ELISA
Enabling Linux in Safety Applications

WORKSHOP

ELISA Workshop
Munich, Germany

November 18-20, 2025
Co-hosted with Red Hat

Towards Practical Program Verification for the Linux Kernel

Keisuke Nishimura, Jean-Pierre Lozi, Julia Lawall

Outline

- Overview: What is verification?
- Case Study: Applying verification to a small function, is_core_idle()
- Towards practical verification of the Linux kernel

Outline

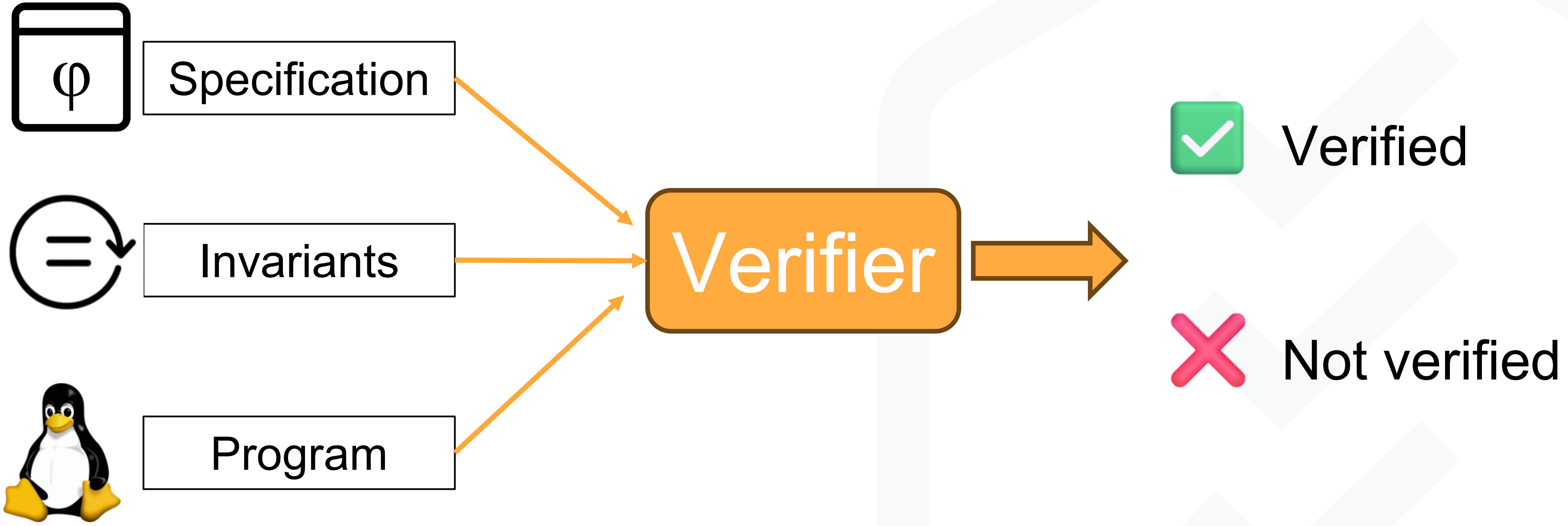
- Overview: What is verification?
- Case Study: Applying verification to a small function, is_core_idle()
- Towards practical verification of the Linux kernel

The Linux Kernel

- In theory, the operating system should always run properly.

- In reality, a lot of bugs:
  - Known bug patterns (e.g., NULL dereference)
    - There are many tools to detect them (but there are still many such bugs!)
  - Semantic bugs (e.g., missing privilege checks)
    - These are hard to detect systematically.

Deductive Program Verification: Overview



How Verifiers Work: Hoare Triple

- The verifier decides if a given program is correct.
- A correct program Prog is defined using the Hoare triple:

$$\{ \text{Pre-condition} \}\ \text{Prog}\ \{ \text{Post-condition} \}$$

How Verifiers Work: Hoare Triple

- The verifier decides if a given program is correct.

