A Blocking Bound for Nested FIFO Spin Locks

Alessandro Biondi,[*†] Björn B. Brandenburg[*]
and Alexander Wieder[*]

[*] *MPI-SWS, Kaiserslautern, Germany*

[†] *Scuola Superiore Sant'Anna, Pisa, Italy*

Max Planck Institute for Software Systems

Retis Real-Time Systems Laboratory

THIS TALK

1

Analysis of **nested locks**: a **practical** but **very difficult** problem

Show effects that do **not happen** with **non-nested** locks

2

This work: a novel analysis method

The first **fine-grained** analysis for **nested** FIFO spin locks

3

Experimental Evaluation

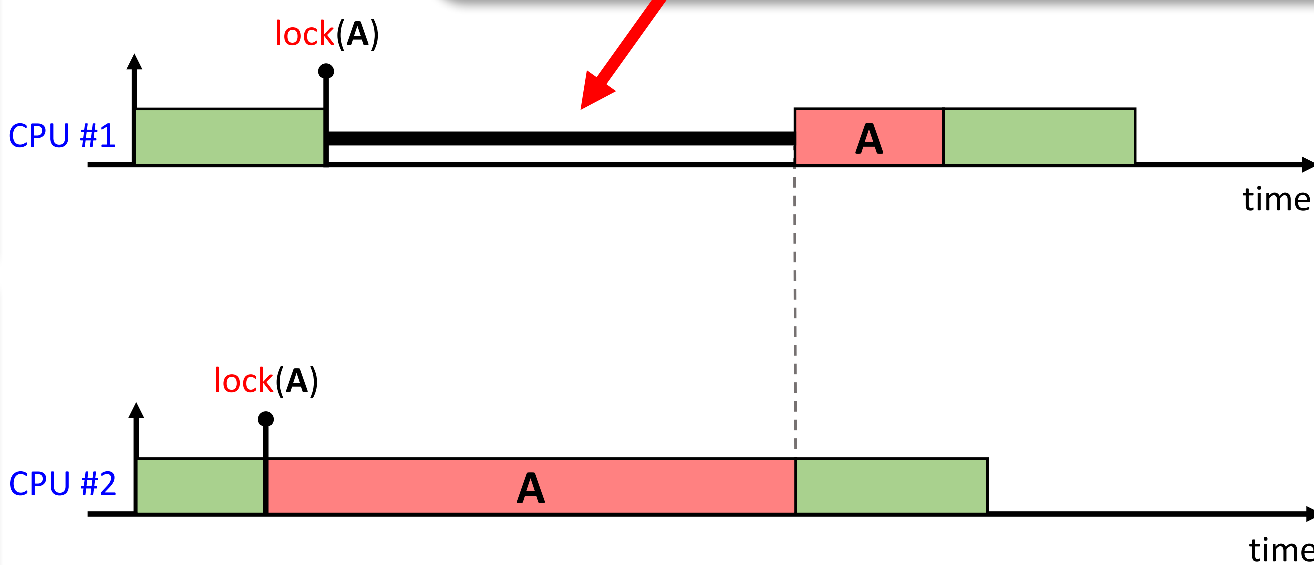
INTRODUCTION

- Bounding the worst-case blocking time due to *lock contention* is a **fundamental problem** in the analysis of multiprocessor real-time systems

Blocking analysis problem
bound the **delay** incurred by tasks due to lock contention

```
TASK(Task1)
{
    lock(A)
    <…>
    unlock(A)
}
```

```
TASK(Task2)
{
    lock(A)
    <…>
    unlock(A)
}
```



NESTED LOCKS

```
TASK(Task2)
{
    lock(A)
        lock(B)
        <…>
        unlock(B)
    unlock(A)
}
```

Nested critical section

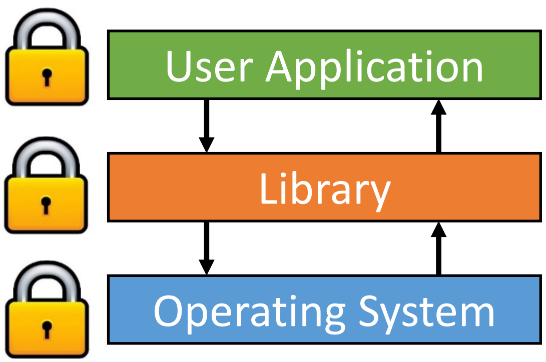
- Concerning **nested locks**, limited progress has been made in 25+ years of research on multiprocessor real-time synchronization

  Notable exceptions: *Ward and Anderson (RNLP), Takada and Sakamura (scalability of nested spin locks), Faggioli et al. (MBWI)*

- **No** fine-grained analysis was available, even for simple (and widely adopted) lock types such as FIFO spin locks

MOTIVATION

The analysis of **nested locks** is a **practically** relevant problem as *nesting* is not a rarity in many real-world systems



Nesting may happen *unintentionally* due to the natural **layering** of well-structured software

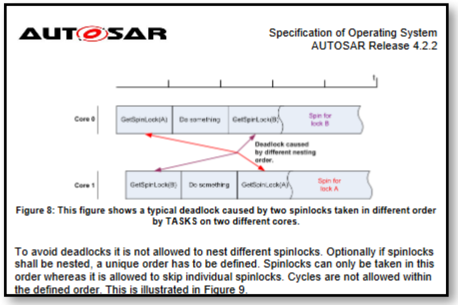

Specification of Operating System
AUTOSAR Release 4.2.2

Figure 8: This figure shows a typical deadlock caused by two spinlocks taken in different order by TASKS on two different cores.

To avoid deadlocks it is not allowed to nest different spinlocks. Optionally if spinlocks shall be nested, a unique order has to be defined. Spinlocks can only be taken in this order whereas it is allowed to skip individual spinlocks. Cycles are not allowed within the defined order. This is illustrated in Figure 9.

Nested locks are officially supported by **standards** (e.g., AUTOSAR)

ANALYZING NESTED LOCKS IS HARD!

A VERY CHALLENGING PROBLEM

Computational complexity

Even simple blocking analysis problems with nested locks on multiprocessors are **NP-HARD**

A. Wieder and B. B. Brandenburg, "*On the complexity of worst-case blocking analysis of nested critical sections*", RTSS 2014

Human intuition

Reasoning about the blocking generated by nested locks is **very difficult** due to a number of complications that do not arise in conventional (*non-nested*) analyses

COMPLICATIONS DUE TO NESTING

Let's see 3 **examples** of negative phenomena that can happen with **nested spin locks**, <u>but not</u> with typical *non-nested* ones

- **Transitive blocking**
- **Scheduling anomalies**
- **Implicit serializations**

TRANSITIVE BLOCKING

CPU #1

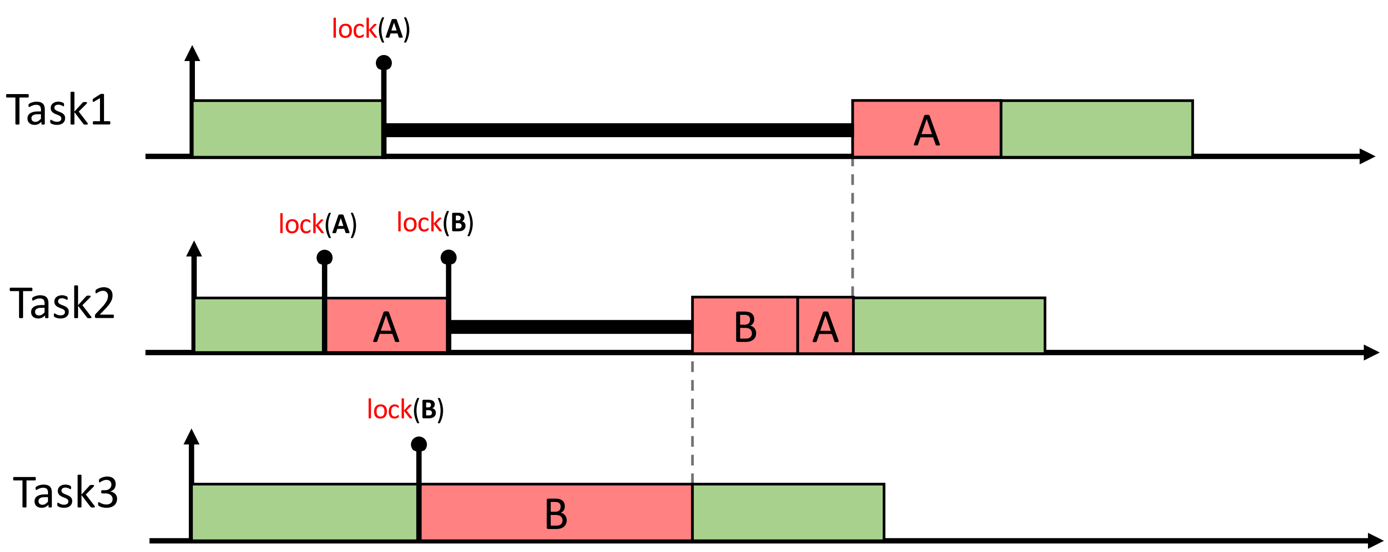
```
TASK(Task1)
{
    lock(A)
    <…>
    unlock(A)
}
```

CPU #2

```
TASK(Task2)
{
    lock(A)
        lock(B)
        <…>
        unlock(B)
    unlock(A)
}
```

