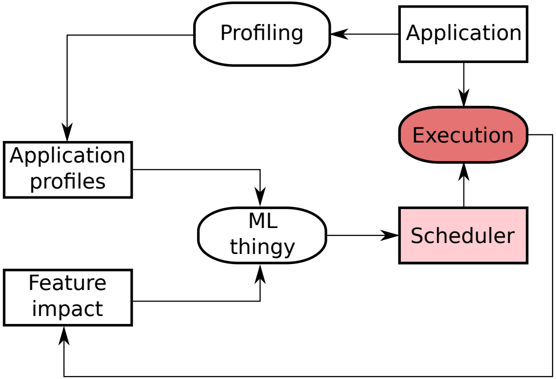
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

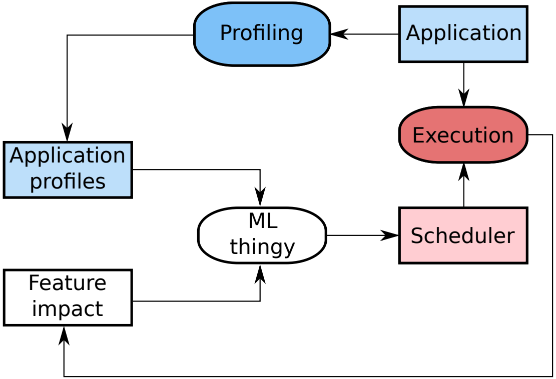
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

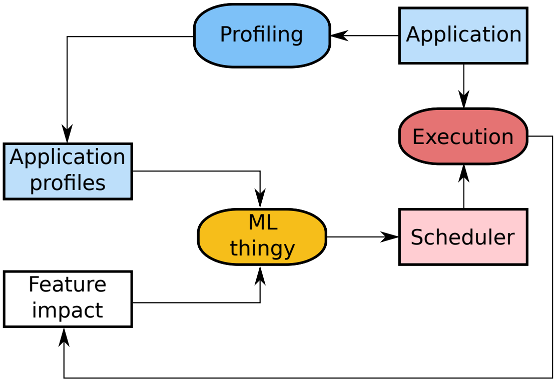
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

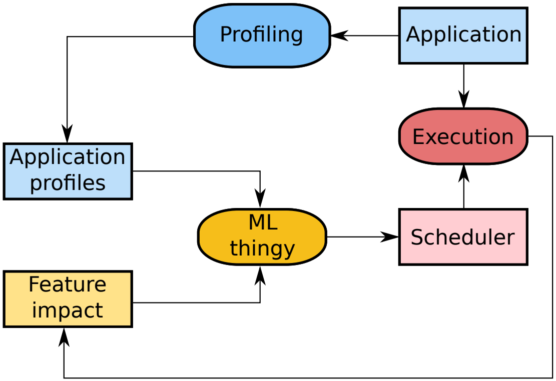
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



We already have:

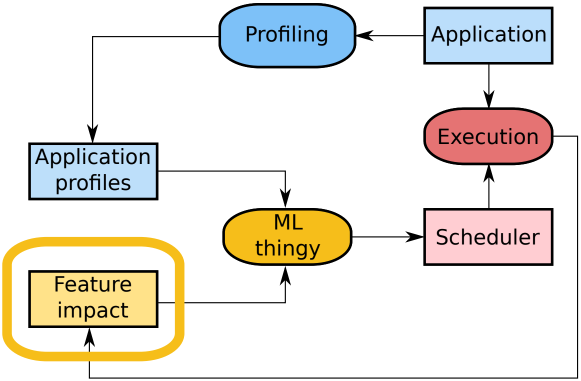
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Performance-Driven Feature Search

We propose the following framework:



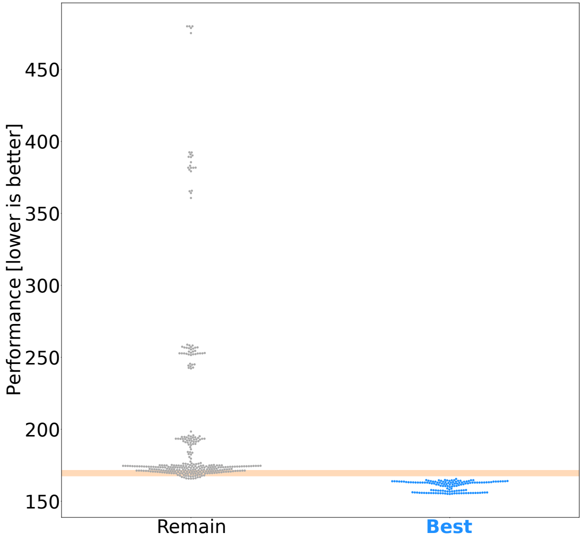
We already have:

- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`

What we still need:

- **ML algorithm:** match scheduler to application
- **Inputs for ML:** features' impact on applications

Feature Evaluation: the `facesim` Example



Isolating Features

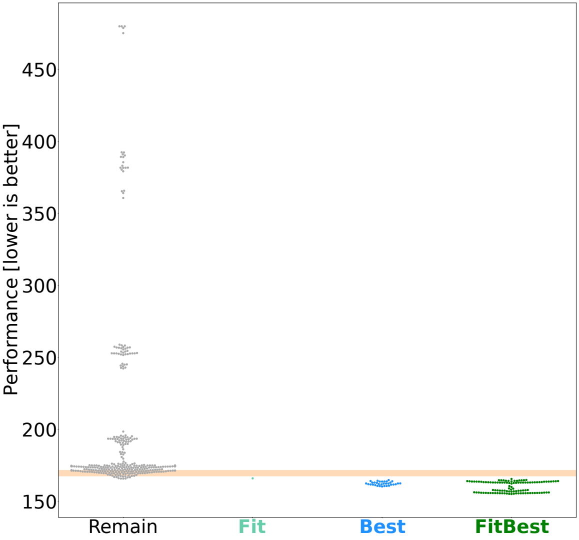
Count the occurrences of each feature in **Best**:
$> 80\% \Rightarrow$ **good**, $< 20\% \Rightarrow$ **bad**

FitBest: schedulers with all **good** features and no **bad** ones in **Best**.

Fit: schedulers with all **good** features and no **bad** ones **not** in **Best**.

```
{ idle = yes
  Load metric ≠ nrRunBlock
  Placement distance ≠ SMT }
```

Feature Evaluation: the facesim Example



Isolating Features

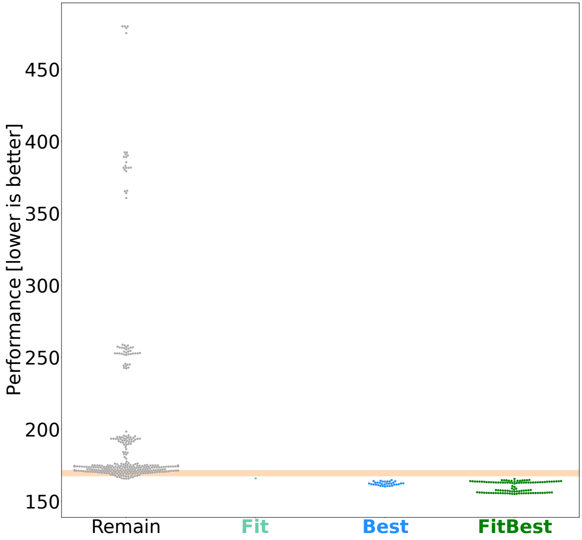
Count the occurrences of each feature in **Best**:
$> 80\% \Rightarrow$ **good**, $< 20\% \Rightarrow$ **bad**

FitBest: schedulers with all **good** features and no **bad** ones in **Best**.

