ATC'20

Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

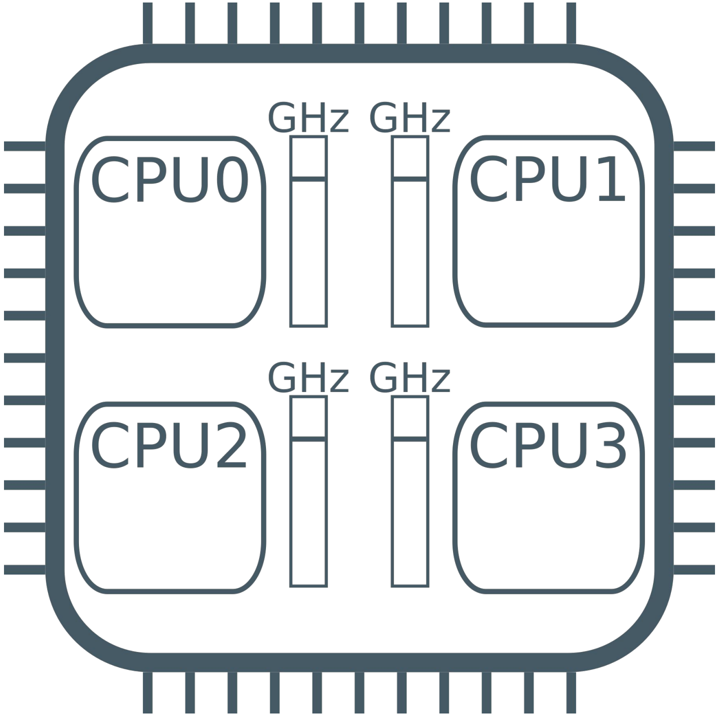
R. Gouicem, D. Carver, J. Sopena, J. Lawall, G. Muller
Sorbonne University, LIP6, Inria

B. Lepers, W. Zwaenepoel
University of Sydney

J.-P. Lozi
Oracle

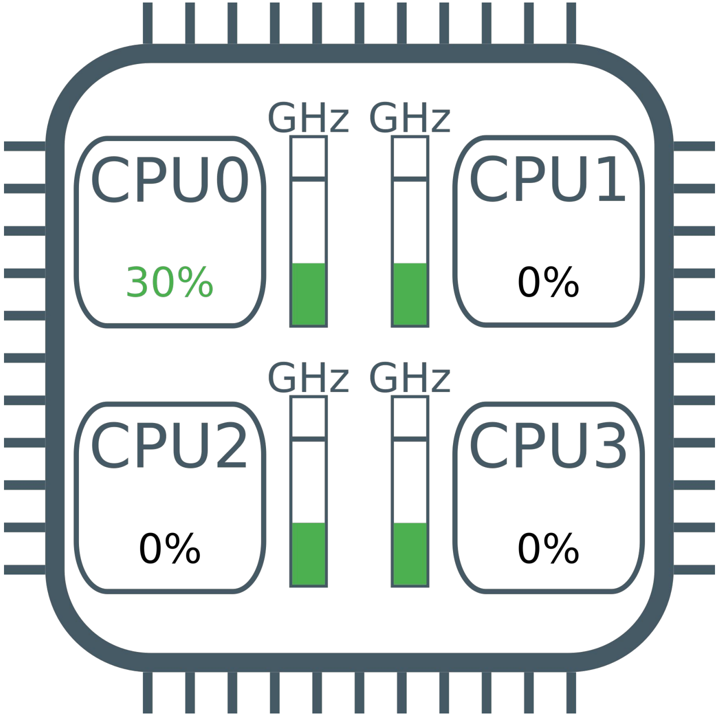
N. Palix
Université Grenoble Alpes

Dynamic Frequency Scaling Before



CPU frequency changes depending on load

Dynamic Frequency Scaling Before

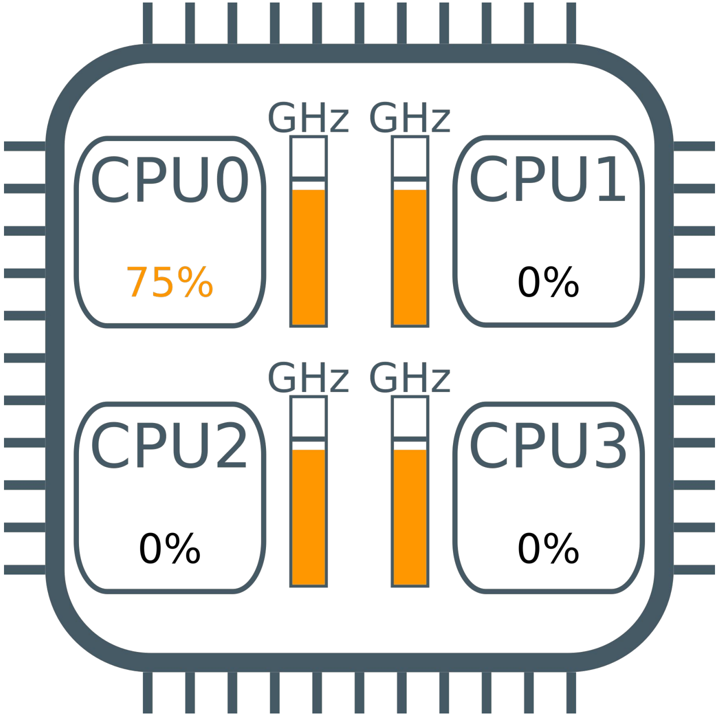


CPU frequency changes depending on load

Frequency is managed at **chip granularity**

The load of a single CPU impacts the frequency of all CPUs on the chip

Dynamic Frequency Scaling Before

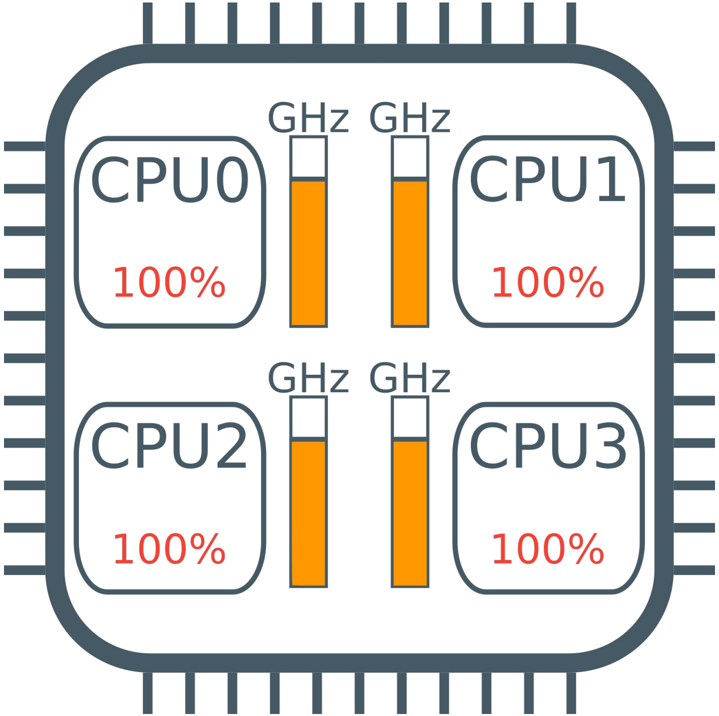


CPU frequency changes depending on load

Frequency is managed at **chip granularity**

The load of a single CPU impacts the frequency of all CPUs on the chip

Dynamic Frequency Scaling Before



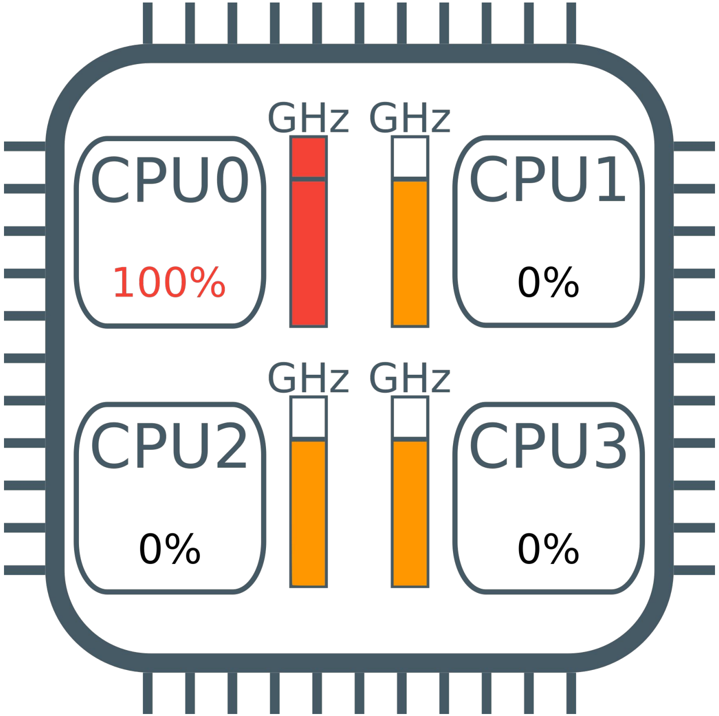
CPU frequency changes depending on load

Frequency is managed at **chip granularity**

The load of a single CPU impacts the frequency of all CPUs on the chip

With all CPUs fully loaded, **nominal frequency** is guaranteed

Dynamic Frequency Scaling Before



CPU frequency changes depending on load

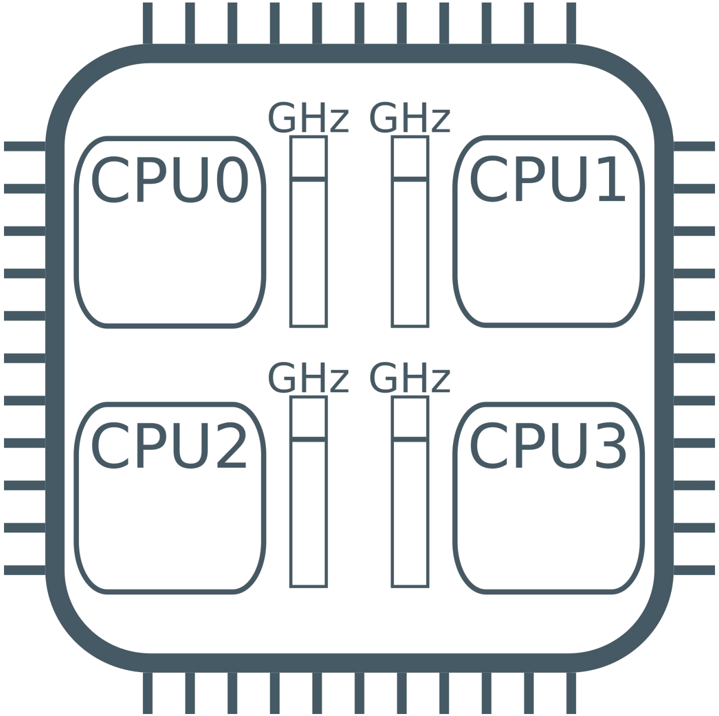
Frequency is managed at **chip granularity**

The load of a single CPU impacts the frequency of all CPUs on the chip

With all CPUs fully loaded, **nominal frequency** is guaranteed

Turbo mode: when some CPUs are idle, busy CPUs can use even higher frequencies

Dynamic Frequency Scaling Now

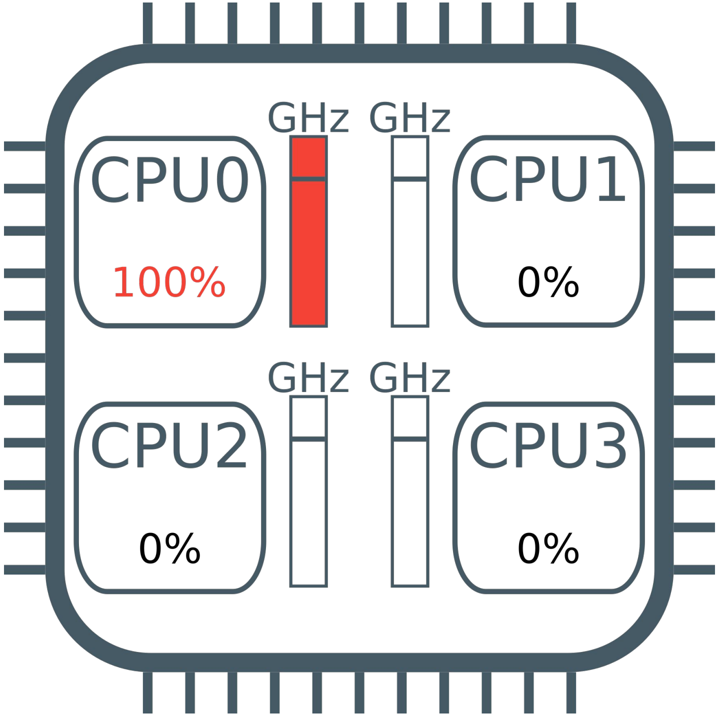


Frequency is managed at **core granularity**

At least since:

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

Dynamic Frequency Scaling Now



Frequency is managed at **core granularity**

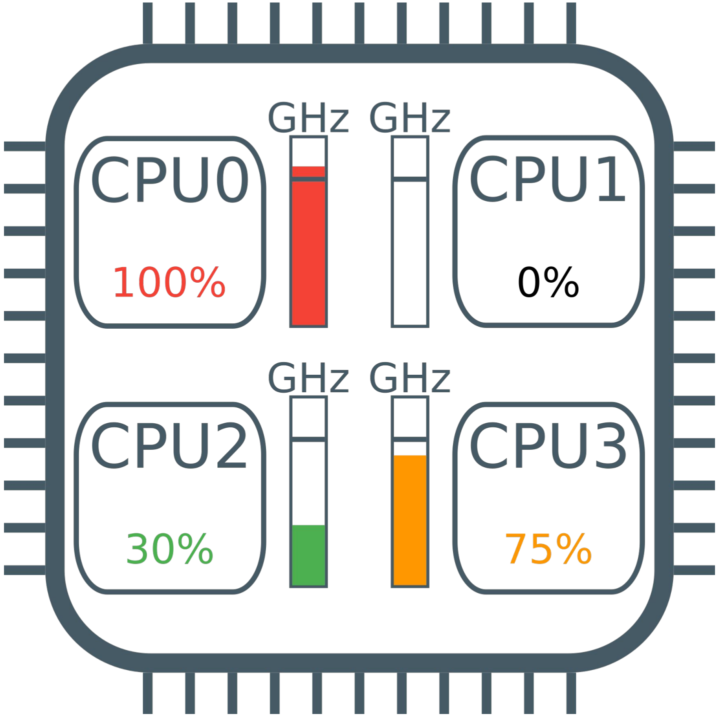
At least since:

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

Idle cores can run at minimal frequency while other cores run at maximal frequency

→ Energy savings

Dynamic Frequency Scaling Now



Frequency is managed at **core granularity**

At least since:

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

Idle cores can run at minimal frequency while other cores run at maximal frequency

→ Energy savings

Each core individually sets a frequency that matches **its** load

Previous Work

Focus on changing the frequency to match load

- Linux scaling governors (ondemand, schedutil)
- hardware frequency scaling (Intel)

Frequency scaling was used to

- maximize instructions per joule metric (Weiser’94)
- reduce contention (Merkel’10, Zhang’10)
- reduce energy usage (Bianchini’03)

Recent work by the Linux scheduler community

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

Case Study: Compiling Linux

Setup:

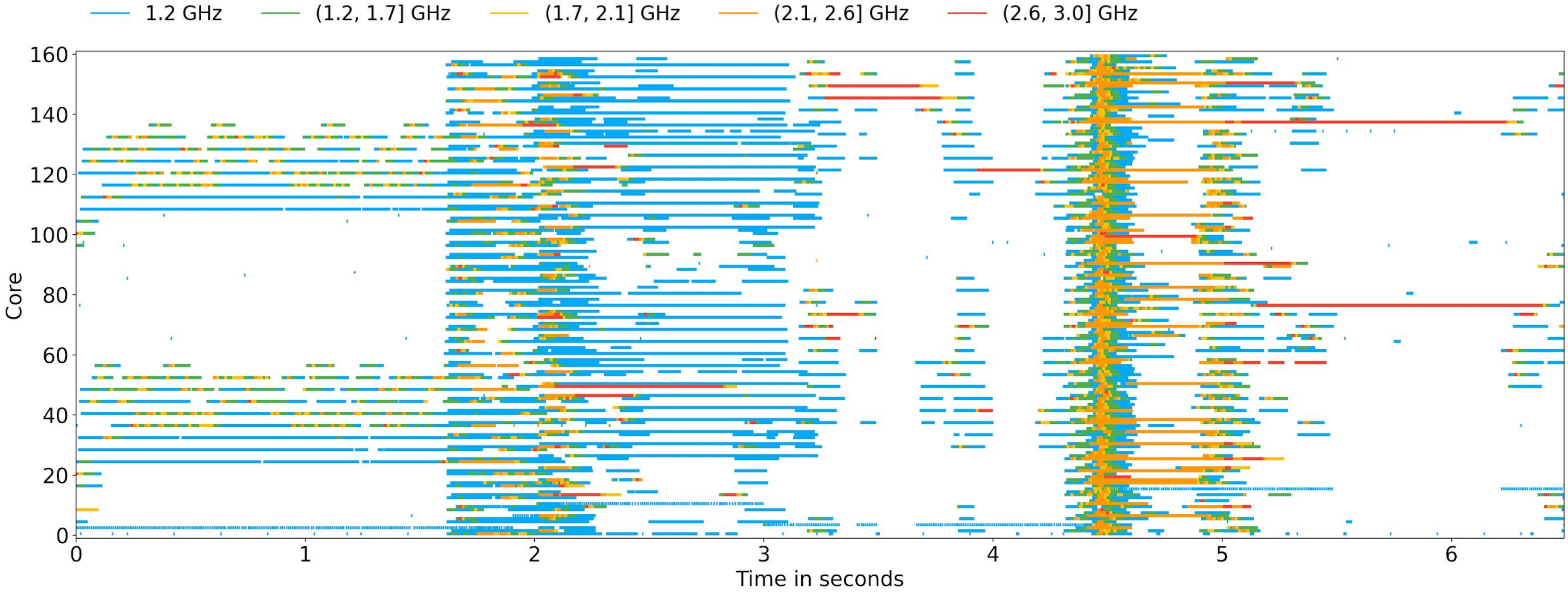
- 4x20-core Intel® Xeon E7-8870 v4 (160 HW threads with HyperThreading)
- 2.1 GHz nominal frequency, up to 3.0 GHz with Turbo Boost®
- Per-core frequency scaling
- 512 GB of RAM
- Debian 10 Buster with Linux 5.4

Maximum Turbo frequencies:

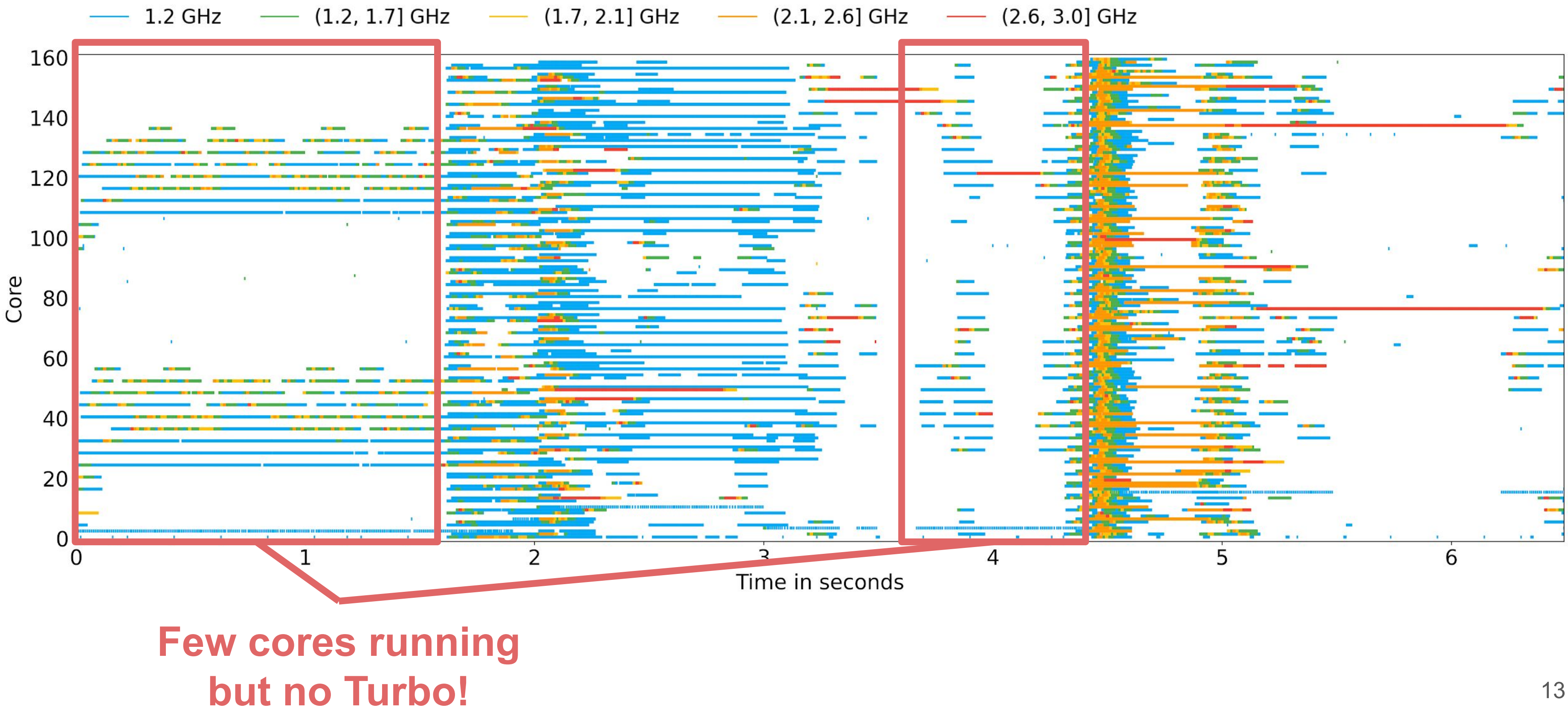
Active cores	1-2	3	4	5-8	>8
Max Turbo	3.0 GHz	2.8 GHz	2.7 GHz	2.6 GHz	2.1 GHz