- A correct program Prog is defined using the Hoare triple:

{ Pre-condition } Prog { Post-condition }

If this pre-condition holds.

How Verifiers Work: Hoare Triple

- The verifier decides if a given program is correct.

- A correct program Prog is defined using the Hoare triple:

{ Pre-condition } Prog { Post-condition }

Then, after the execution of Prog

How Verifiers Work: Hoare Triple

- The verifier decides if a given program is correct.

- A correct program Prog is defined using the Hoare triple:

{ Pre-condition } Prog { Post-condition }

This post-condition holds

ELISA Enabling Linux in Safety Applications

WORKSHOP

How Verifiers Work: Hoare Triple

- Hoare triple: Example

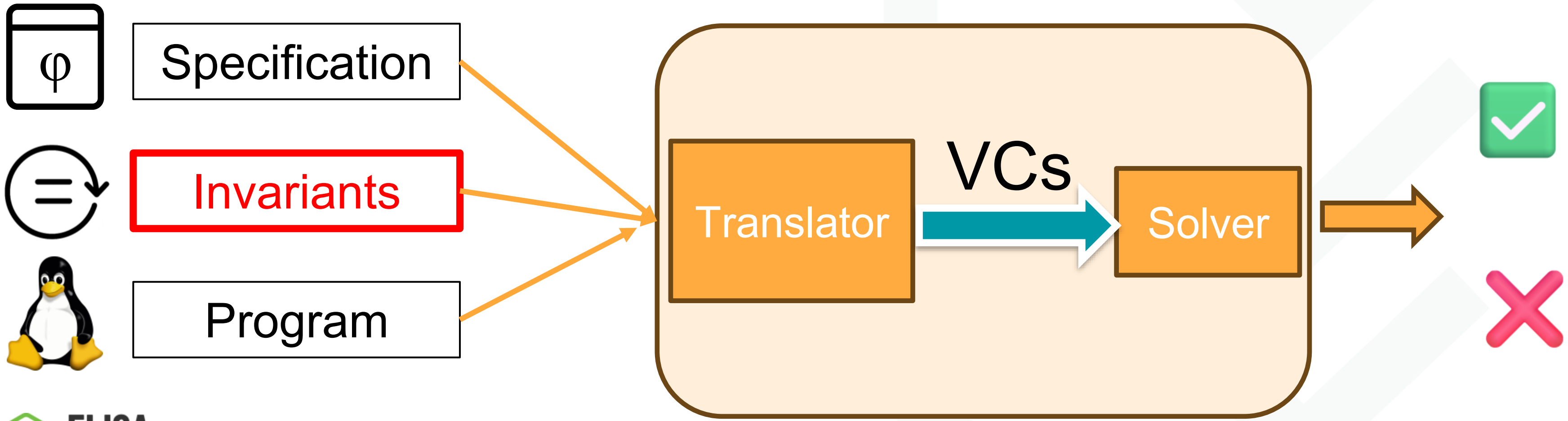
Pre-condition Post-condition

✅ `{ x > 1 } x := x+1 { x > 2 }`

✅ `{ x > 1 } x := x+1 { x > -10 }`

From Hoare Triple to Verification

- Verifiers translate a program and specifications to logical formulas:
  - If and only if the logical formulas hold (are satisfiable), the program is correct.
  - Such logical formulas are called verification conditions (VCs).