Fit: schedulers with all **good** features and no **bad** ones **not** in **Best**.

```
{ idle = yes
  Load metric ≠ nrRunBlock
  Placement distance ≠ SMT }
```

Feature Evaluation: the `facesim` Example



Conclusion

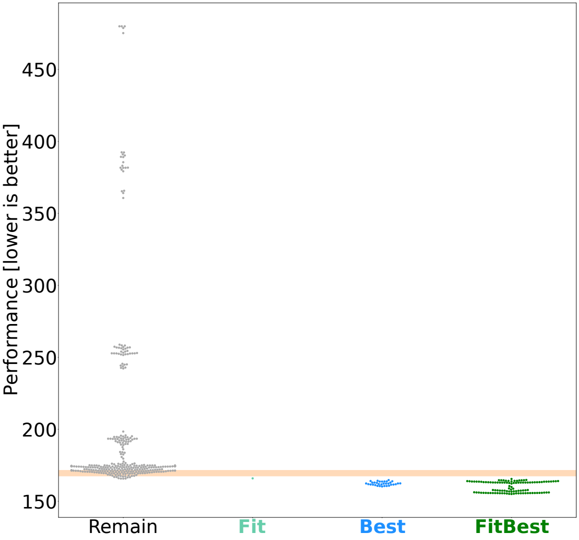
We **isolate** the **best features** for an application.

Representativeness: $\mathcal{R}=\frac{|FitBest|}{|Best|}=76\%$

Precision, *i.e. false positives*: $\mathcal{P}=\frac{|FitBest|}{|Fit \cup FitBest|}=99\%$

We reduce the noise in experimental data and pave the way for ML-based approaches.

Feature Evaluation: the `facesim` Example



Conclusion

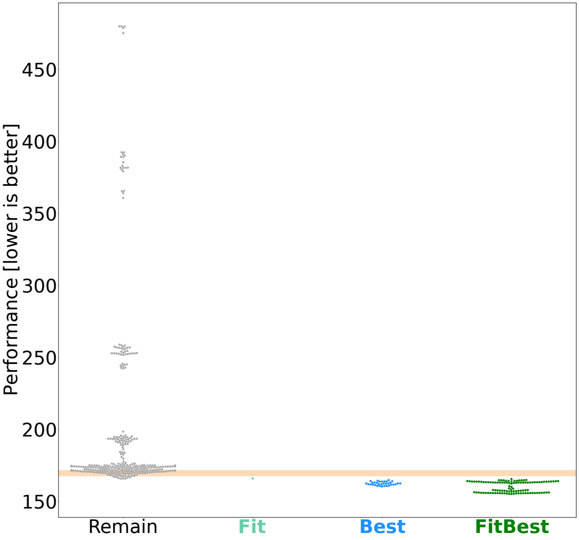
We **isolate** the **best features** for an application.

Representativeness: $\mathcal{R} = \frac{|FitBest|}{|Best|} = 76\%$

Precision, *i.e. false positives*: $\mathcal{P} = \frac{|FitBest|}{|Fit \cup FitBest|} = 99\%$

We reduce the noise in experimental data and pave the way for ML-based approaches.

Feature Evaluation: the `facesim` Example



Conclusion

We **isolate** the **best features** for an application.

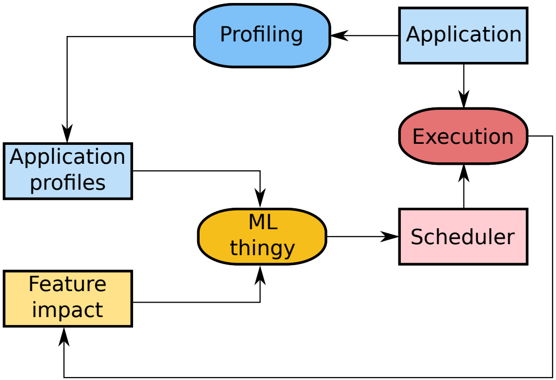
Representativeness: $\mathcal{R} = \frac{|FitBest|}{|Best|} = 76\%$

Precision, *i.e. false positives*: $\mathcal{P} = \frac{|FitBest|}{|Fit \cup FitBest|} = 99\%$

We reduce the noise in experimental data and pave the way for ML-based approaches.