CPU #3

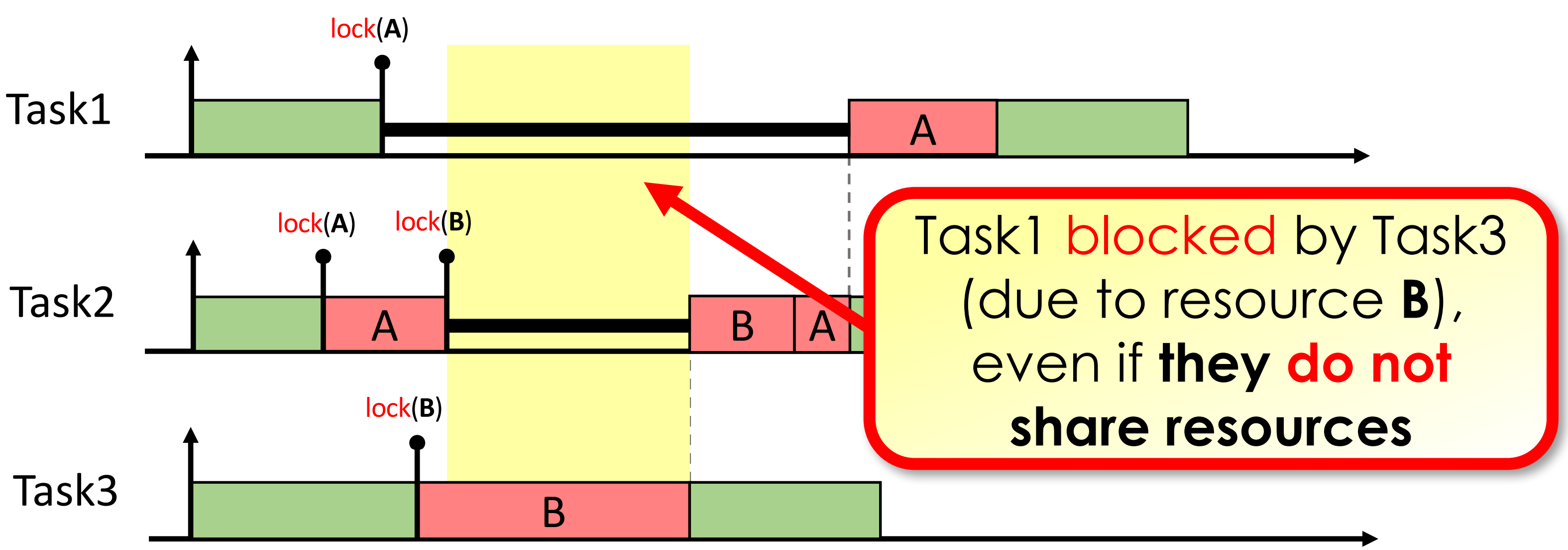
```
TASK(Task3)
{
    lock(B)
    <…>
    unlock(B)
}
```



TRANSITIVE BLOCKING

CPU #1	CPU #2	CPU #3

TASK(Task1)
{
lock(**A**)
<…>
unlock(**A**)
}
	TASK(Task2)
	{
	lock(**A**)
	lock(**B**)
	<…>
	unlock(**B**)
	unlock(**A**)
	}
		TASK(Task3)
		{
		lock(**B**)
		<…>
		unlock(**B**)
		}



TRANSITIVE BLOCKING

CPU #1

```
TASK(Task1)
{
    lock(A)
    <...>
    unlock(A)
}
```

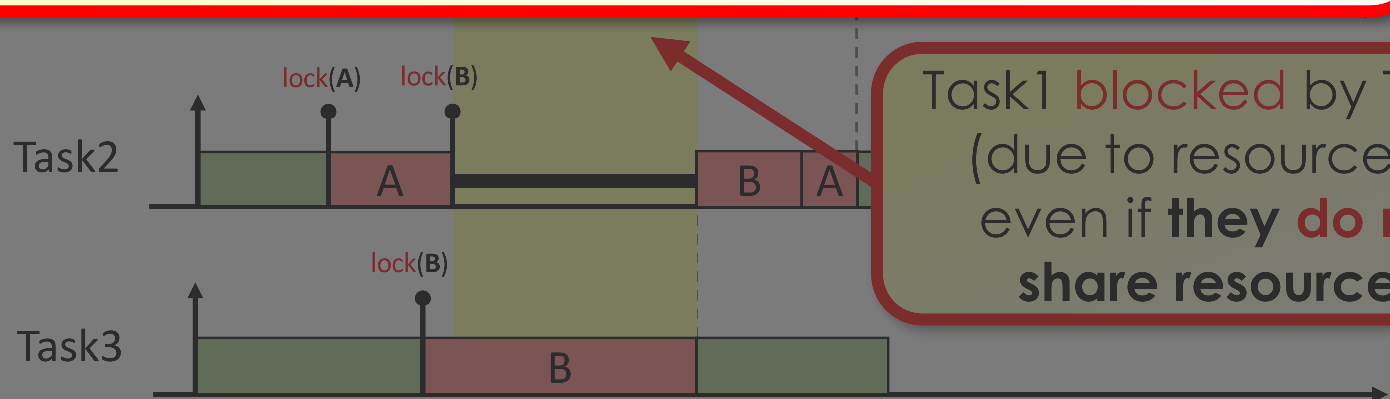
CPU #2

```
TASK(Task2)
{
    lock(A)
        lock(B)
        <...>
        unlock(B)
```

CPU #3

```
TASK(Task3)
{
    lock(B)
    <...>
    unlock(B)
}
```

In the presence of **nested** critical sections, tasks may experience **transitive blocking**

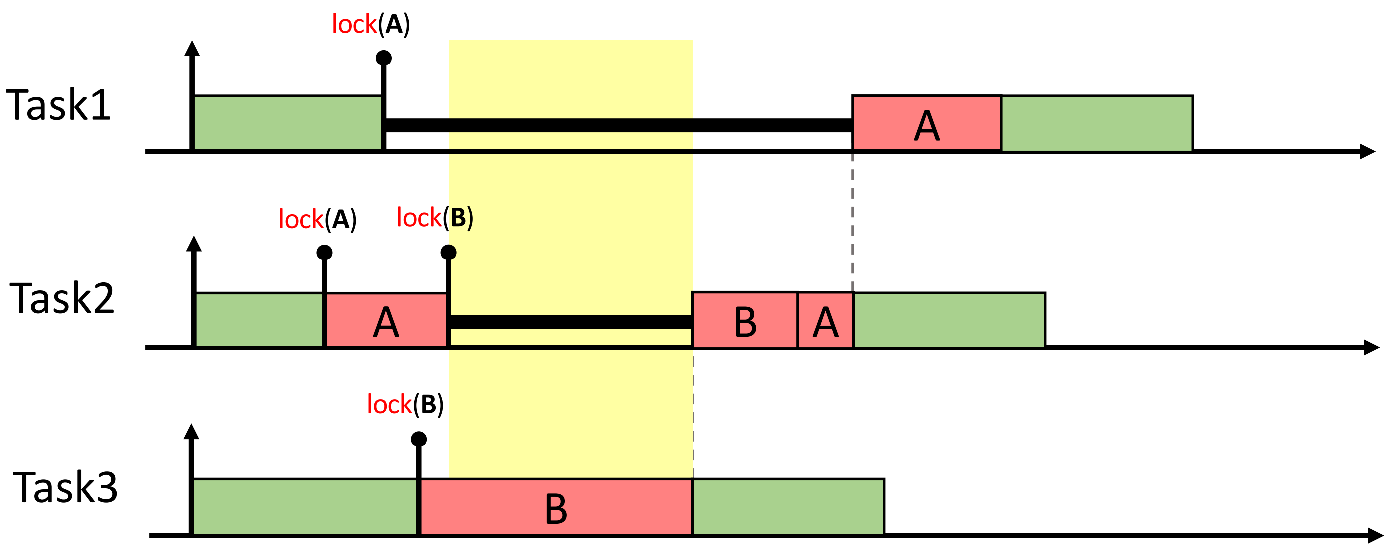


Task1 blocked by Task3 (due to resource **B**), even if **they do not share resources**