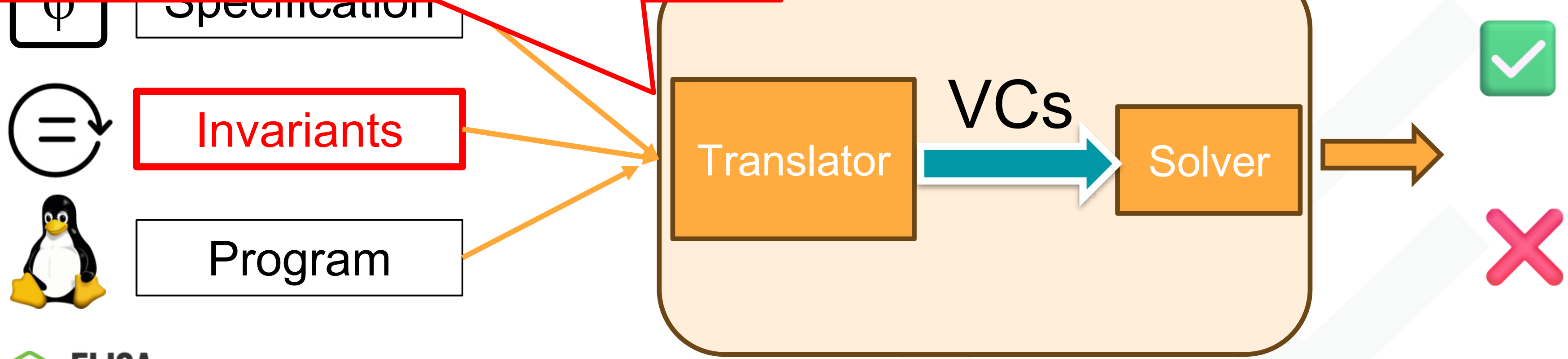
From Hoare Triple to Verification

Point 1: To generate VCs, the loops need to be annotated with invariants.

d specifications to logical formulas:

ld (are satisfiable), the program is correct.

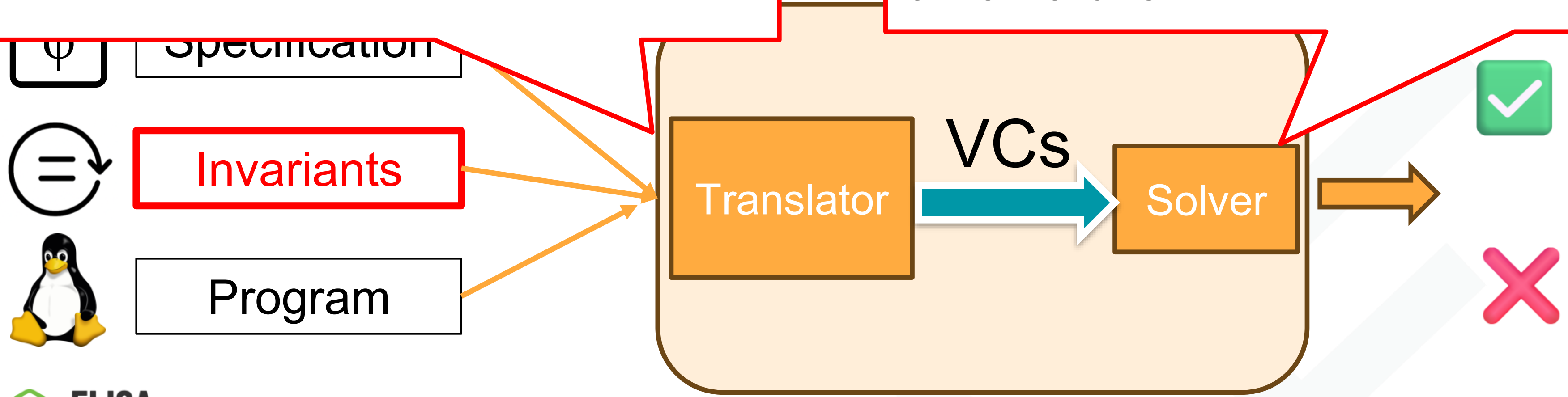
rification conditions (VCs).



From Hoare Triple to Verification

Point 1: To generate VCs, the loops need to be annotated with invariants.

Point 2: Some options to discharge VCs are available.