Performance-Driven Feature Search

We propose the following framework:



We already have:

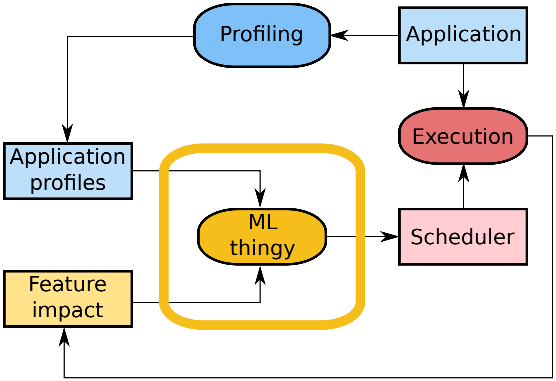
- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`
- **Inputs for ML:** features' impact on applications

We now can:

- **ML algorithm:** match scheduler to application

Performance-Driven Feature Search

We propose the following framework:



We already have:

- **Execution framework:** SaaKM
- **Scheduler generator:** library of features
- **Profiling:** stats from `procfs` + `ftrace`
- **Inputs for ML:** features' impact on applications

We now can:

- **ML algorithm:** match scheduler to application

Application-Specific Schedulers

1. A **feature model** representing schedulers as independent features
   - Implemented as a library such that features can be evaluated independently
   - 16 features, 486 generated schedulers in its current state, complies with SaaKM
2. An **ML-based approach** to build application-specific schedulers
   - Application profiling
   - Feature impact analysis
3. A **methodology** to understand the impact of each feature
   - Evaluate the stability of applications and schedulers
   - Build the set of desirable features for a given application

Perspectives

- Expand the model with new features
- Implement our ML engine to automatically build application-specific schedulers

Application-Specific Schedulers

1. A **feature model** representing schedulers as independent features
   - Implemented as a library such that features can be evaluated independently
   - 16 features, 486 generated schedulers in its current state, complies with SaaKM
2. An **ML-based approach** to build application-specific schedulers
   - Application profiling
   - Feature impact analysis
3. A **methodology** to understand the impact of each feature
   - Evaluate the stability of applications and schedulers
   - Build the set of desirable features for a given application

Perspectives

- Expand the model with new features
- Implement our ML engine to automatically build application-specific schedulers

Axis 3: Contributions

Application-Specific Schedulers

1. A **feature model** representing schedulers as independent features
   - Implemented as a library such that features can be evaluated independently
   - 16 features, 486 generated schedulers in its current state, complies with SaaKM
2. An **ML-based approach** to build application-specific schedulers
   - Application profiling
   - Feature impact analysis
3. A **methodology** to understand the impact of each feature
   - Evaluate the stability of applications and schedulers
   - Build the set of desirable features for a given application

Perspectives

- Expand the model with new features
- Implement our ML engine to automatically build application-specific schedulers

Axis 3: Contributions

Application-Specific Schedulers

1. A **feature model** representing schedulers as independent features
   - Implemented as a library such that features can be evaluated independently
   - 16 features, 486 generated schedulers in its current state, complies with SaaKM
2. An **ML-based approach** to build application-specific schedulers
   - Application profiling
   - Feature impact analysis
3. A **methodology** to understand the impact of each feature
   - Evaluate the stability of applications and schedulers
   - Build the set of desirable features for a given application

Perspectives

- Expand the model with new features
- Implement our ML engine to automatically build application-specific schedulers

Conclusion

Contribution Summary

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1
Scheduler Development

Ipanema DSL
SaaKM API
Property verification

Axis 2
Performance Enhancement

High-resolution monitoring tools
Frequency inversion problem
S_{move} solution submitted

Axis 3
Application-Specific Schedulers

Feature model
Feature evaluation methodology
ML-based scheduler building approach

Contribution Summary

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1
Scheduler Development

Ipanema DSL
SaaKM API
Property verification

Axis 2
Performance Enhancement

High-resolution monitoring tools
Frequency inversion problem
S_{move} solution submitted

Axis 3
Application-Specific Schedulers

Feature model
Feature evaluation methodology
ML-based scheduler building approach