TRANSITIVE BLOCKING

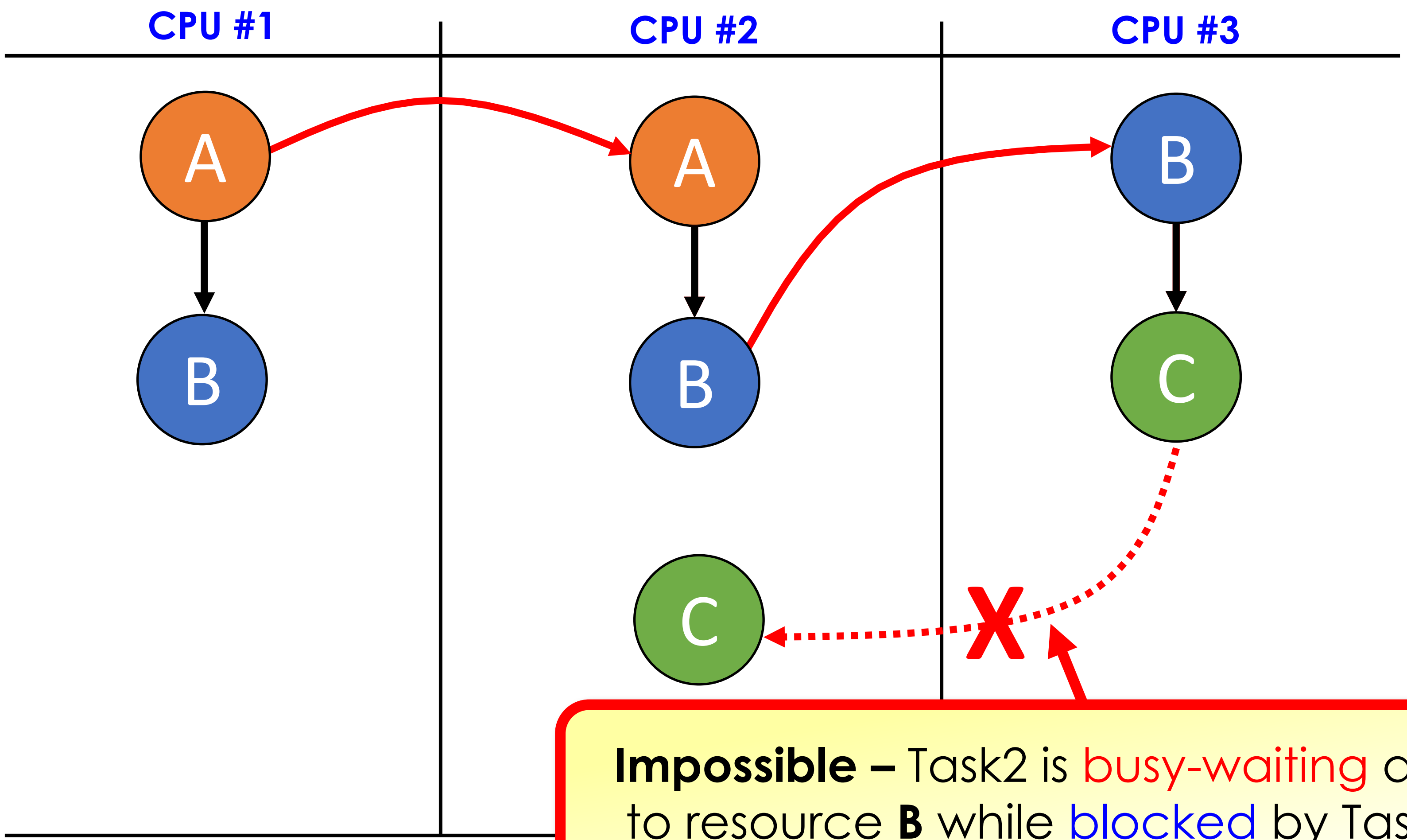
CPU #1	CPU #2	CPU #3

TASK(Task1)
{
lock(**A**)
<…>
unlock(**A**)
}
	TASK(Task2)
	{
	lock(**A**)
	lock(**B**)
	<…>
	unlock(**B**)
	unlock(**A**)
	}
		TASK(Task3)
		{
		lock(**B**)
		<…>
		unlock(**B**)
		}



SCHEDULING ANOMALIES

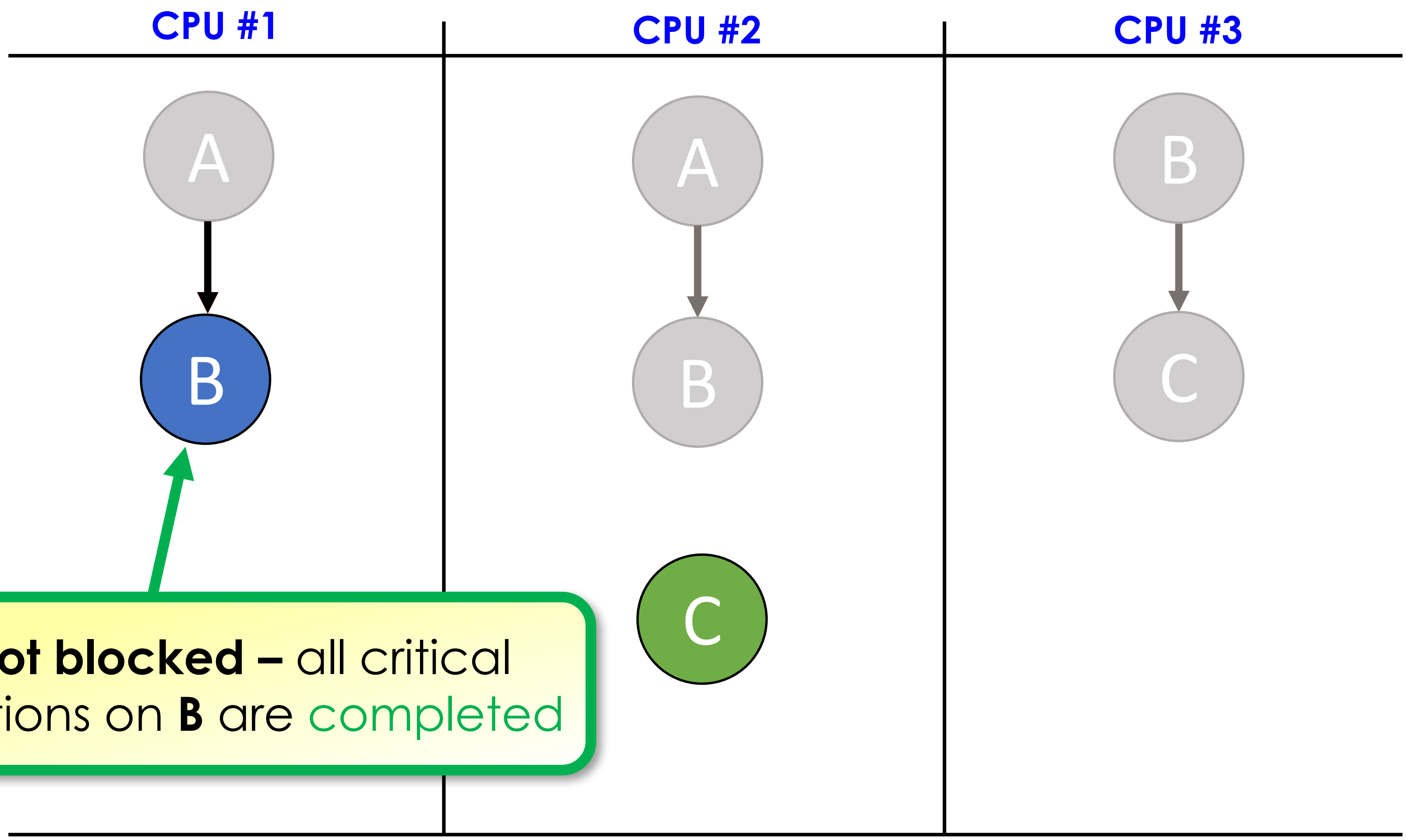
How much is the **blocking** incurred by **Task1** due to resources **A** and **B**?



Impossible – Task2 is busy-waiting due to resource **B** while blocked by Task3

SCHEDULING ANOMALIES

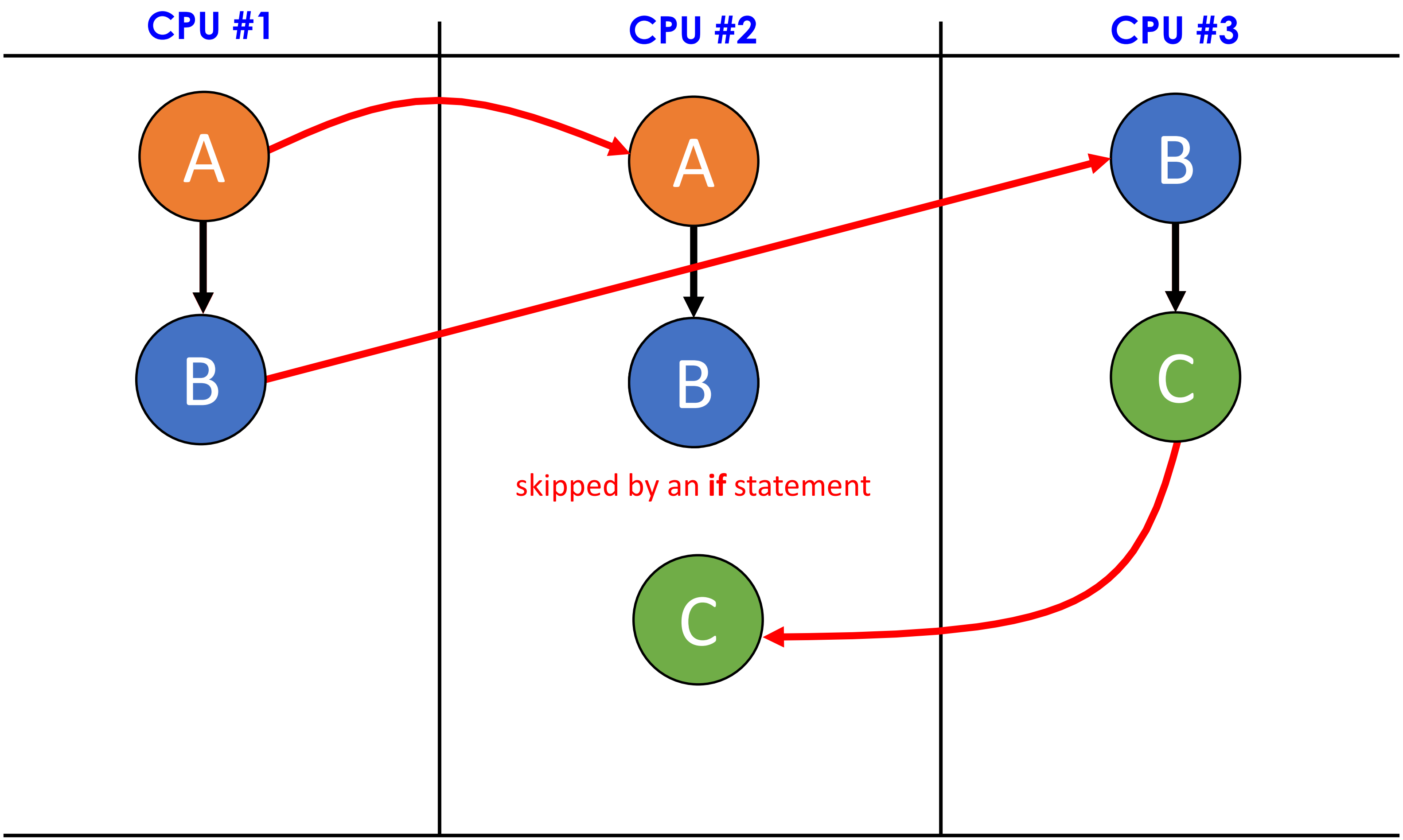
How much is the **blocking** incurred by **Task1** due to resources **A** and **B**?