Generating Verification Conditions

```
{ x > 10 }

x = x + 1

{ x > 0 }
```

Idea: Finding the weakest preconditions by reasoning in the backward direction

The weakest preconditions: `x > -1`

VC: `x > -1` $\Rightarrow$ `x > 10`

Weakest Precondition   Given Precondition

Generating Verification Conditions

```
{ x > 10 }

  while(c)
    loop-body

{ x > 0 }
```

~~Idea: Finding the weakest preconditions by reasoning in the backward direction~~

How many iterations?

Does it terminate?

In general, automatically finding such conditions is impossible.

Manual Annotation for Loop Invariants

- With specifications, loop invariants also need to be specified.
- The loop invariants holds at both the beginning and the end of a loop

```
{ Precondition }
    while(c)
        { Invariant }
        loop-body
        { Invariant }
{ Postcondition }
```

One needs to find conditions that hold before and after the body.

VCs

$$\text{Precondition} \Rightarrow \text{Invariant}$$

$$\text{Invariant} \;\&\&\; !c \Rightarrow \text{Postcondition}$$

ELISA Enabling Linux in Safety Applications WORKSHOP

Manual Annotation for Loop Invariants

- With specifications, loop invariants also need to be specified.
- The loop invariants holds at both the beginning and the end of a loop

```
i = 0;
while (i<=10) {
      a[i] = 0;
      i++;
}
```

```
∀j, 0 <= j < i ⇒ a[j] == 0
&&
0 <= i <= 10
```

```
∀j, 0 <= j < i ⇒ a[j] == 0
&&
0 <= i <= 10
```

Manual Annotation for Loop Invariants

- Writing invariants for every loops in a function to be verified can be a major stumbling block in verification.

- For practical verification of the Linux kernel, some degree of automation is needed and doable; we will see it later.

Handling Verification Conditions

- How to discharge verification conditions?

Example

$$\forall j,\ 0 <= j < i \Rightarrow a[j] == 0$$
$$\&\&$$
$$0 <= i <= 10$$

- Options:
  - Pen & paper proof
  - Manually writing proof in interactive theorem provers.
  - Automatically discharging VCs using SMT solvers.

Handling Verification Conditions

- How to discharge verification conditions?

Example

```
∀j, 0 <= j < i ⇒ a[j] == 0
            &&
      0 <= i <= 10
```

😊 Flexibility
😞 Needs expertise and effort.

- Options:
  - ~~Pen & paper proof~~
  - Manually writing proof in interactive theorem provers.
  - Automatically discharging VCs using SMT solvers.

Handling Verification Conditions

- How to discharge verification conditions?

Example

```
∀j, 0 <= j < i ⇒ a[j] == 0
            &&
     0 <= i <= 10
```

- Options:
    - ~~Pen & paper proof~~
    - Manually writing proof in interactive theorem provers.
    - Automatically discharging VCs using SMT solvers.

😊 Automation
😞 No guarantee of success

ELISA Enabling Linux in Safety Applications WORKSHOP

Deductive Program Verification: Verifiers

- There are multiple available verifiers for C.
  - Frama-C
  - Verifast
  - CN
  - RefinedC
  - VerifiableC
  - etc.

Design Spaces:

- How to verify Hoare triples (or a similar approach)?
- Utilizing SMT solvers or interactive theorem provers?
- What kind of specifications?
- Maturity?
- …

Deductive Program Verification: Verifiers

- There are multiple available verifiers for C.
  - Frama-C
  - Verifast
  - CN
  - RefinedC
  - VerifiableC
  - etc.

Our research group uses Frama-C:

- Relatively limited support for pointers
- Using SMT solvers
- Expressive specifications

)?
ers?