For clarity, we only present the compilation of the scheduler subsystem

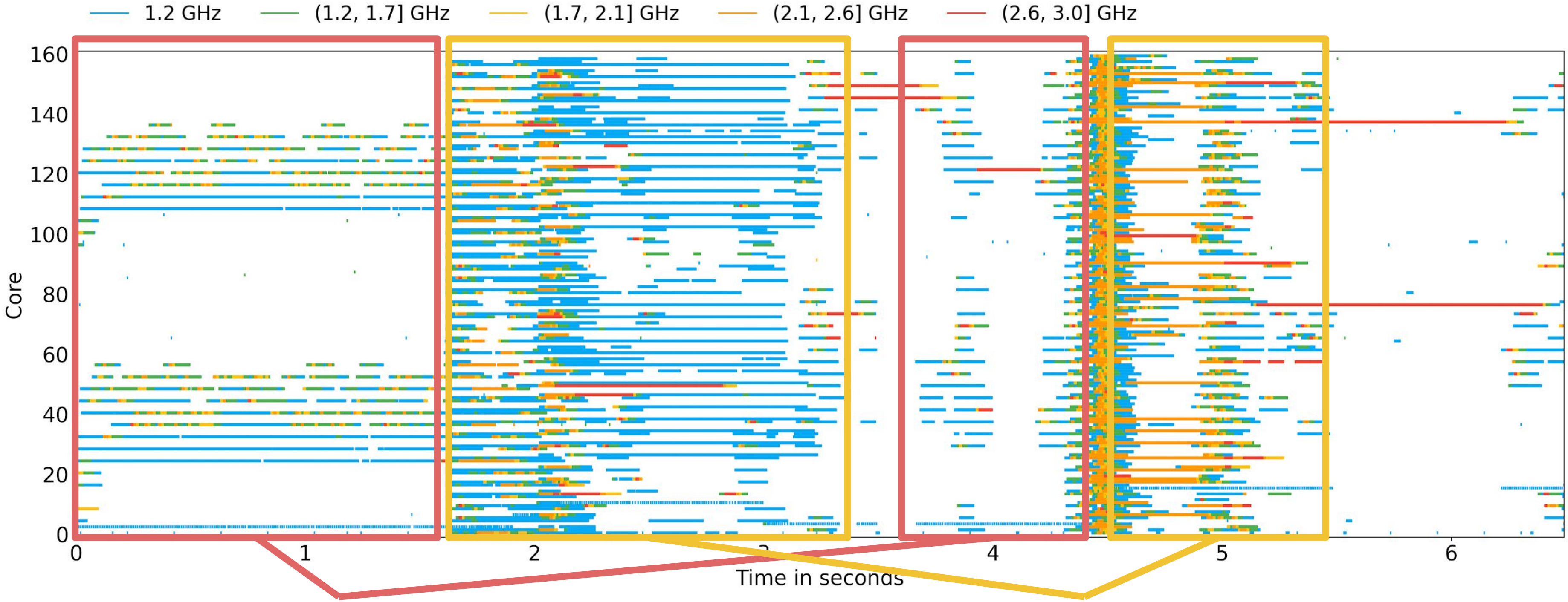
Case Study: Tracing the Frequency



Case Study: Tracing the Frequency



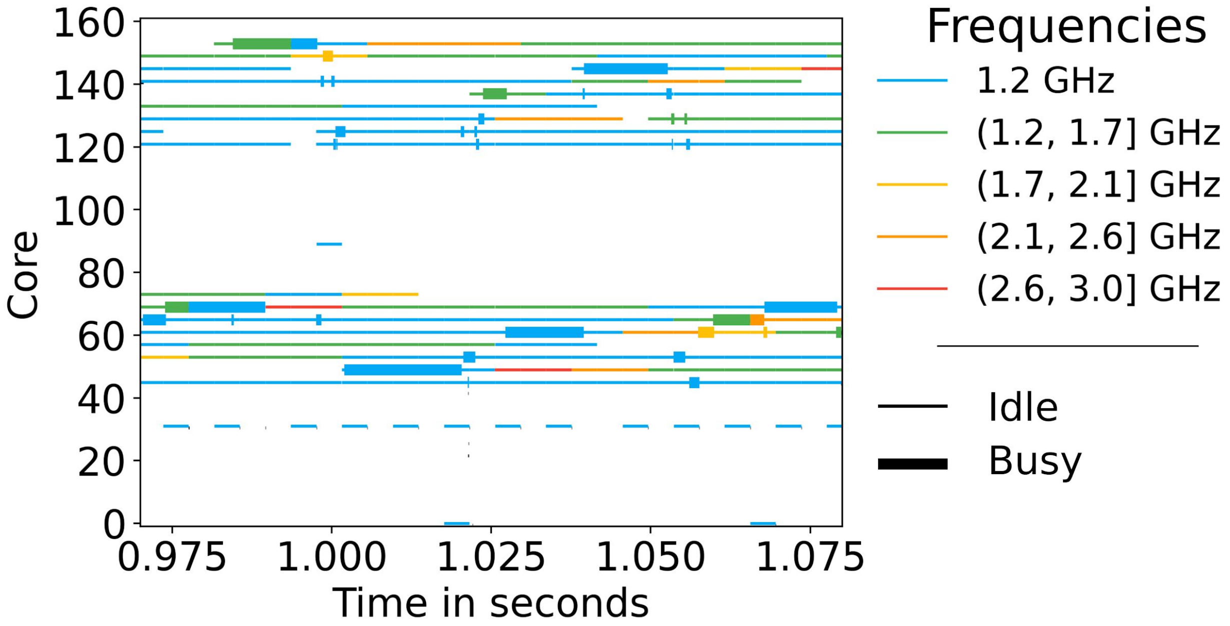
Case Study: Tracing the Frequency



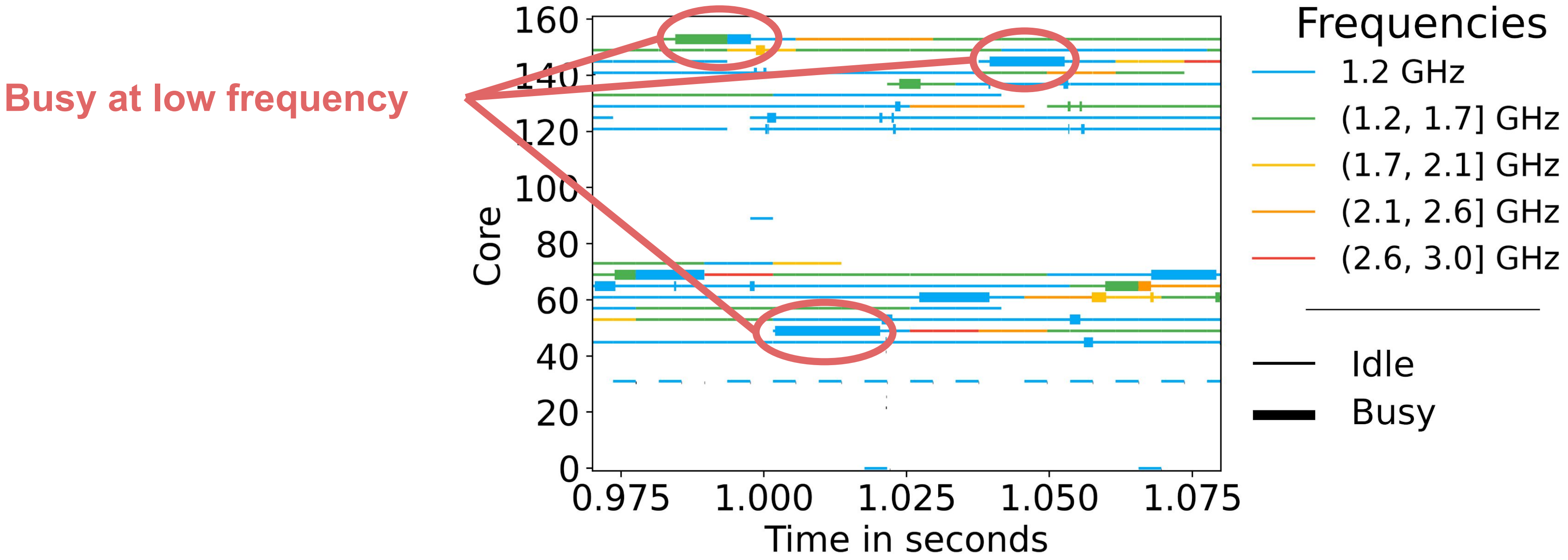
Few cores running but no Turbo!

Most cores used, but frequency is lower than nominal!

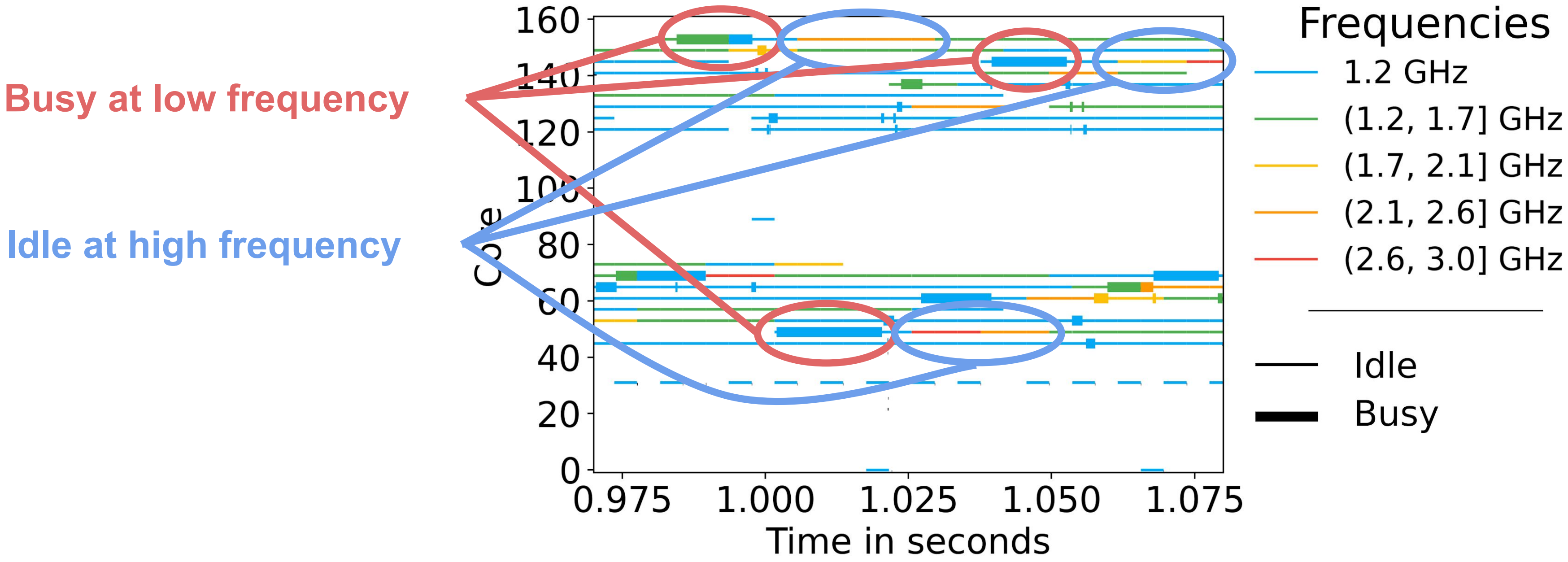
Case Study: Zooming In



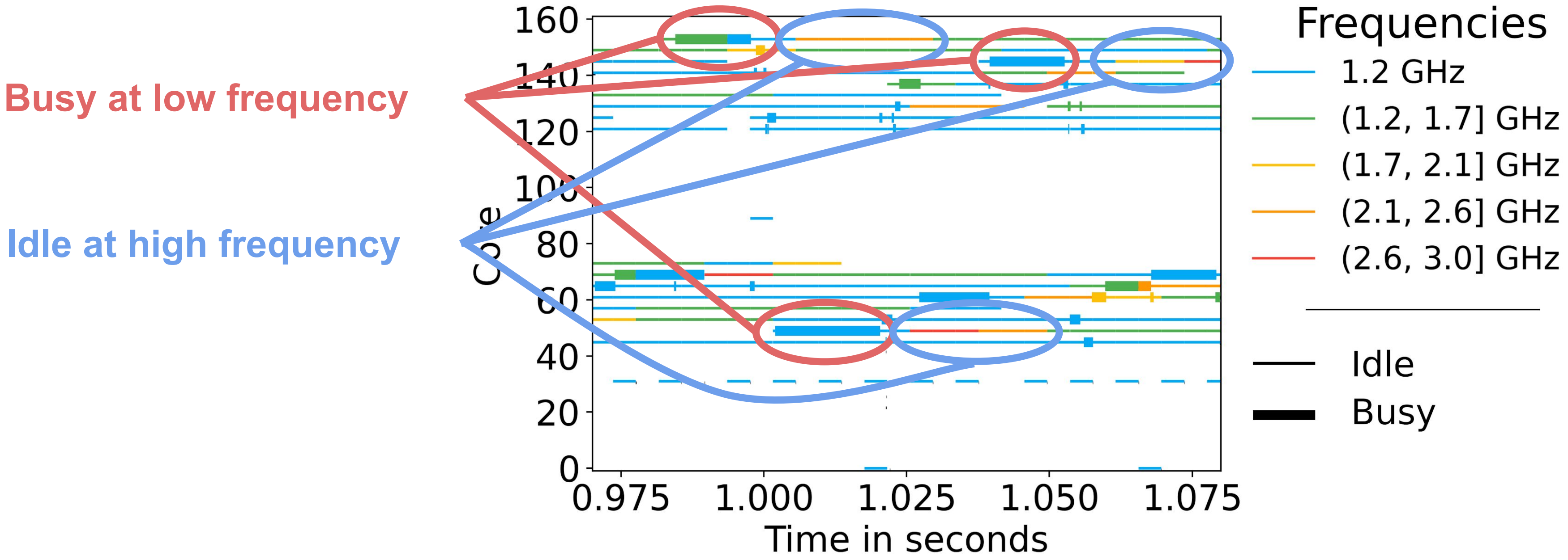
Case Study: Zooming In



Case Study: Zooming In



Case Study: Zooming In



Frequency and load are mismatched!

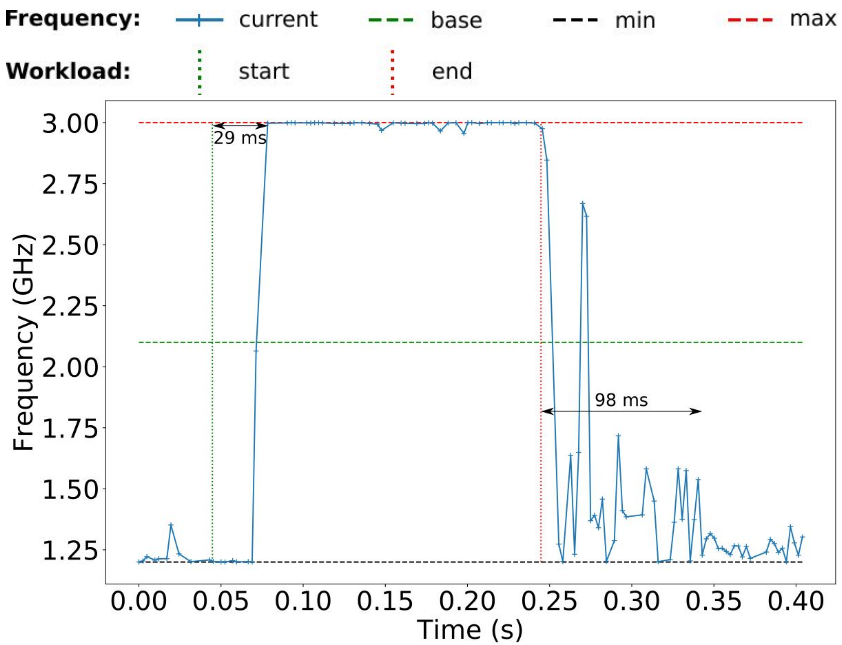
Frequency Transition Latency

FTL: Latency between a change of load and change of frequency
We measure it from idleness to 100% load on our server

Frequency Transition Latency

FTL: Latency between a change of load and change of frequency
We measure it from idleness to 100% load on our server

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

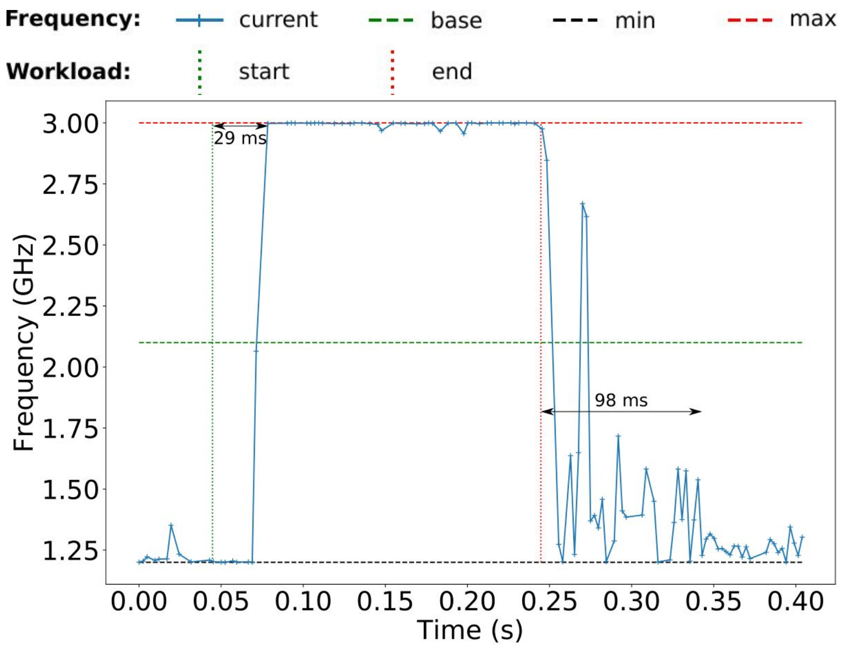


Frequency Transition Latency

FTL: Latency between a change of load and change of frequency
We measure it from idleness to 100% load on our server

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

Changing frequency is not instantaneous!