Not blocked – all critical sections on **B** are completed

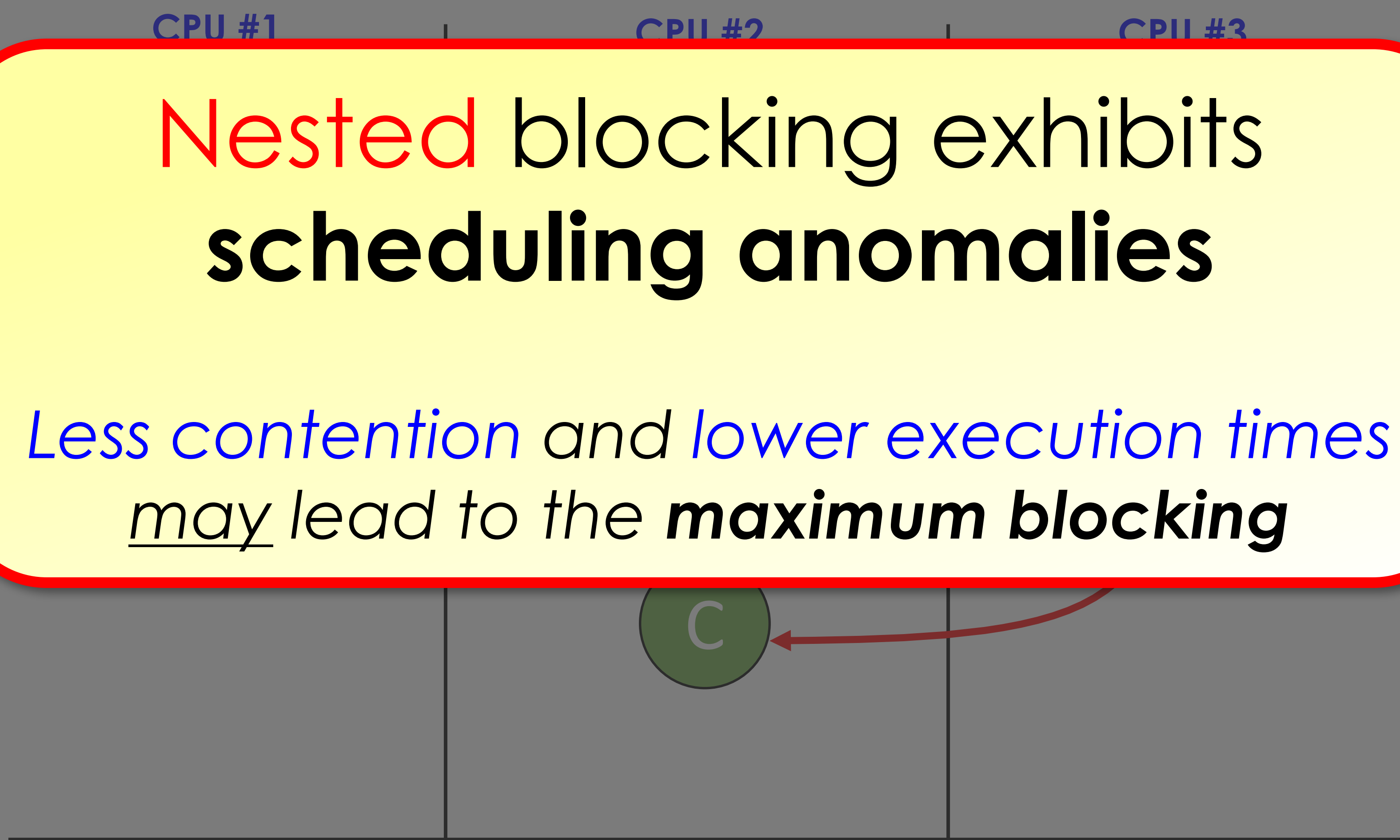
SCHEDULING ANOMALIES

How much is the **blocking** incurred by **Task1** due to resources **A** and **B**?



SCHEDULING ANOMALIES

How much is the **blocking** incurred by **Task1** due to resources **A** and **B**?

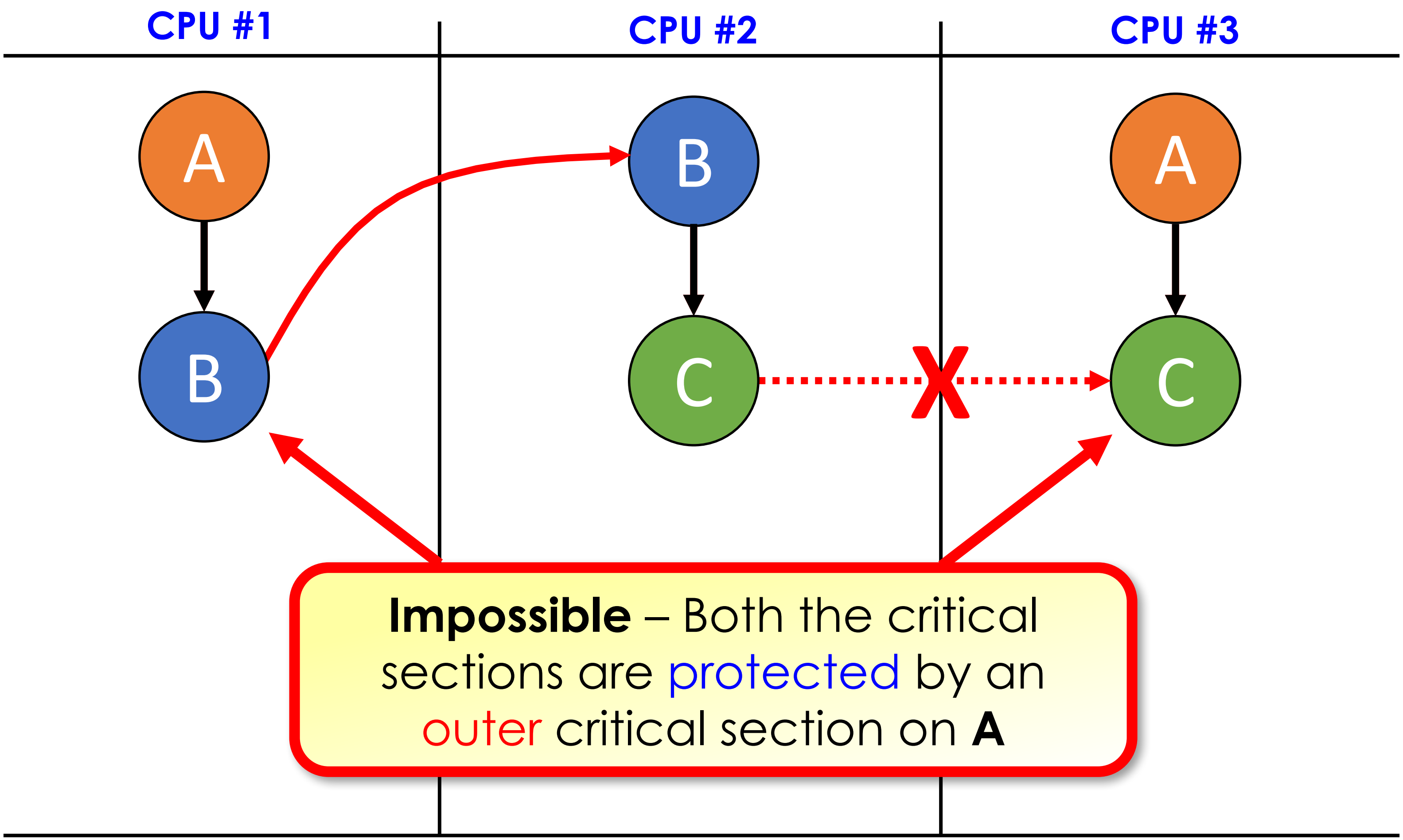


Nested blocking exhibits **scheduling anomalies**

Less contention and lower execution times *may* *lead to the* ***maximum blocking***

IMPLICIT SERIALIZATIONS

Can Task1 be **transitively** blocked by Task3?