- What kind of specifications?
- Maturity?
- …

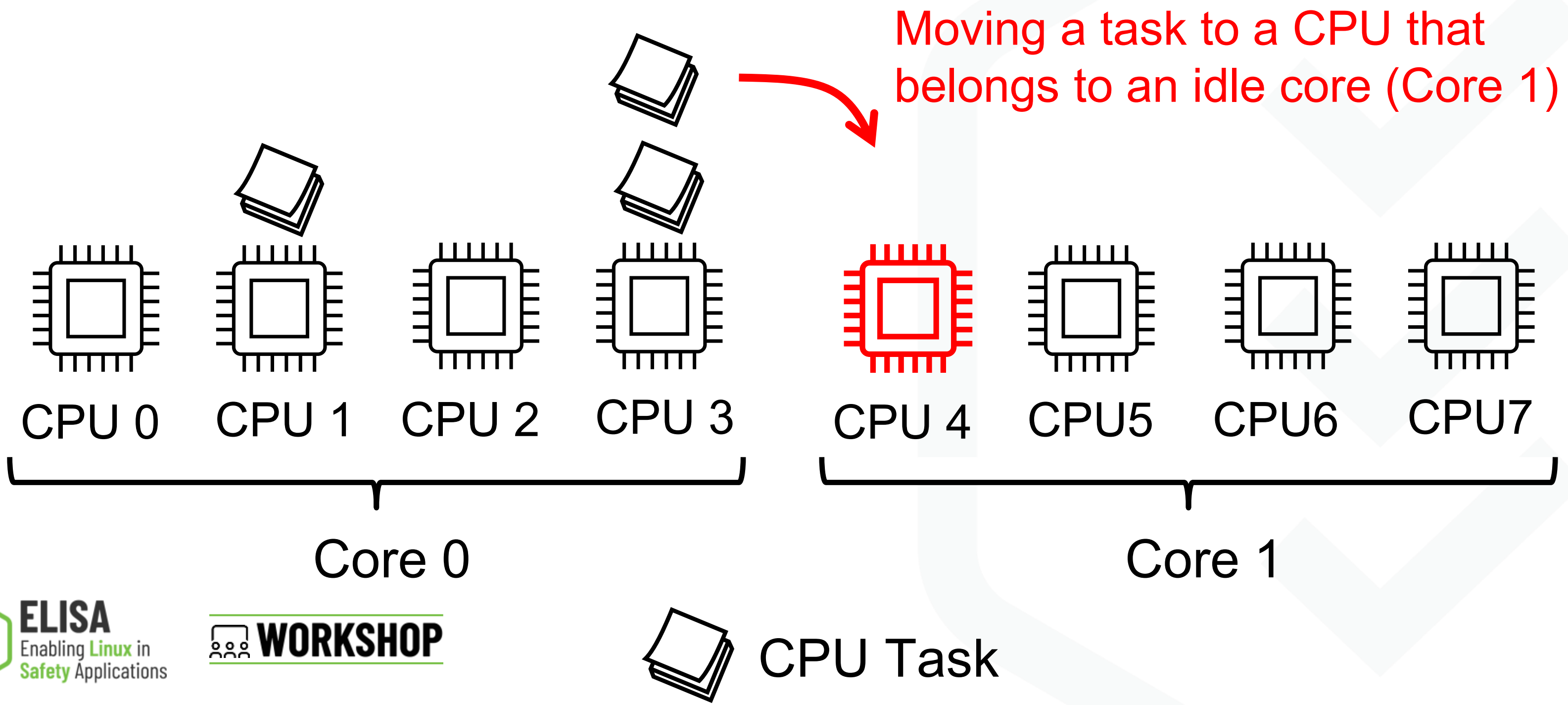
ELISA Enabling Linux in Safety Applications WORKSHOP

Outline

- Overview: What is verification?
- Case Study: Applying verification to a small function, is_core_idle()
- Towards practical verification of the Linux kernel

Background: Task Scheduling in the Kernel

- The task scheduler load-balances tasks among CPUs and cores.