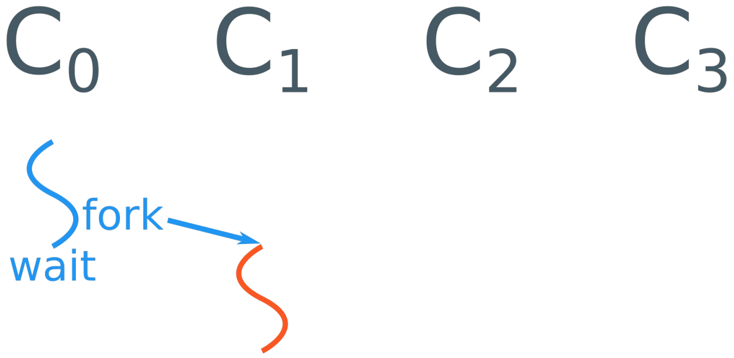


Tracing Scheduler Events

Behavior of Linux scheduler (**CFS**):

New and waking threads are placed on **idle** cores if available

→ **work conserving**



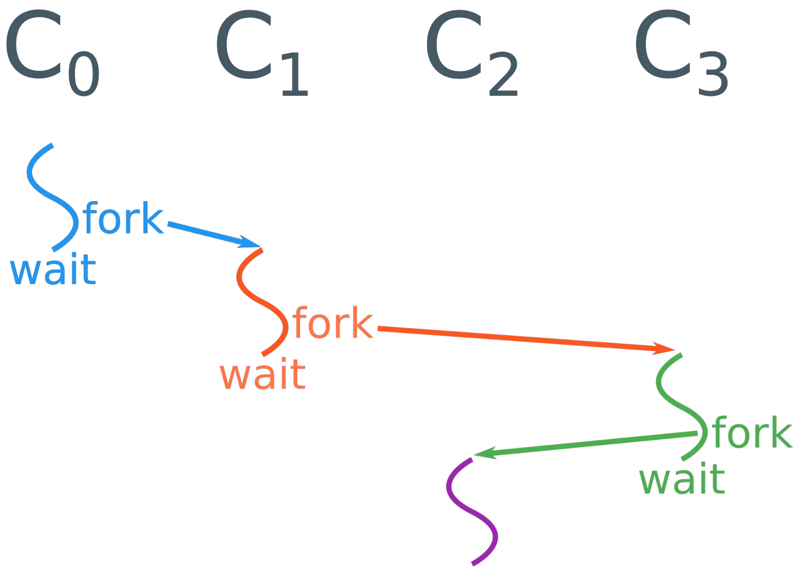
Tracing Scheduler Events

Behavior of Linux scheduler (**CFS**):

New and waking threads are placed on **idle** cores if available

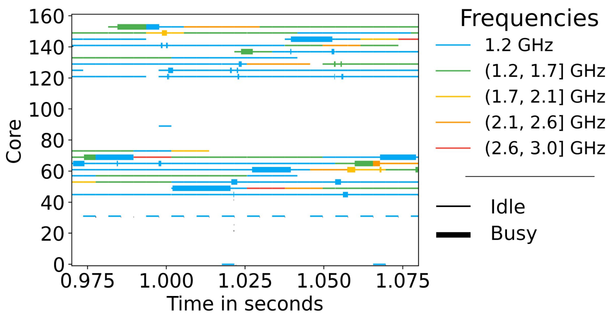
→ **work conserving**

This repeated **fork/wait** pattern is a common occurrence in our case study.



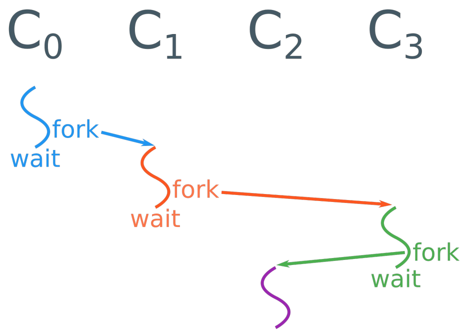
Problem: Frequency Inversion

Long FTLs



+

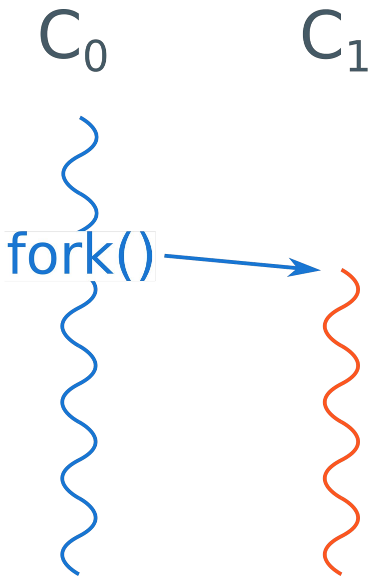
Work conserving scheduler



=

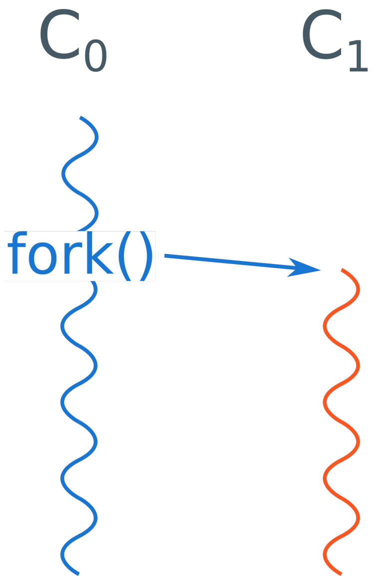
The frequencies at which two cores operate are inverted as compared to their load

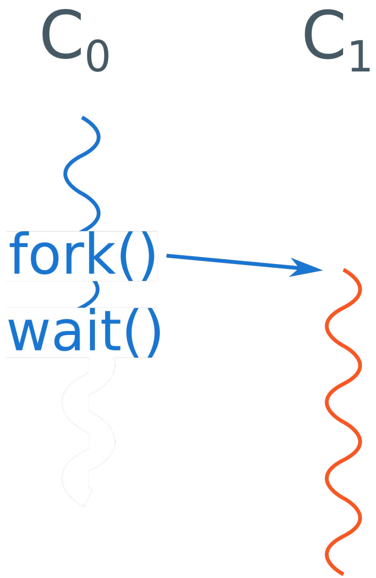
Problem: With CFS



Ideal situation,
both cores are busy

Problem: With CFS





Ideal situation,
both cores are busy

Two cores used for a sequential work,
prone to **frequency inversion**

Solution 1: Local Placement

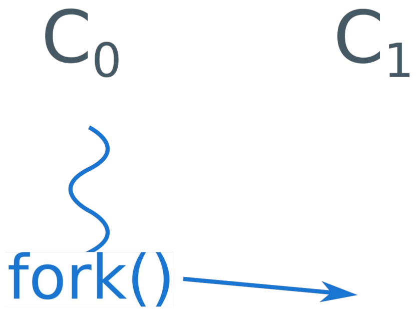
We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

C_0 C_1

Solution 1: Local Placement

We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

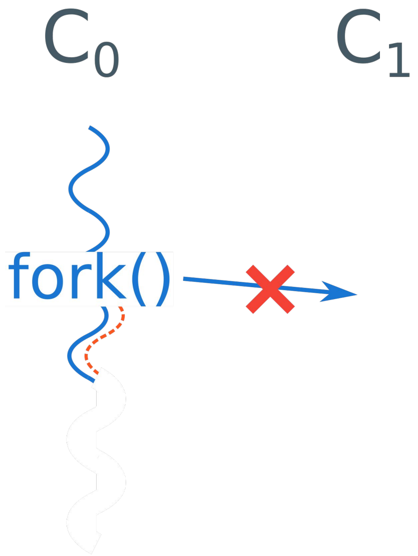


Solution 1: Local Placement

We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.



Solution 1: Local Placement

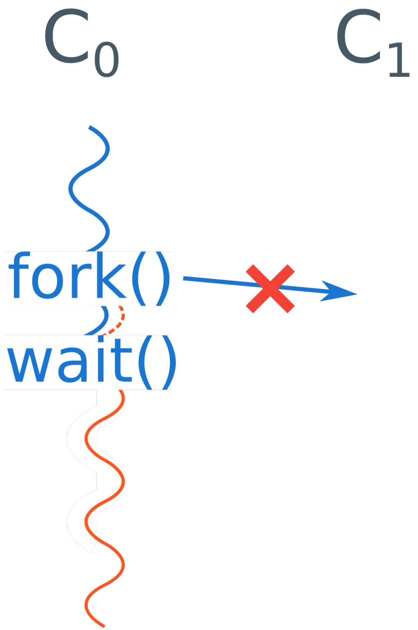
We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.

Parent thread calls the wait() syscall, the **child thread** is scheduled on C_0.

We use a **single** core for a sequential work.

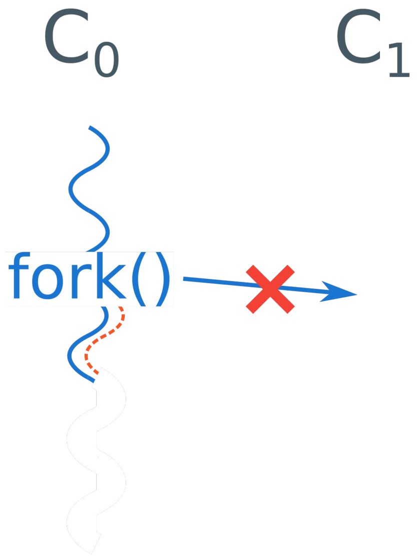


Solution 1: Local Placement

We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.



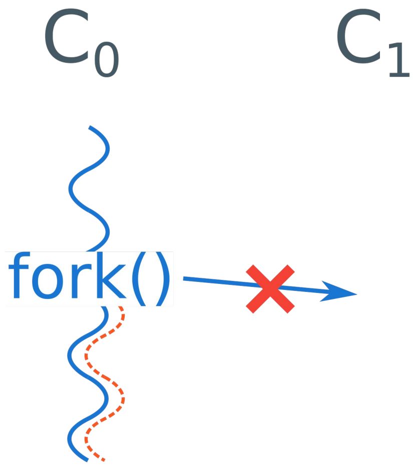
Solution 1: Local Placement

We propose local placement with $\mathbf{S_{local}}$.

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

Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.

If the **parent thread** keeps running, C_0 will stay overloaded,



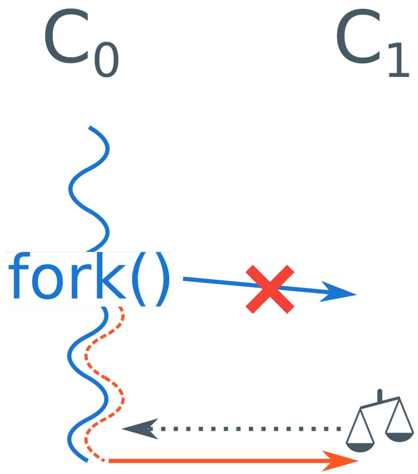
Solution 1: Local Placement

We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.

If the **parent thread** keeps running, C_0 will stay overloaded, until **load balancing** migrates the **child thread** on C_1.



Solution 1: Local Placement

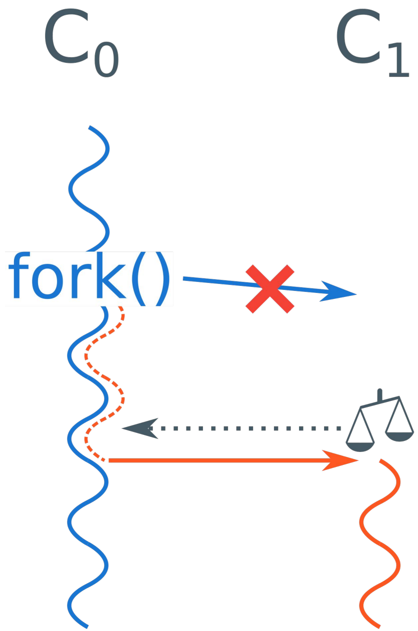
We propose local placement with $\mathbf{S}_{\mathbf{local}}$.