Impossible – Both the critical sections are protected by an outer critical section on **A**

IMPLICIT SERIALIZATIONS

Can Task1 be **transitively** blocked by Task3?

CPU #1

CPU #2

CPU #3

Some blocking interactions are **impossible** due to **implicit serializations**, which depend on the *"path"* reaching a critical section - thus making local reasoning ineffective

Impossible – Both the critical sections are protected by an outer critical section on **A**

NEED FOR NOVEL TECHNIQUES

- X **Transitive blocking**
- X **Scheduling anomalies**
- X **Implicit serializations**
- X …

Existing approaches fail in capturing **fundamental** aspects of the problem

THIS WORK

A **novel** analysis method to bound
the worst-case blocking in the presence
of **nested locks**

CONSIDERED SETTING

- Partitioned Fixed-Priority (**P-FP**) scheduling
- Shared resources protected by the Multiprocessor Stack Resource Policy (**MSRP**), but allowing nested locks (forbidden by the original protocol)

Focus on non-preemptive FIFO spin locks

A >> B >> C >> D

Given **lock order** to avoid deadlock

- Typical **good practice** (e.g., any violations in the Linux kernel are flagged as serious bugs)
- Explicitly mandated by **AUTOSAR** (the order must be specified in the OIL configuration)

```
lock(A);
    lock(B)
    <...>
    unlock(B)
unlock(A)
```

Fully contained critical sections

PROPOSED APPROACH

To tackle the intrinsic complexity of the problem, we proposed a novel analysis approach based on **4 steps**

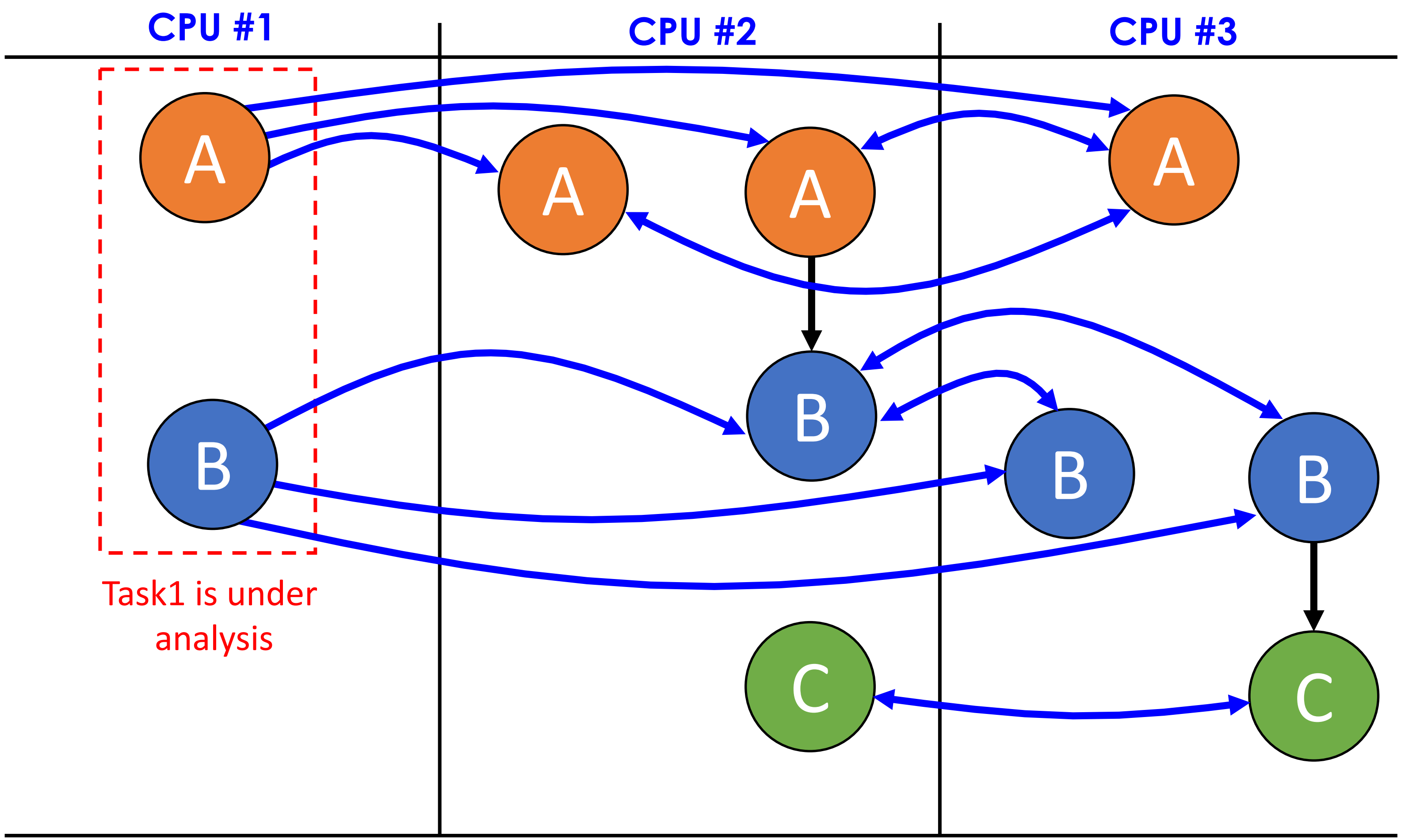
1. Definition of a novel **graph abstraction** that encodes *all* possible blocking interactions
2. **Mapping** from schedules to instances of the graph abstraction, which yield schedule-specific blocking bounds
3. Identification of **invariants** that must hold for any **valid** instance of the graph
4. Computation of a **maximal subgraph** that dominates *all possible* valid graph instances, thus obtaining a **safe** blocking bound

STEP 1 – STATIC BLOCKING GRAPH

Unambiguously model **all** possible **blocking interactions** for a given task, eliding irrelevant details

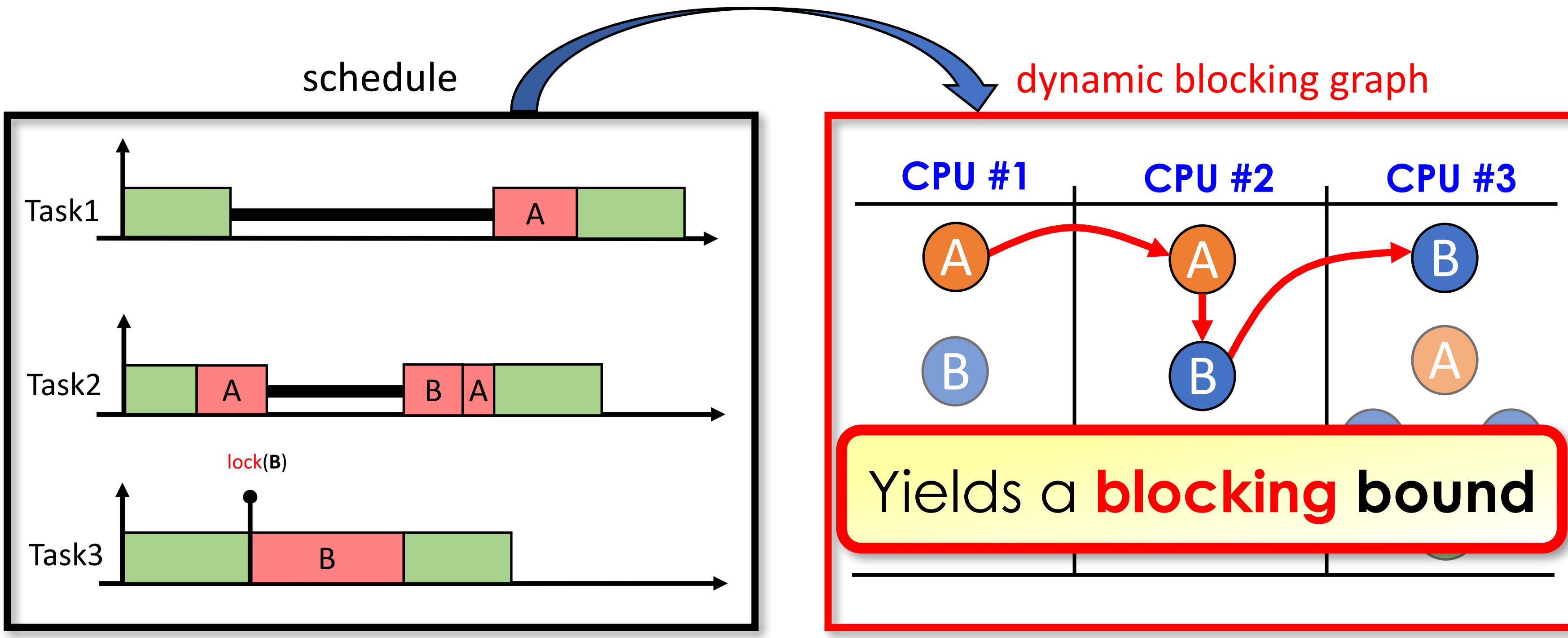
STEP 1 – STATIC BLOCKING GRAPH

Unambiguously model **all** possible **blocking interactions** for a given task, eliding irrelevant details