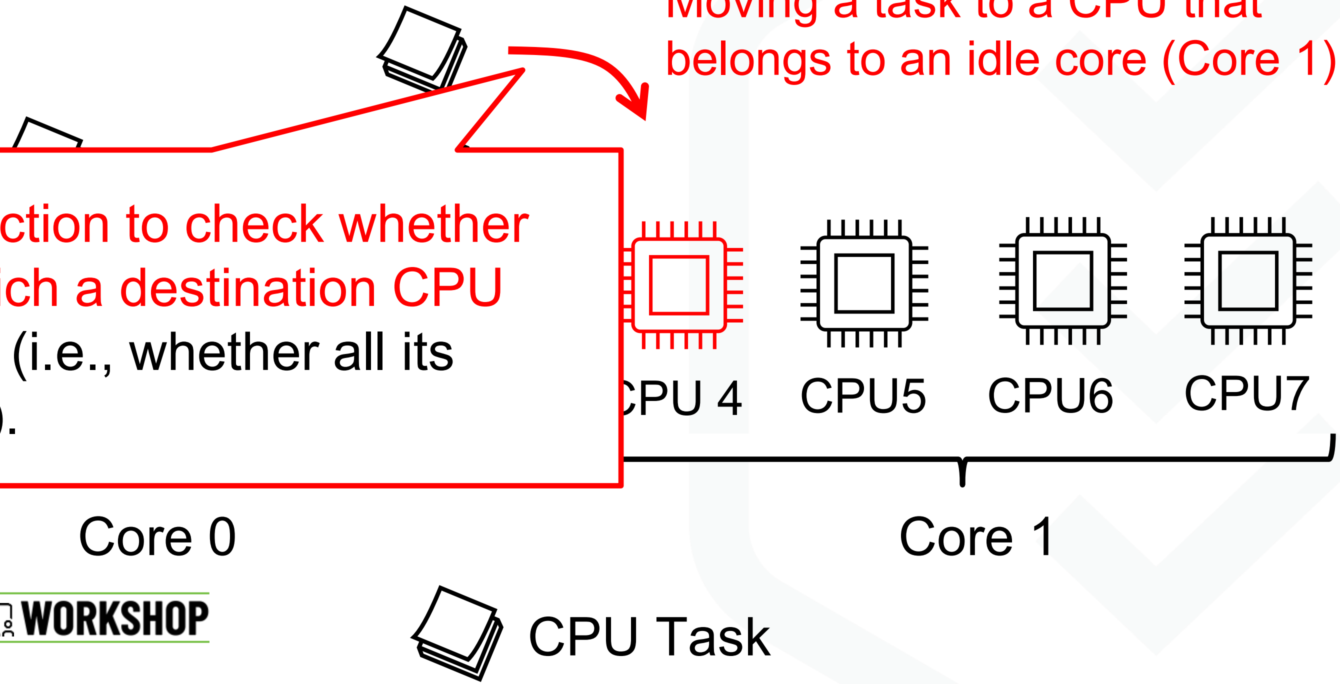
ELISA Enabling Linux in Safety Applications WORKSHOP

Background: Task Scheduling in the Kernel

- The task scheduler load-balances tasks among CPUs and cores.

Moving a task to a CPU that belongs to an idle core (Core 1)

We want a function to check whether the core to which a destination CPU belongs is idle (i.e., whether all its CPUs are idle).



Example: Verification of is_core_idle()

- We verify the function is_core_idle()
  - A function defined in the task scheduler (kernel/sched/fair.c)
  - A predicate that determines whether the core where the CPU belongs is idle.
    - Input: a CPU ID
    - Output: true iff the core is idle

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

Example: Verification of is_core_idle()

- is_core_idle() checks if all CPUs are idle.
  - This iterates over CPU IDs in a cpumask (a set of CPUs).
  - Once a non-idle CPU is found, it returns false immediately.
  - If all CPUs are idle, it returns true.