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

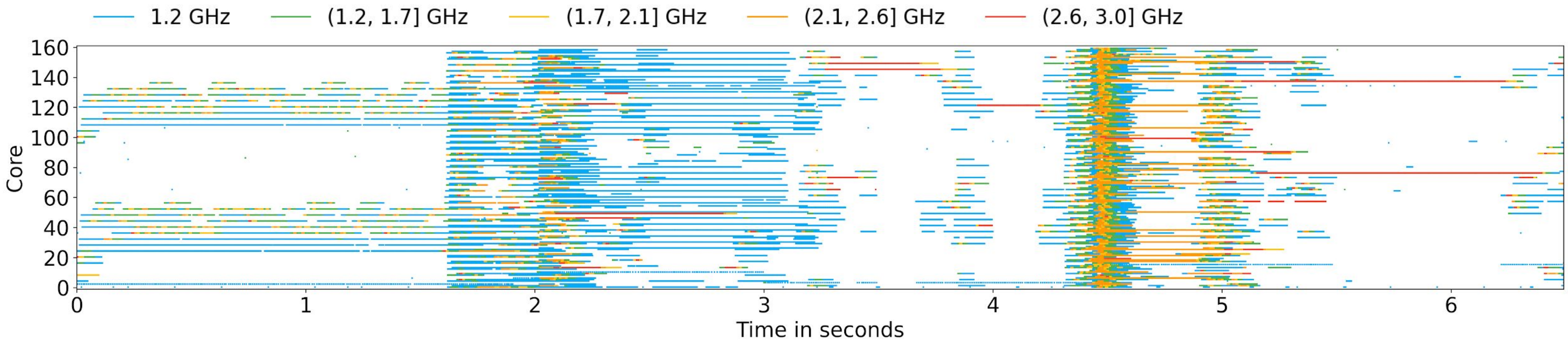
Instead, S_{local} **cancels** this migration and always places the **child thread** on C_0.

If the **parent thread** keeps running, C_0 will stay overloaded, until **load balancing** migrates the **child thread** on C_1.

Both cores are used, but we lost tens of milliseconds of execution for the **child thread**.

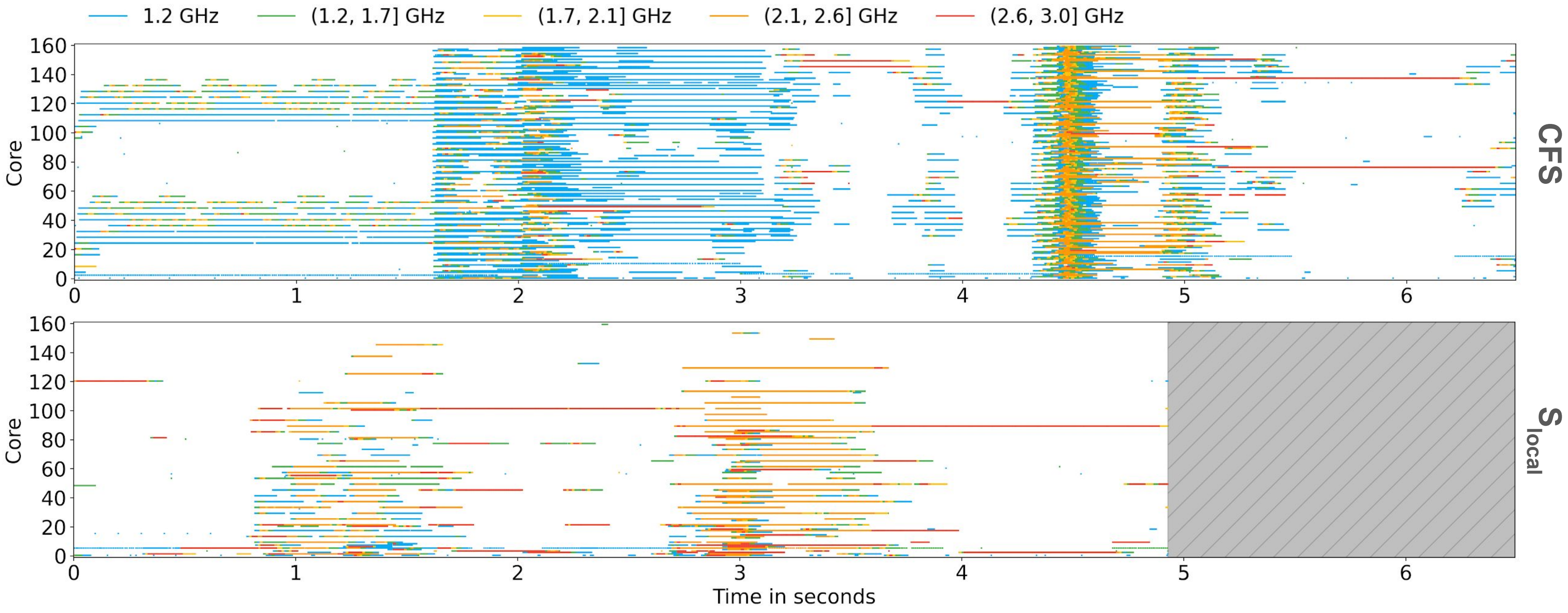


Solution 1: Local Placement

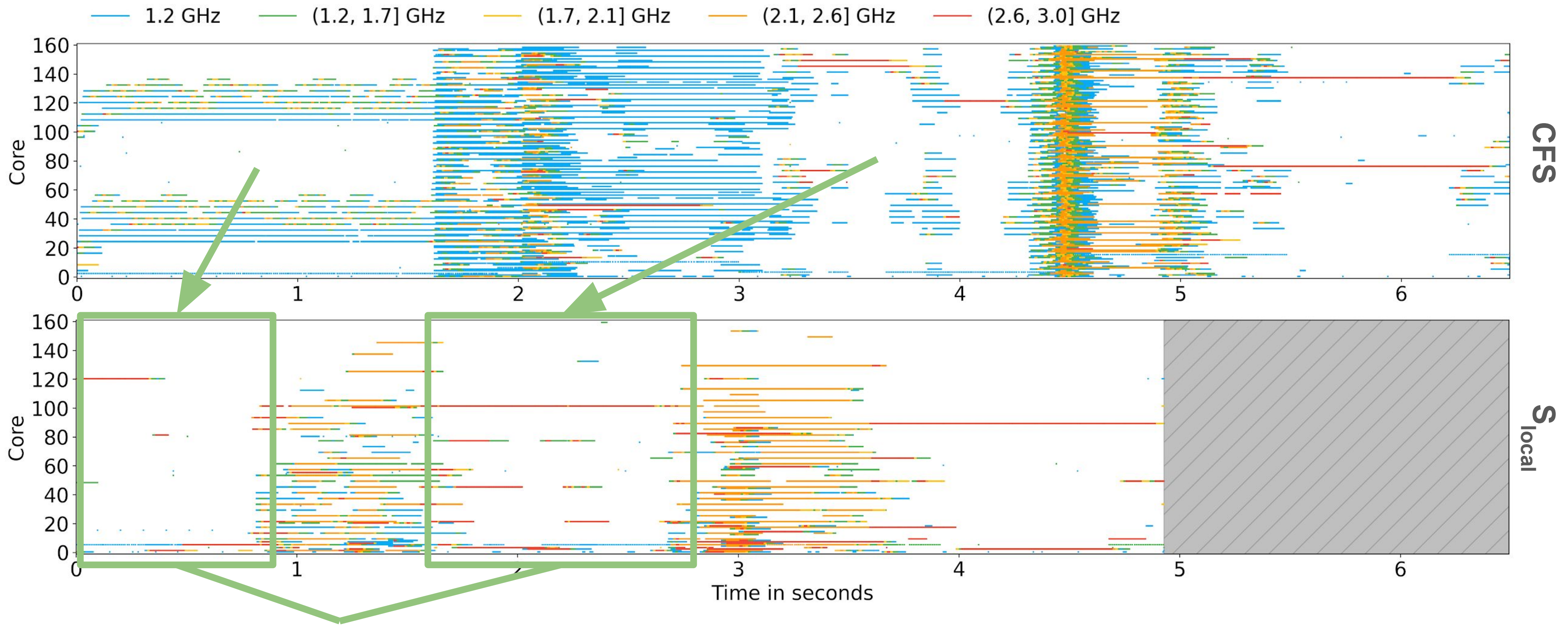


CFS

Solution 1: Local Placement

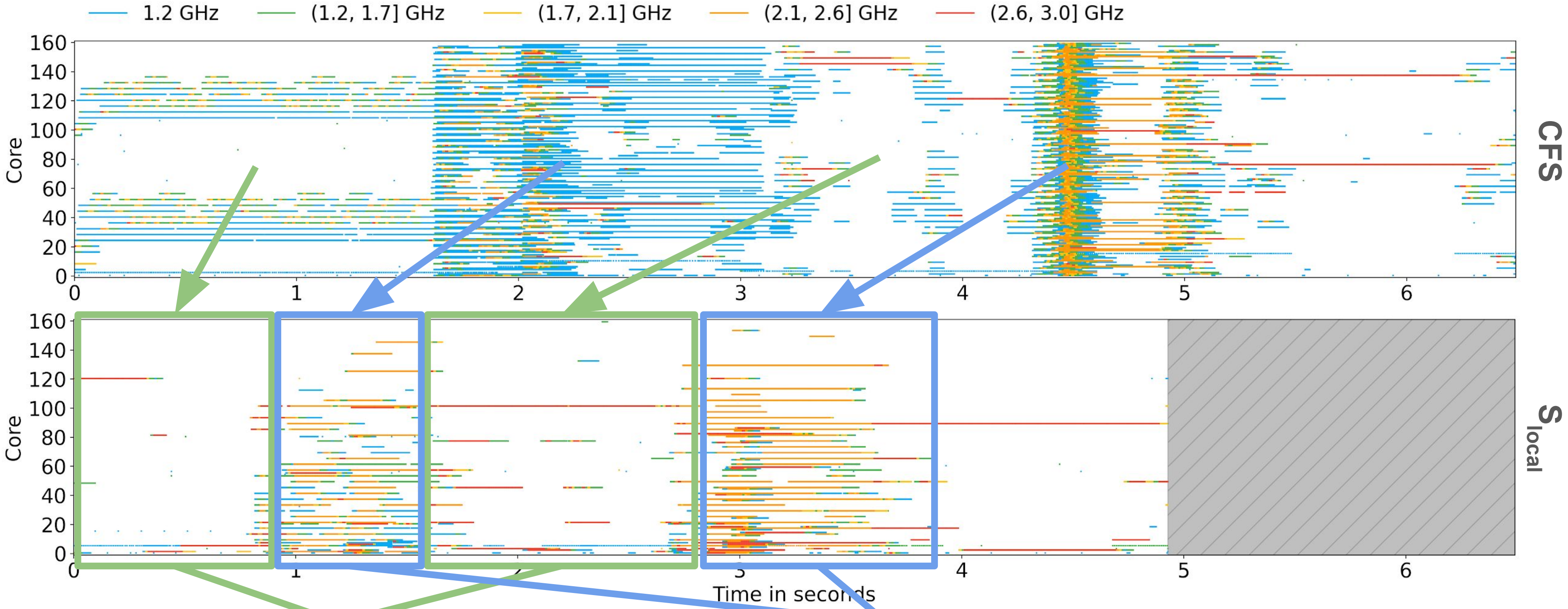


Solution 1: Local Placement



<5 cores used, more Turbo

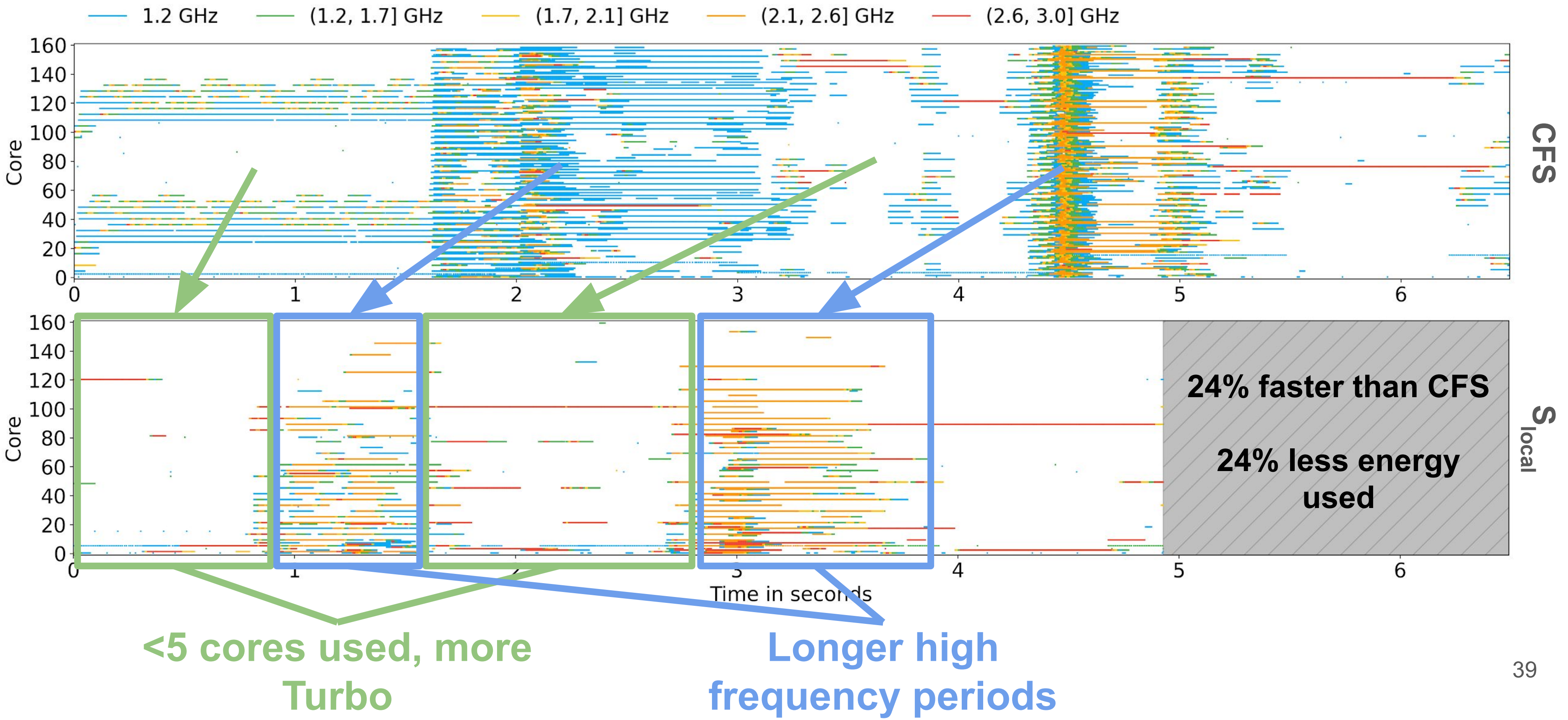
Solution 1: Local Placement



<5 cores used, more Turbo

Longer high frequency periods

Solution 1: Local Placement



Solution 2: Deferring Thread Migrations

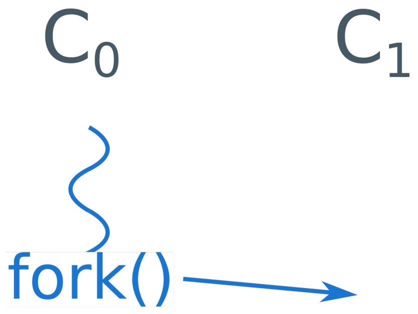
We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