STEP 2 – DYNAMIC BLOCKING GRAPH

We established a **mapping** between an arbitrary (but fixed) schedule and an instance of the **graph-based abstraction**



STEP 3 – INVARIANTS

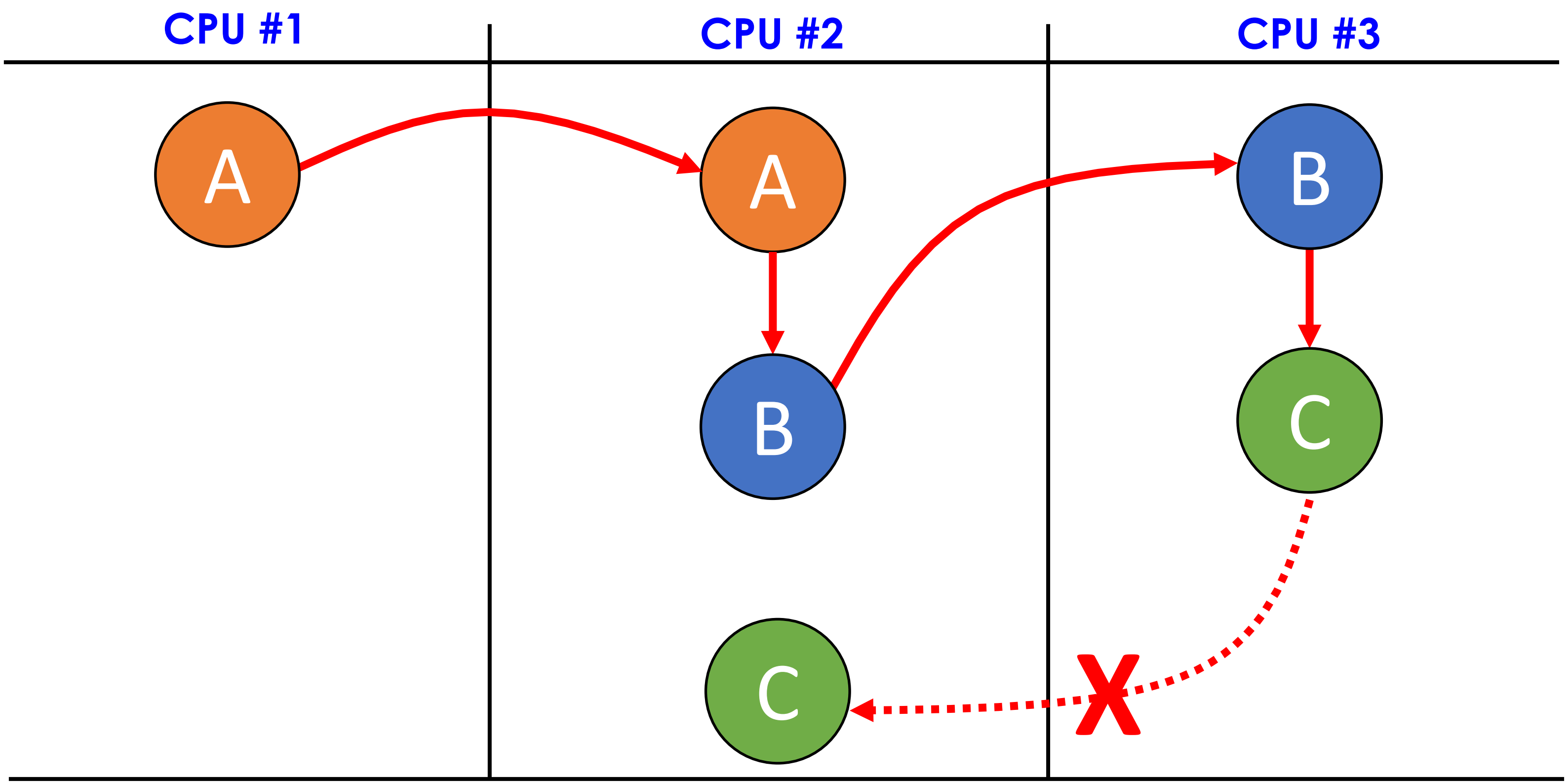
- We proved 13 **invariants**, i.e., structural properties that hold in all possible **valid** dynamic blocking graphs
- *In other words*, there cannot exists a valid dynamic blocking graph which violates such invariants

Why the invariants?

- Lots of **limited-scope reasoning**
- Provide precise foundations for
  - **rigorous proofs** on the graph-based model
  - analysis **safe by construction** ruling out impossible scenarios

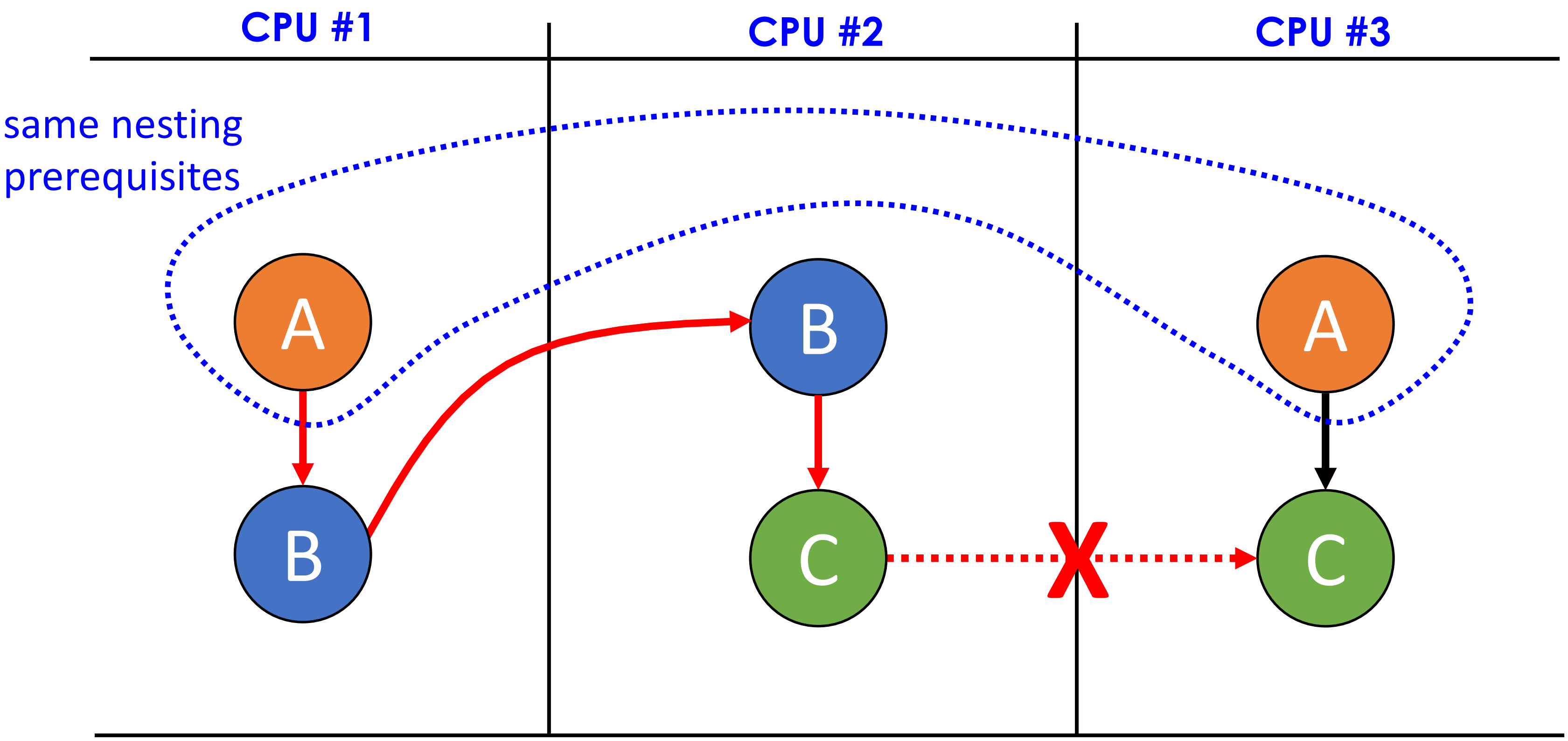
EXAMPLE OF INVARIANT

In any **valid** dynamic blocking graph there cannot be paths that **circle back** to **already visited processors**



EXAMPLE OF INVARIANT

In any **valid** dynamic blocking graph there cannot be paths connecting critical sections in different processors that **share** the same **nesting prerequisites**