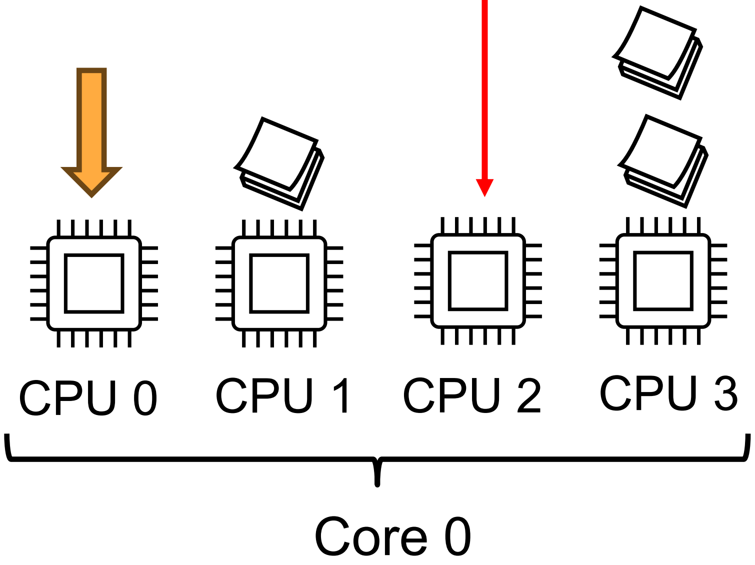
```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

ELISA Enabling Linux in Safety Applications

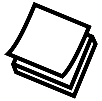
WORKSHOP

Example: Verification of is_core_idle()

cpu == 2

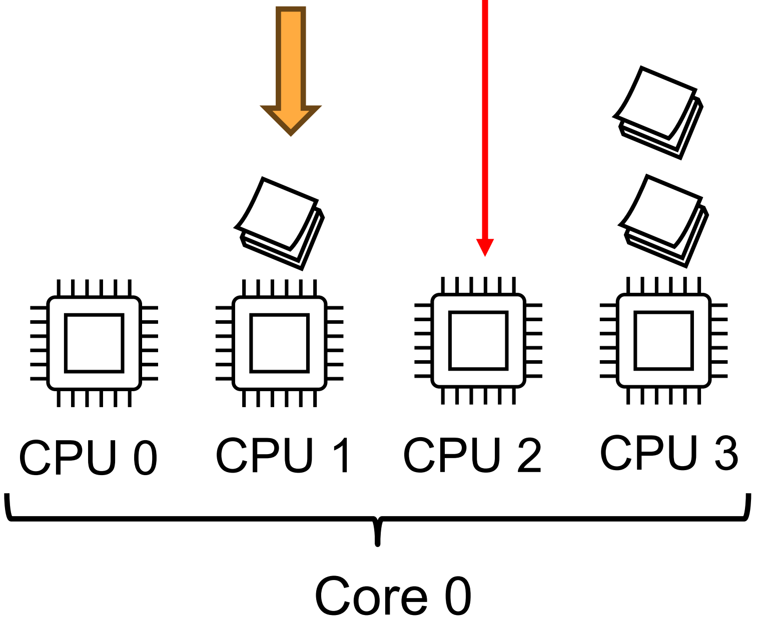



```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
        int sibling;

        for_each_cpu(sibling, cpu_smt_mask(cpu)) {
                if (cpu == sibling)
                        continue;

                if (!idle_cpu(sibling))
                        return false;
        }
#endif

        return true;
}
```


CPU Task

Example: Verification of is_core_idle()

cpu == 2

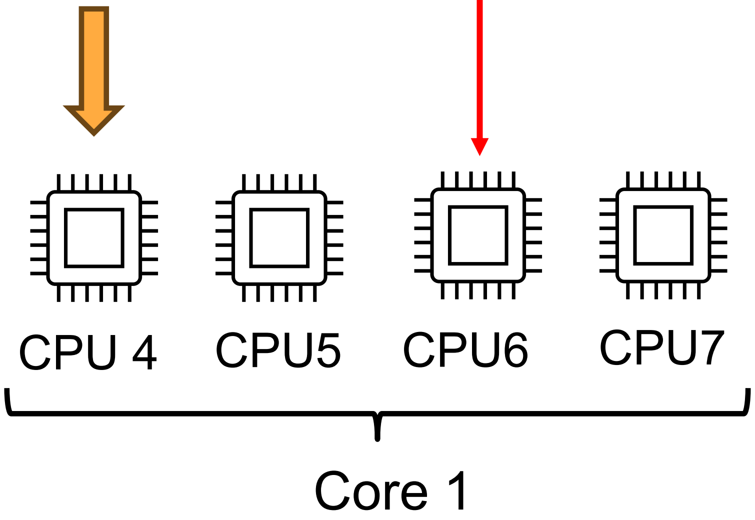