Contribution Summary

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1
Scheduler Development

Ipanema DSL
SaaKM API
Property verification

Axis 2
Performance Enhancement

High-resolution monitoring tools
Frequency inversion problem
S_{move} solution submitted

Axis 3
Application-Specific Schedulers

Feature model
Feature evaluation methodology
ML-based scheduler building approach

Contribution Summary

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1
Scheduler Development

Ipanema DSL
SaaKM API
Property verification

Axis 2
Performance Enhancement

High-resolution monitoring tools
Frequency inversion problem
S_{move} solution submitted

Axis 3
Application-Specific Schedulers

Feature model
Feature evaluation methodology
ML-based scheduler building approach

Long Term Perspectives

Axes 1 and 2:

- Extend the Ipanema standard library with more hardware features (e.g. frequency)

Axes 1 and 3:

- Extend the Ipanema standard library and the feature model to better account for other resources such as memory, disks, network, etc ...

Axes 2 and 3:

- Expand feature model with more hardware-specific features (frequency, heterogeneity, ...)

Publications

- **Towards Proving Optimistic Multicore Schedulers**. *Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Julia Lawall and Gilles Muller*. **HotOS, 2017**
- **Ipanema : un Langage Dédié pour le Développement d'Ordonnanceurs Multi-Coeur Sûrs**. *Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi and Nicolas Palix*. **ComPAS, 2017**
- **The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS**. *Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller and Julien Sopena*. **ATC, 2018**
- **Understanding Scheduler Performance: a Feature-Based Approach**. *Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi and Nicolas Palix*. **ComPAS, 2019**
- **Fork/Wait and Multicore Frequency Scaling: a Generational Clash**. *Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall and Gilles Muller*. **PLOS, 2019**
- **Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance**. *Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall and Gilles Muller*. **ATC, 2020**
- **Provable Multicore Schedulers with Ipanema: Application to Work Conservation**. *Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall and Gilles Muller*. **EuroSys, 2020**

Contribution Summary

How can we help **developers** implement **efficient** schedulers in a **safe** and **easy** way?

How can we help **users** get the best **performance** for their applications?

Axis 1
Scheduler Development

Ipanema DSL
SaaKM API
Property verification

Axis 2
Performance Enhancement

High-resolution monitoring tools
Frequency inversion problem
S_{move} solution submitted

Axis 3
Application-Specific Schedulers

Feature model
Feature evaluation methodology
ML-based scheduler building approach

References

1. **A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation**. *Gilles Muller, Julia L. Lawall, Hervé Duchesne*. **HASE, 2005**
2. **Scheduling for Reduced CPU Energy**. *Mark Weiser, Brent B. Welch, Alan J. Demers, Scott Shenker*. **OSDI, 1994**
3. **Resource-conscious Scheduling for Energy Efficiency on Multicore Processors**. *Andreas Merkel, Jan Stoess, Frank Bellosa*. **EuroSys, 2010**
4. **An Evaluation of Per-chip Nonuniform Frequency Scaling on Multicores**. *Xiao Zhang, Sandhya Dwarkadas, Rongrong Zhong*. **ATC, 2010**
5. **Power and Energy Management for Server Systems**. *Ricardo Bianchini, Ram Rajamony*. **Computer, 2004**

Backup Slides

Feature Evaluation: the `facesim` Example

Election	RBtree 44 (31.43%)	Linked list 48 (34.29%)	FIFO 48 (34.29%)
Time slice	Infinite 47 (33.57%)	Fixed 46 (32.86%)	Split 47 (33.57%)
Load metric	nrRun 54 (38.57%)	nrRunBlock 33 (23.57%)	usedTime 53 (37.86%)
Placement distance	SMT 33 (23.57%)	LLC 54 (38.57%)	all 53 (37.86%)
Executor	all 36 (25.71%)	node 51 (36.43%)	core 53 (37.86%)
Idle	no 16 (11.43%)	yes 124 (88.57%)	

Phase 3: Isolating the best features

Count the occurrences of each feature in **Best**.

If feature is $> 80\% \Rightarrow$ **good**
If feature is $< 20\% \Rightarrow$ **bad**

We call this set of features a **scheduler frame**.