STEP 4 – MAXIMAL SUBGRAPH

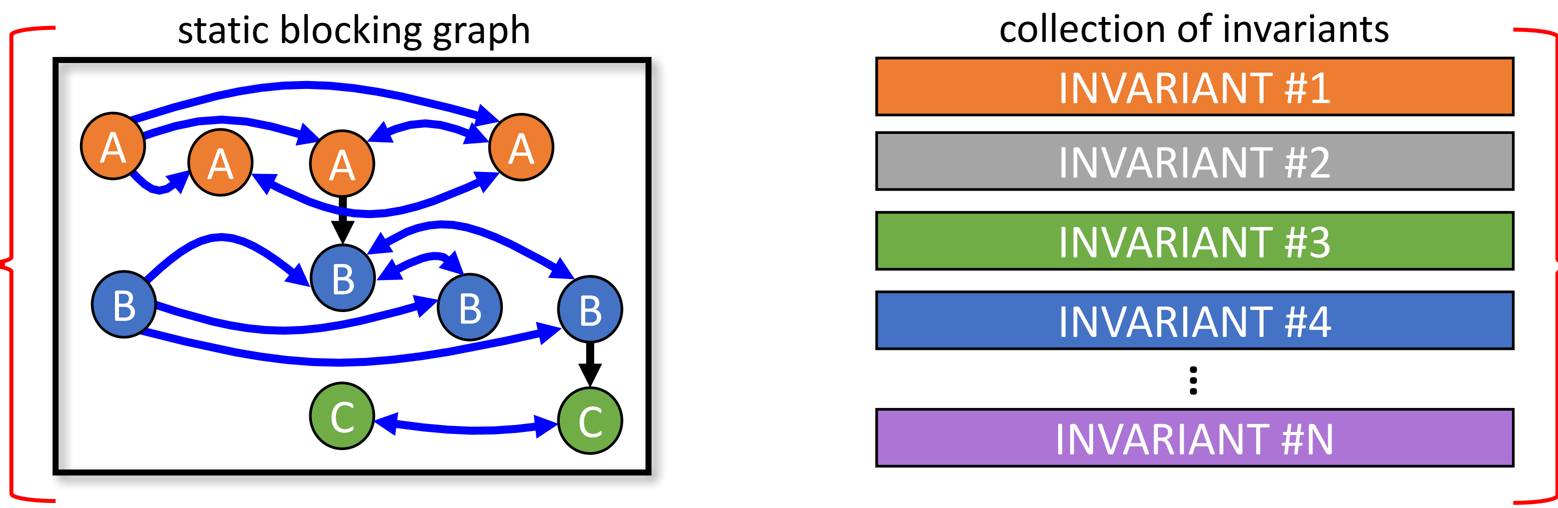
Goal: Compute a **safe** blocking bound, i.e., coping with the **maximum** blocking in every possible schedule

each *schedule* corresponds to a dynamic blocking graph, which is a **subgraph** of the static blocking graph

If we find a **maximal subgraph**, which dominates all possible dynamic graphs, we have a **safe** blocking bound

STEP 4 – MAXIMAL SUBGRAPH

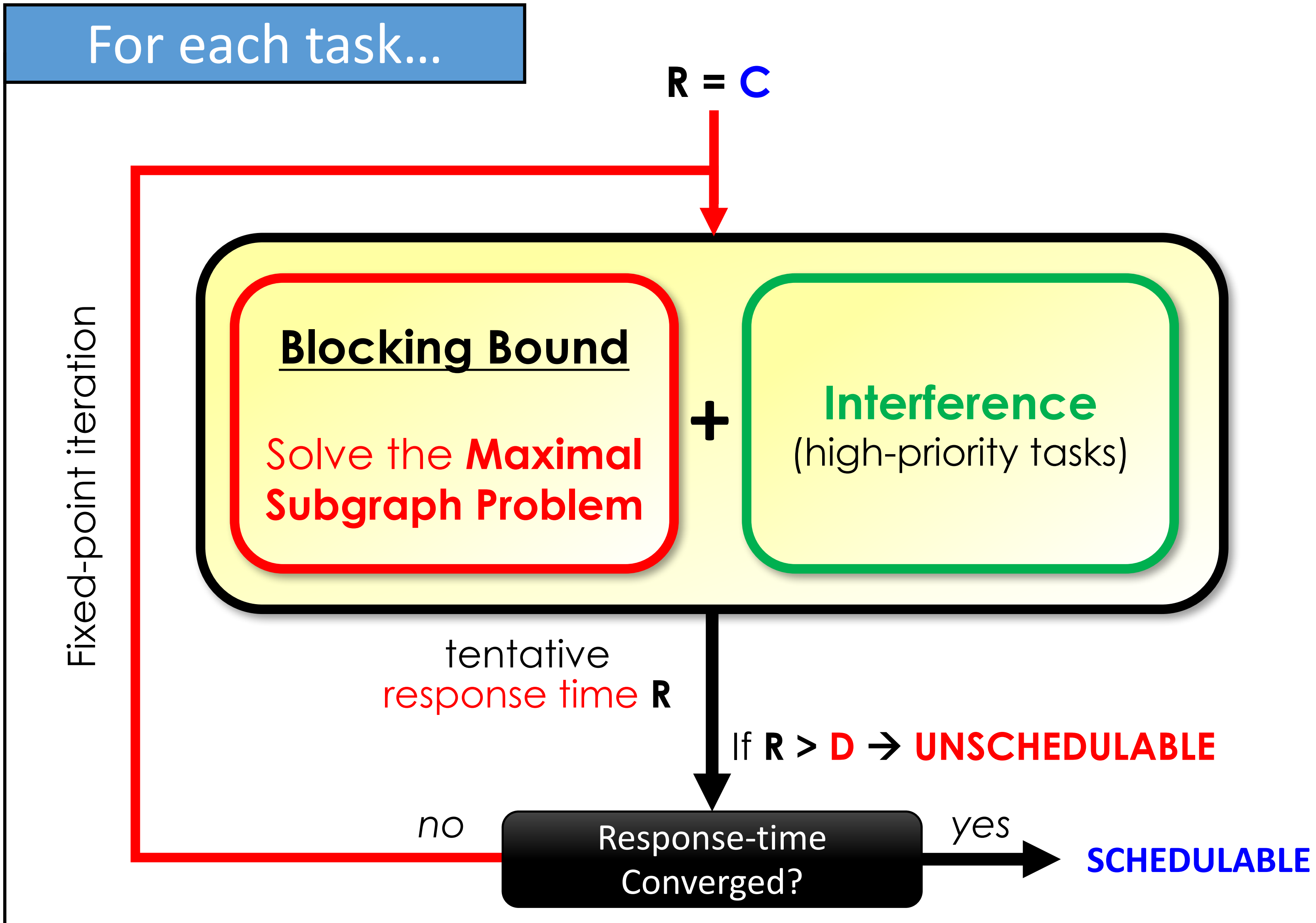
To find the **maximal subgraph** we can maximize the blocking out of all possible *subgraphs* **not** excluded by a set of constraints derived from the **invariants**



Optimization problem (ILP) to find the **maximal subgraph**

Problem structure	**Constraints**
• variables • obj. function	• … • …

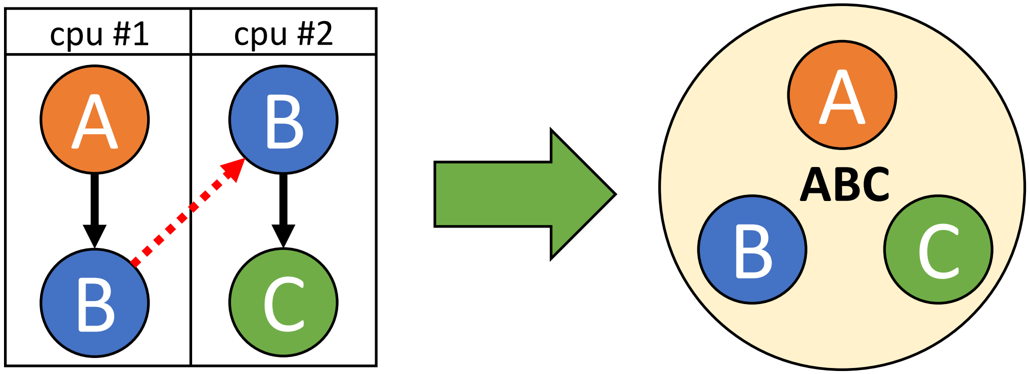
IMPLEMENTATION



EVALUATION

EMPIRICAL EVALUATION

- Tested *synthetic workload* under different configurations of the workload generator
- Comparison with **group locks** – reduce fine-grained nested critical sections into coarse-grained non-nested ones