C_0 C_1

Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

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

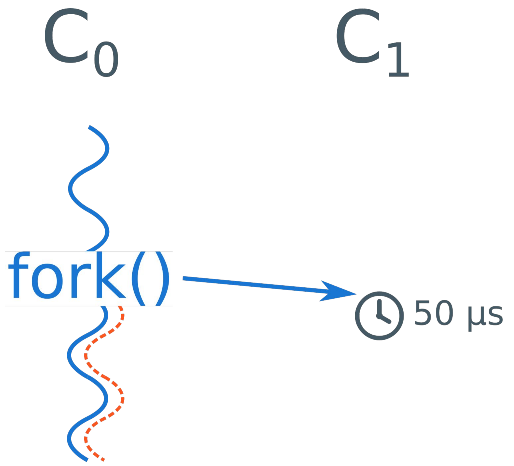


Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

Parent thread runs on C_0, calls the fork() syscall.
CFS decides to place the **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 50µs and place the **child thread** on C_0.



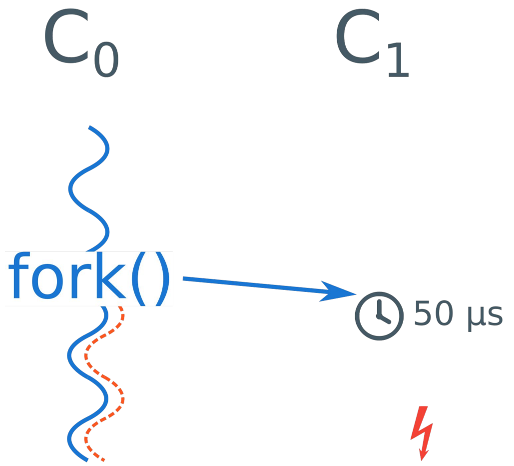
Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

Parent thread runs on C_0, calls the fork() syscall.
CFS decides to place the **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 50µs and place the **child thread** on C_0.

When the timer is **triggered** 50µs later,



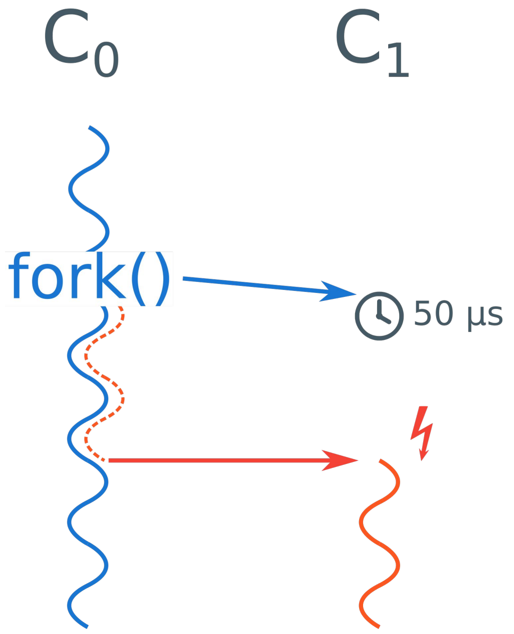
Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

Parent thread runs on C_0, calls the fork() syscall.
CFS decides to place the **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 50µs and place the **child thread** on C_0.

When the timer is **triggered** 50µs later, we migrate the **child thread** to C_1.
We only lose 50µs compared to CFS or S_{local}.

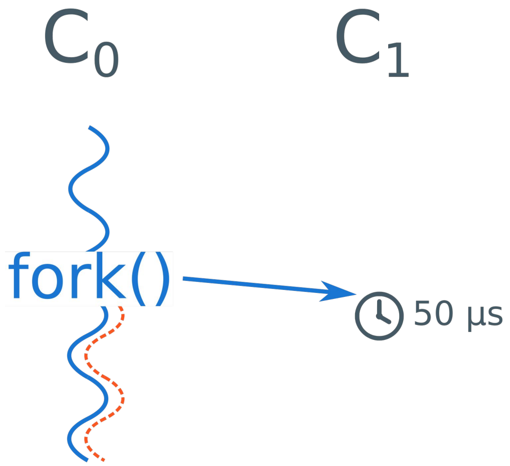


Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

Parent thread runs on C_0, calls the fork() syscall.
CFS decides to place the **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 50µs and place the **child thread** on C_0.



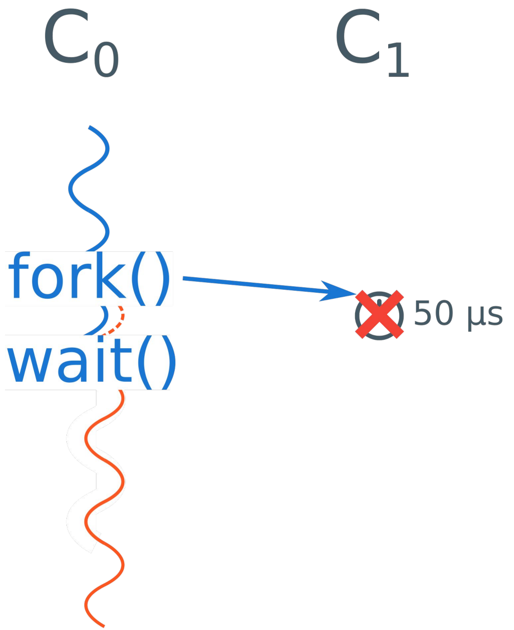
Solution 2: Deferring Thread Migrations

We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

Parent thread runs on C_0, calls the fork() syscall.
CFS decides to place the **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 50µs 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 **cancelled**.



Solution 2: Deferring Thread Migrations

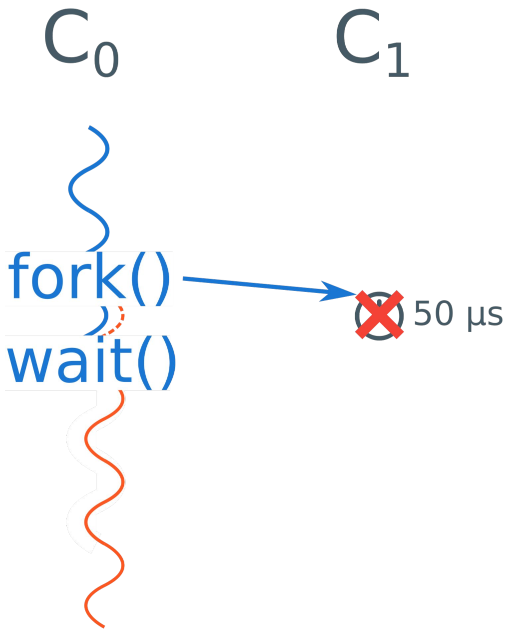
We propose to delay migrations with $\mathbf{S}_{\mathbf{move}}$.

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

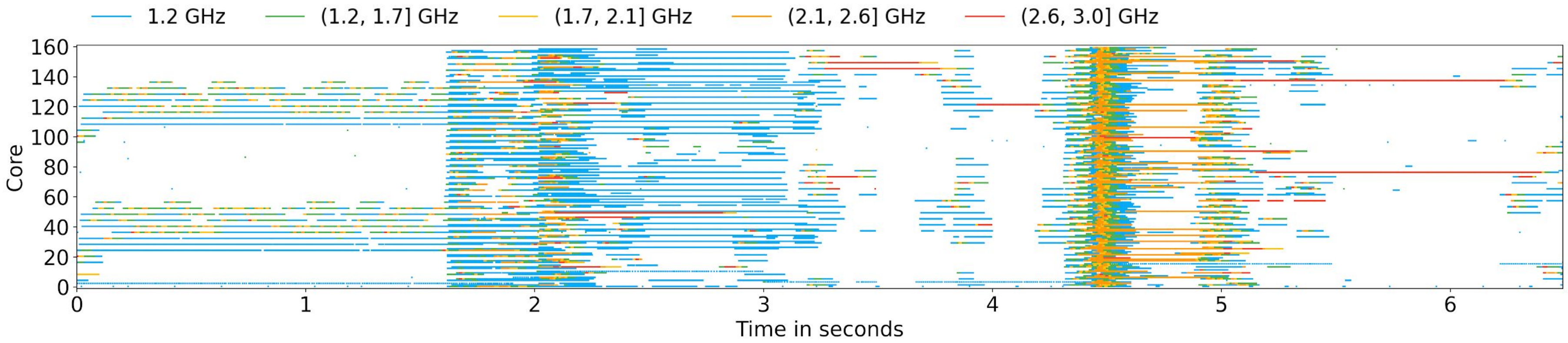
If C_1 runs at a **low frequency**, instead of placing the **child thread** on C_1, we arm a timer that expires in 50µs and place the **child thread** on C_0.

Parent thread calls the wait() syscall, the **child thread** is scheduled on C_0, the timer is **cancelled**.