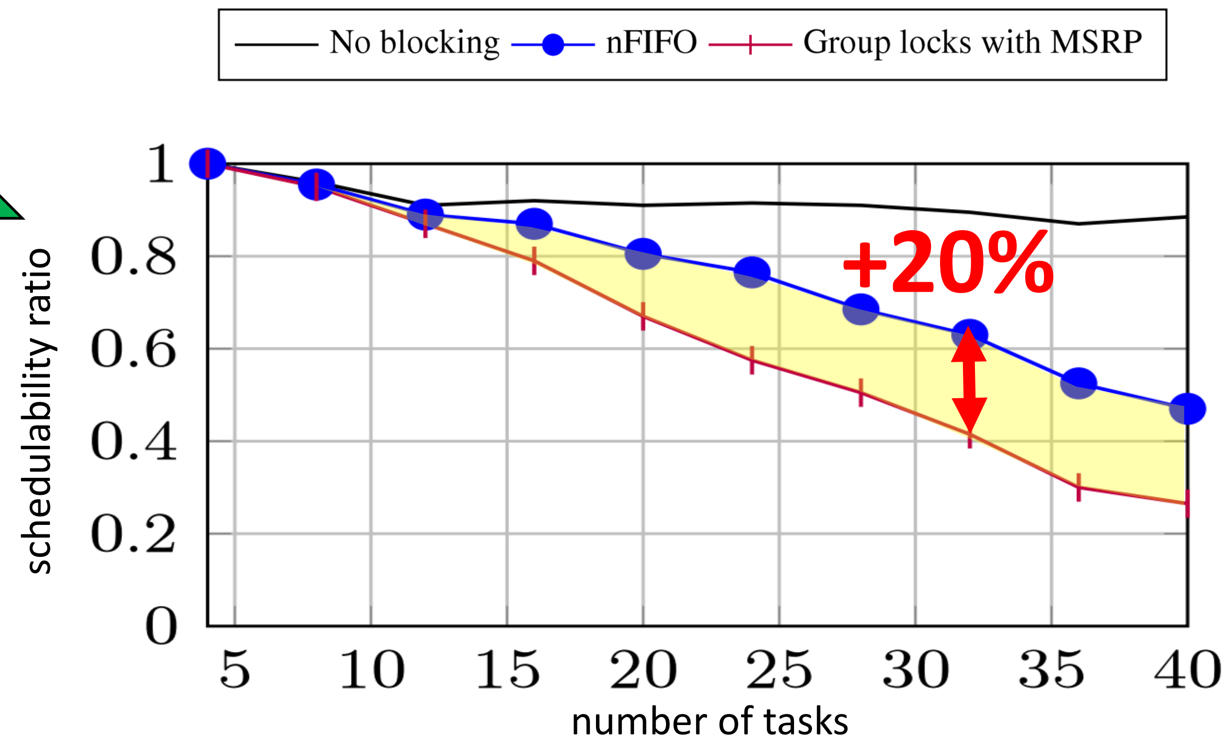


Used analysis for non-nested spin locks: A. Wieder and B. B. Brandenburg, "*On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks*", RTSS 2013

SCHEDULABILITY PERFORMANCE (1)

The **higher** the **better**

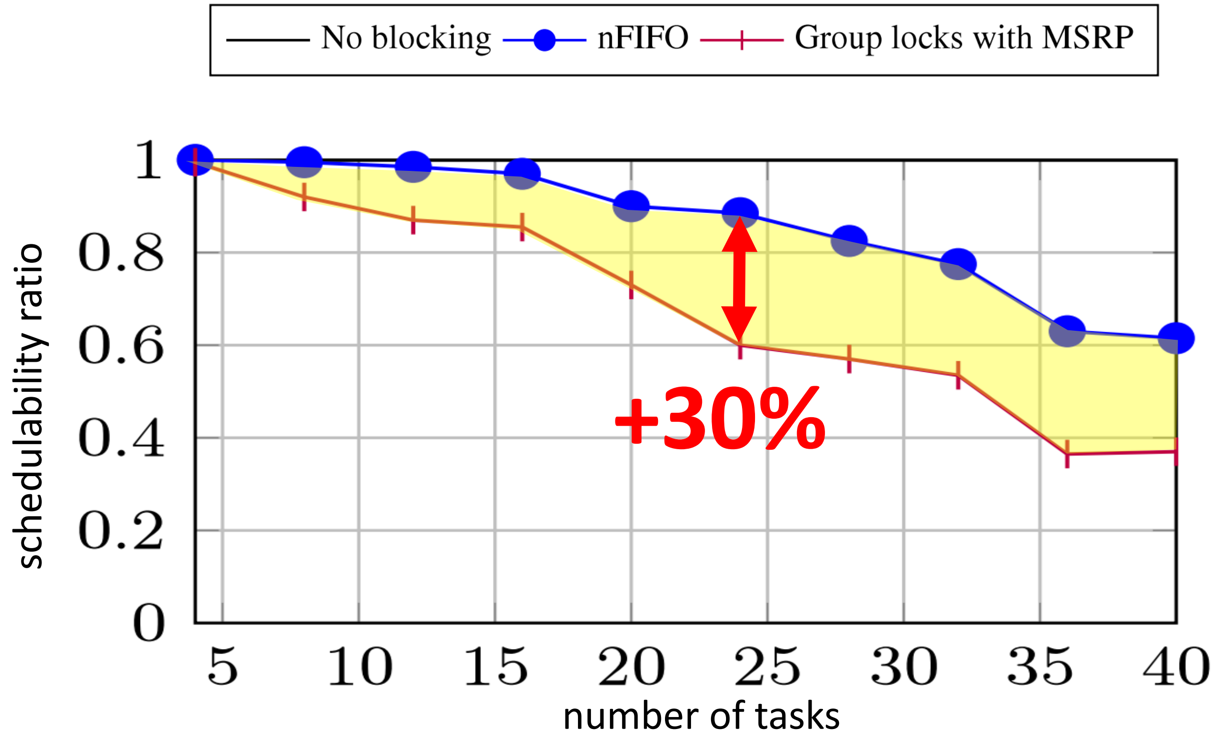
- ...s/task
- utilization ∈ [0.7,0.9]
- up to **2** nesting levels



Contextual increase of critical sections and hence **contention**

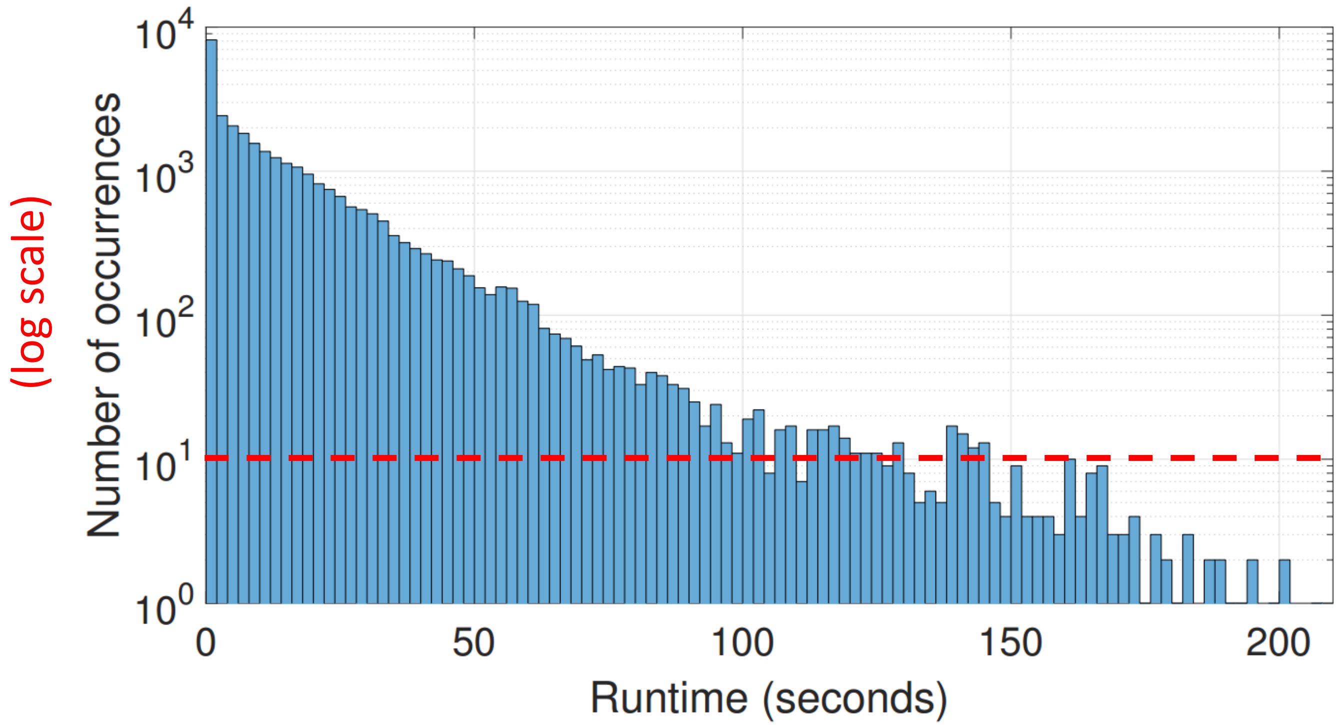
SCHEDULABILITY PERFORMANCE (2)

- **4** processors
- **16** resources
- at most **1** request/task
- utilization ∈ [0.5,0.7]
- up to **4** nesting levels



RUNTIME

Most instances of the maximal subgraph problem have been solved in **less than 2 seconds**.
Only **<2%** exceeded 100 seconds.



IBM CPLEX, 32-core Intel Xeon E5 @ 3.3 GHz

CONCLUSIONS

- First fine-grained analysis for **nested** FIFO non-preemptive spin locks under **P-FP**
- Proposed a novel **graph-based abstraction** that encode all possible blocking interactions in the presence of nesting
- Identified the **structure** of valid instances of the graph-based abstraction
- Computation of a blocking bound by identifying a **maximal subgraph**

If applied to **non-nested** spin locks, this analysis is as **accurate** as the one previously proposed in RTSS'13

A. Wieder and B. B. Brandenburg, "*On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks*", RTSS 2013

A LOOK FORWARD

The graph abstraction proposed in this work can be used to solve other blocking analysis problems in the presence of **nesting**

- **Examples**: nested semaphores, *real-time nested locking protocol* (RNLP), preemptive nested spin locks, MrsP,…

- The application of this analysis method to these mechanisms is our future work

Thank you!

Alessandro Biondi
alessandro.biondi@sssup.it

Max
Planck
Institute
for
Software Systems