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

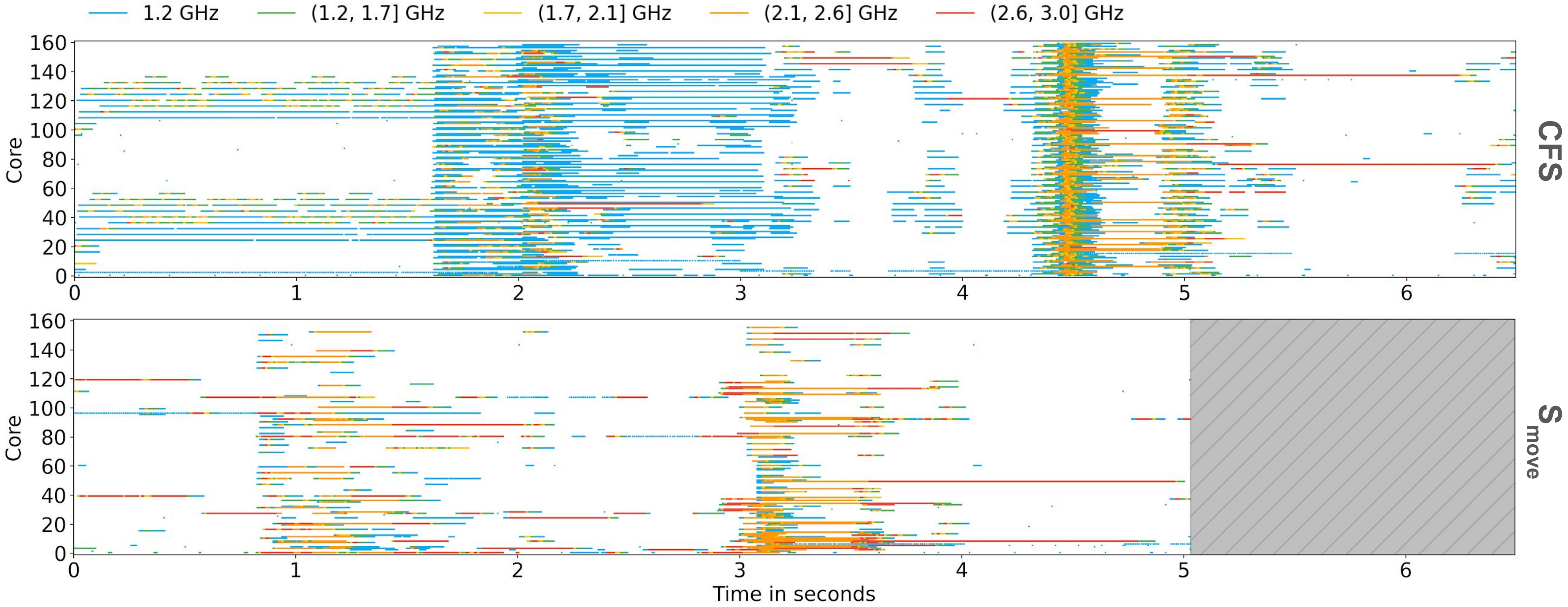


Solution 2: Deferring Thread Migrations

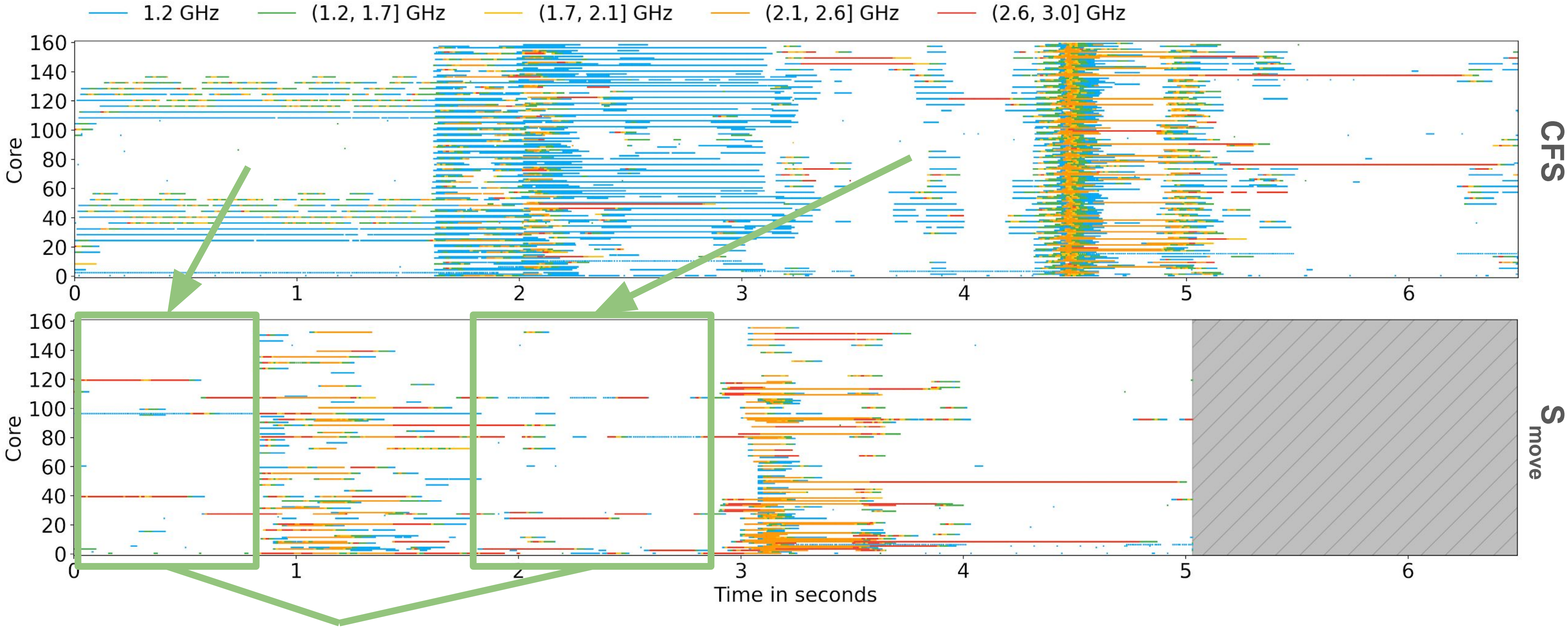


CFS

Solution 2: Deferring Thread Migrations

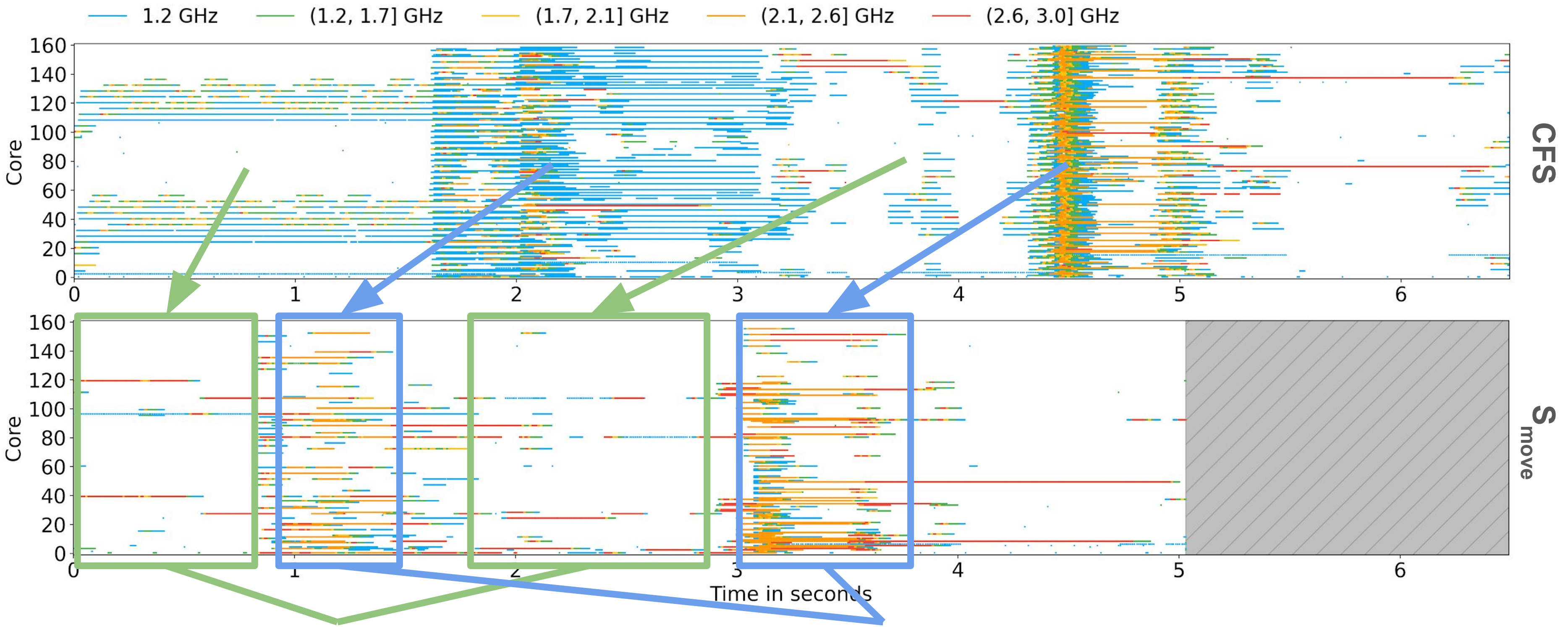


Solution 2: Deferring Thread Migrations



<5 cores used, more Turbo

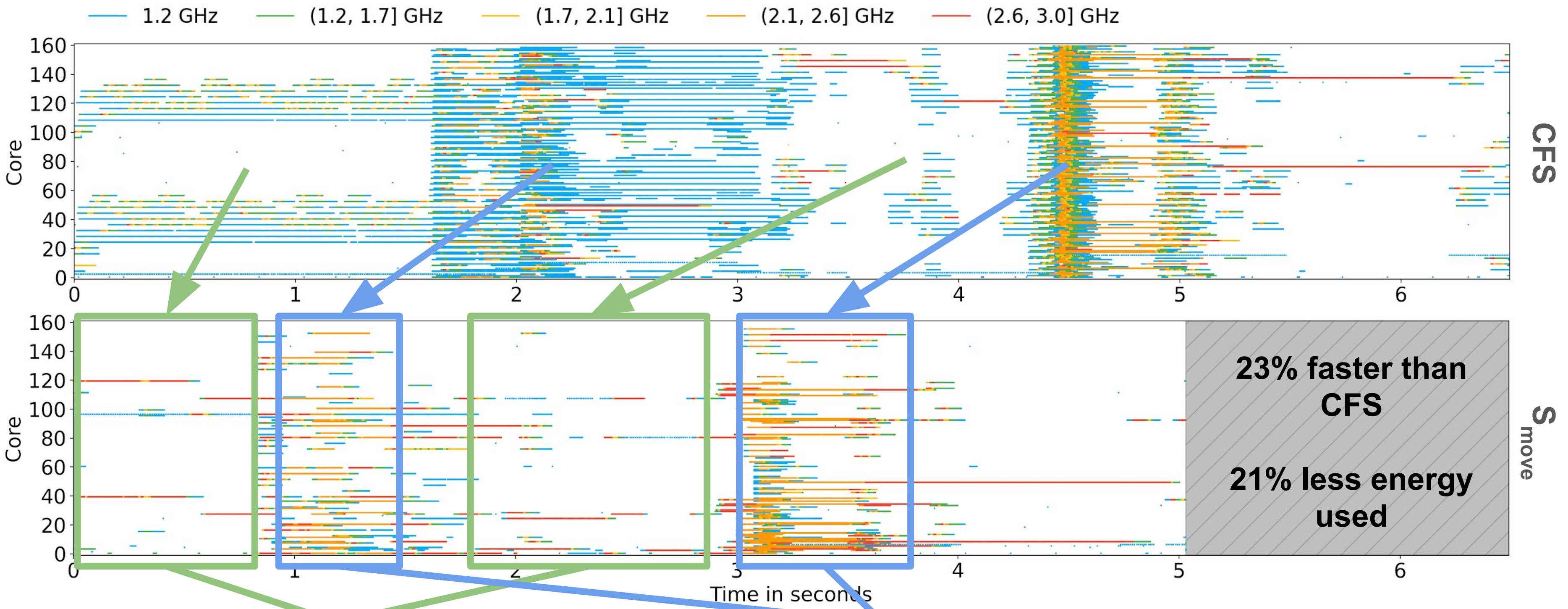
Solution 2: Deferring Thread Migrations



<5 cores used, more Turbo

Longer high frequency periods

Solution 2: Deferring Thread Migrations



<5 cores used, more Turbo

Longer high frequency periods

Our Solutions: S_{local} and S_{move}

Both solutions behave similarly on **our case study**

Our Solutions: S_{local} and S_{move}

Both solutions behave similarly on **our case study**

S_{local} is more aggressive and simple (3 lines of code),
changes the behavior of CFS and heavily relies on **periodic load balancing** to fix mistakes

Our Solutions: S_{local} and S_{move}

Both solutions behave similarly on **our case study**

S_{local} is more aggressive and simple (3 lines of code),
changes the behavior of CFS and heavily relies on **periodic load balancing** to fix mistakes

S_{move} is more balanced, and accounts for **frequency**,
more complicated (124 lines of code, timers), but keeps the overall ideas of CFS

Performance and Energy Evaluation

60 applications from:

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

2 machine markets:

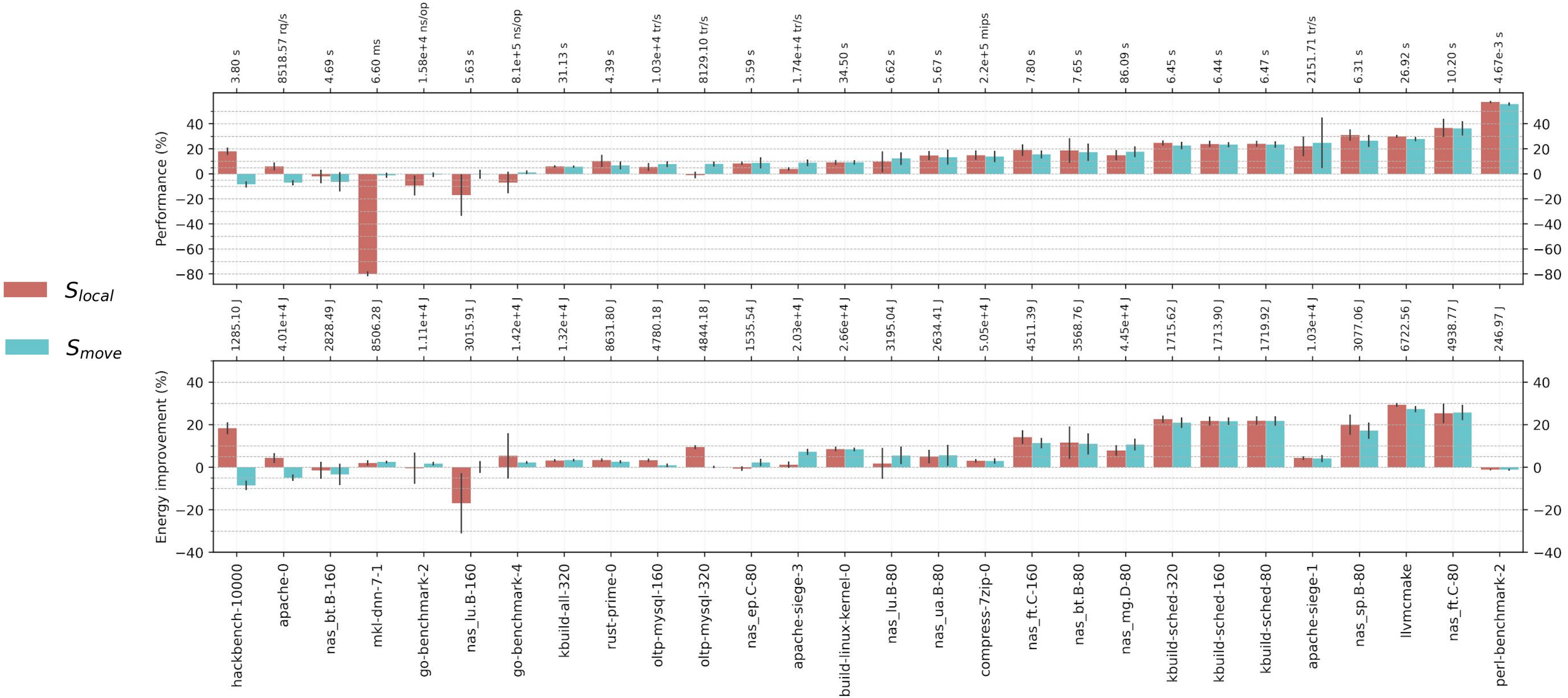
- **Server**: 80-core Intel® Xeon E7-8870 v4 (160 HW threads)
- **Desktop**: 4-core AMD® Ryzen 5 3400G (8 HW threads)

2 frequency scaling governors:

- **powersave**
- **schedutil**

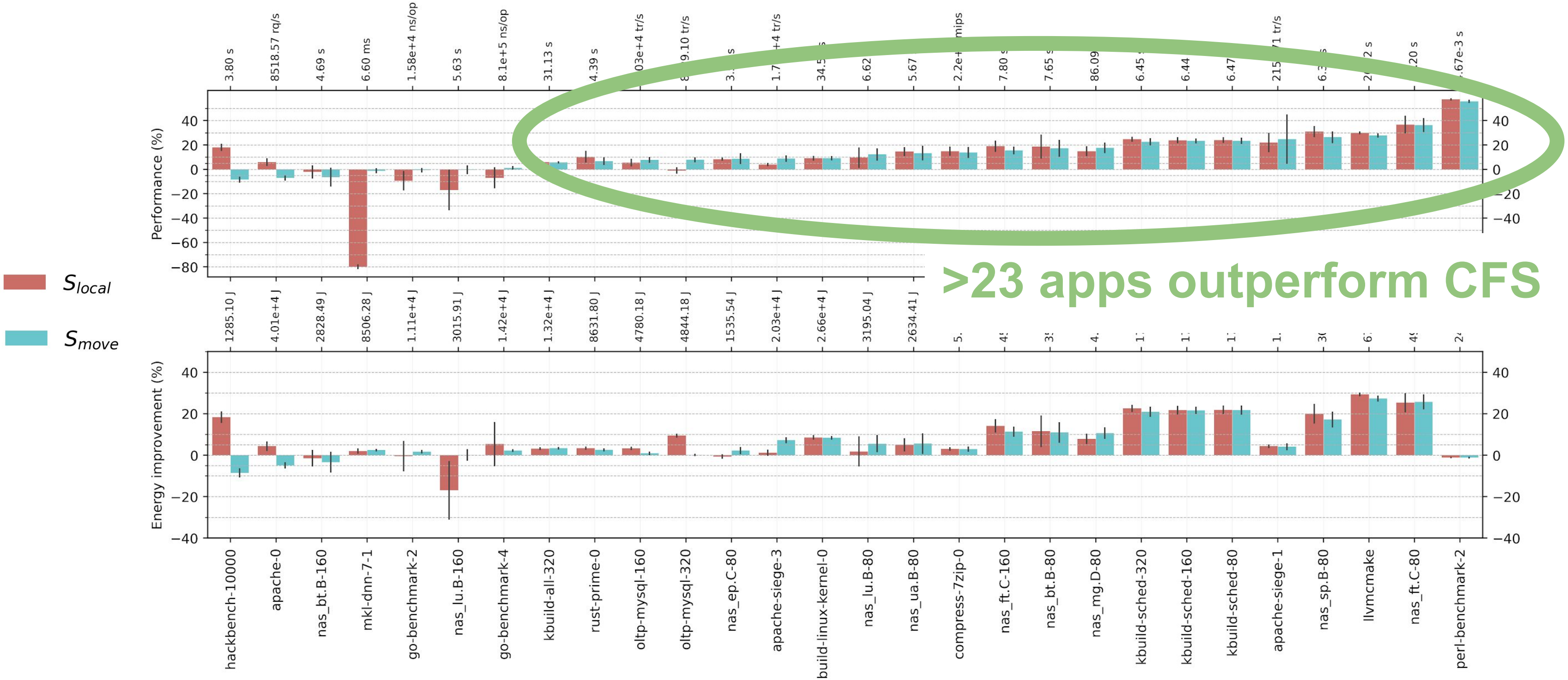
Performance and Energy Evaluation

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



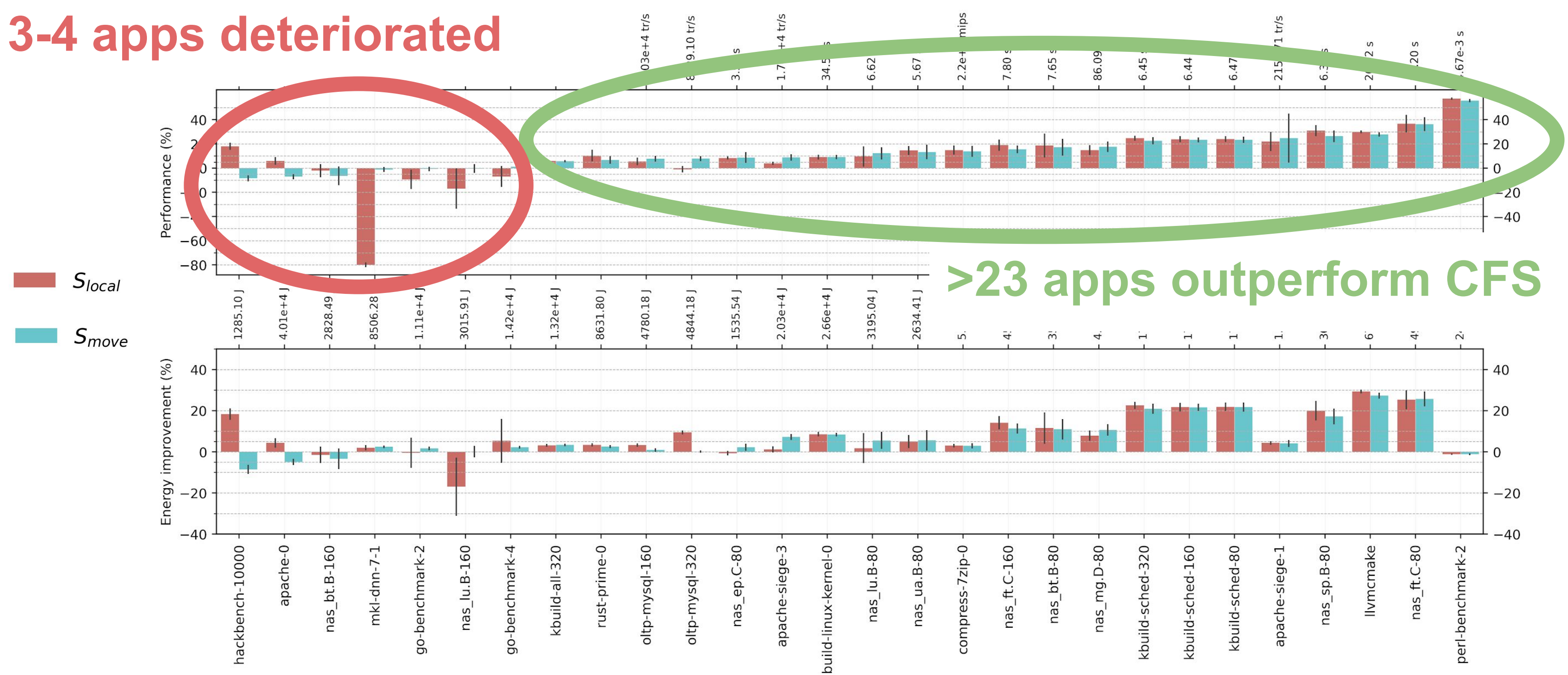
Performance and Energy Evaluation

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



Performance and Energy Evaluation

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



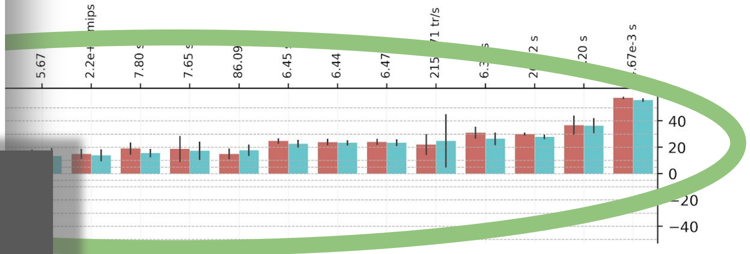
Performance and Energy Evaluation

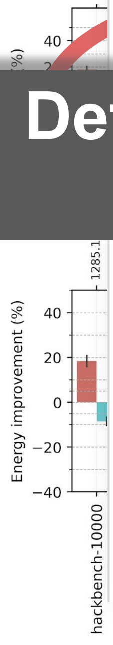
Compared to CFS, server machine, powersave governor, higher is better

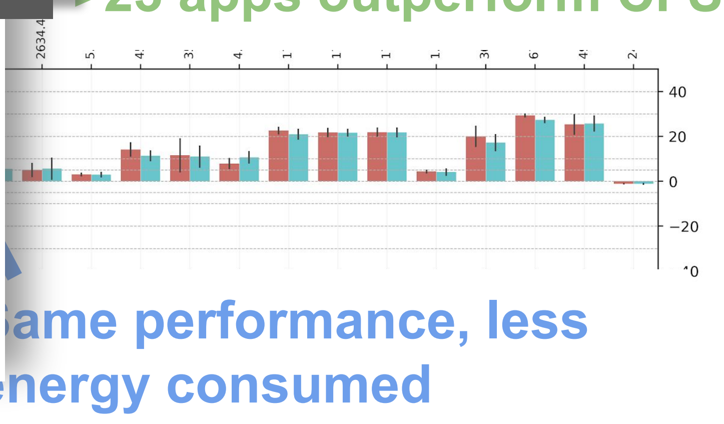
Perform… …valuation

Compared … …ave governor, higher is better

3-4 apps d…







$S_{lo…}$

S_{move}

Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

Redha Gouicem, Damien Carver — *Sorbonne University, LIP6, Inria*
Jean-Pierre Lozi — *Oracle Labs*
Julien Sopena — *Sorbonne University, LIP6, Inria*
Baptiste Lepers, Willy Zwaenepoel — *University of Sydney*
Nicolas Palix — *Université Grenoble Alpes*
Julia Lawall, Gilles Muller — *Inria, Sorbonne University, LIP6*

Abstract

In modern server CPUs, individual cores can run at different frequencies, which allows for fine-grained control of the performance/energy tradeoff. Adjusting the frequency, however, incurs a high latency. We find that this can lead to a problem

…

for 23 applications, and worsens performance by more than 5% (at most 8%) for only 3 applications. On a 4-core AMD Ryzen we obtain performance improvements up to 56%.

1 Introduction

Striking a balance between performance and energy consumption has long been a battle in the development of computing systems. For several decades, CPUs have supported Dynamic Frequency Scaling (DFS), allowing the hardware or the software to update the CPU frequency at runtime. Reducing CPU frequency can reduce energy usage, but may also decrease overall performance. Still, reduced performance may be acceptable for tasks that are often idle or are not very urgent, making it desirable to save energy by reducing the frequency in many use cases. While on the first multi-core machines, all cores of a CPU had to run at the same frequency, recent server CPUs from Intel® and AMD® make it possible to update the frequency of individual cores. This feature allows for much finer-grained control, but also raises new challenges.

One source of challenges in managing core frequencies is the *Frequency Transition Latency (FTL)*. Indeed, transitioning a core from a low to a high frequency, or conversely, has an FTL of dozens to hundreds of milliseconds. FTL leads to a problem of *frequency inversion* in scenarios that are typical of the use of the standard POSIX fork() and wait() system

…

until recently, and C_{CFS}, on which T_{woken} is running, is likely to be executing at a low frequency because it was previously idle. Consequently, the frequencies at which C_{waker} and C_{CFS} operate are *inverted* as compared to the load on the cores. This frequency inversion will not be resolved until C_{waker} reaches a low frequency and C_{CFS} reaches a high frequency, *i.e.*, for the duration of the FTL. Current hardware and software DFS policies, including the schedutil policy [9] that was recently added to CFS cannot prevent frequency inversion as their only decisions consist in updating core frequencies, thus paying the FTL each time. Frequency inversion reduces performance and may increase energy usage.

In this paper, we first exhibit the problem of frequency inversion in a real-world scenario through a case study of the behavior of CFS when building the Linux kernel on a Intel® Xeon-based machine with 80 cores (160 hardware threads). Our case study finds repeated frequency inversions when processes are created through the fork() and wait() system calls, and our profiling traces make it clear that frequency inversion leads to tasks running on low frequency cores for a significant part of their execution.

Detailed analysis in the paper!

>23 apps outperform CFS

Same performance, less energy consumed

Take away

Frequency inversion problem

- FTL + frequency agnostic scheduler
- New because of per-core dynamic frequency scaling

Solutions implemented in Linux

- S_{local}: simple, aggressive, relies on load balancing
- S_{move}: frequency-aware, more balanced
- Both are available at: https://gitlab.inria.fr/whisper-public/atc20

Possible extensions:

- Fully frequency aware scheduler
- Modeling the frequency behavior of a CPU (#active cores, temperature, instruction set, …)
- Shortening FTL with faster frequency reconfiguration