```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

CPU Task

Example: Verification of is_core_idle()

cpu == 6

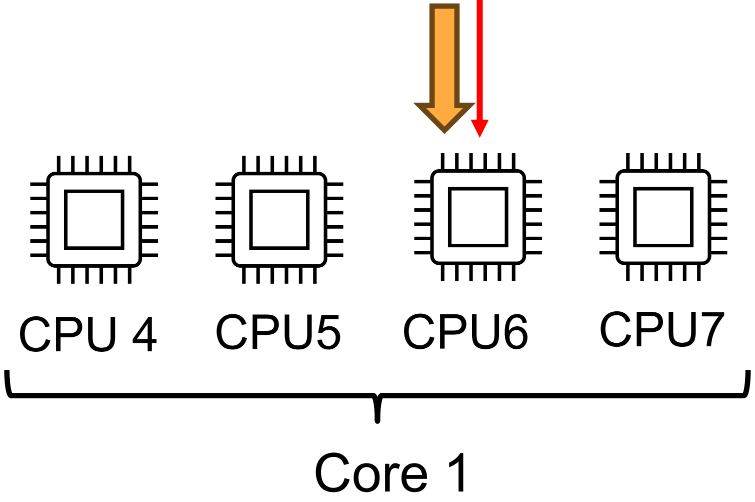



```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

CPU Task

Example: Verification of is_core_idle()

cpu == 6

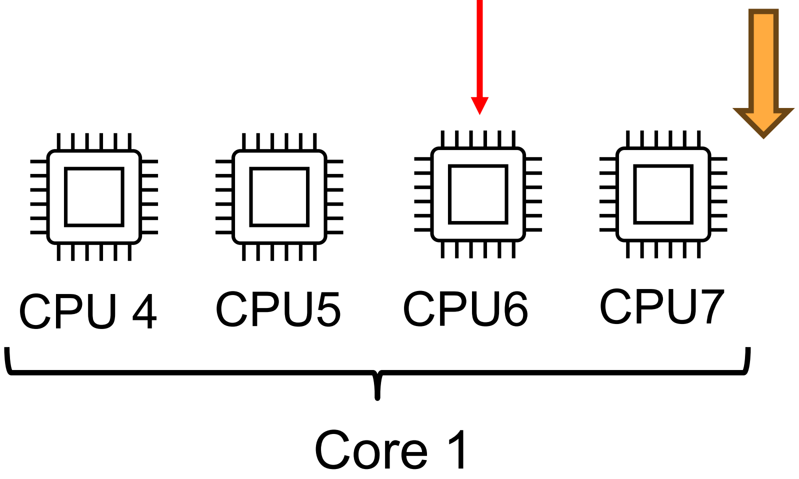



```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
        int sibling;

        for_each_cpu(sibling, cpu_smt_mask(cpu)) {
                if (cpu == sibling)
                        continue;

                if (!idle_cpu(sibling))
                        return false;
        }
#endif

        return true;
}
```

CPU Task

Example: Verification of is_core_idle()

cpu == 6




```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
        int sibling;

        for_each_cpu(sibling, cpu_smt_mask(cpu)) {
                if (cpu == sibling)
                        continue;

                if (!idle_cpu(sibling))
                        return false;
        }
#endif

        return true;
}
```

CPU Task

Example: Verification of is_core_idle()

How can we apply verification in practice?

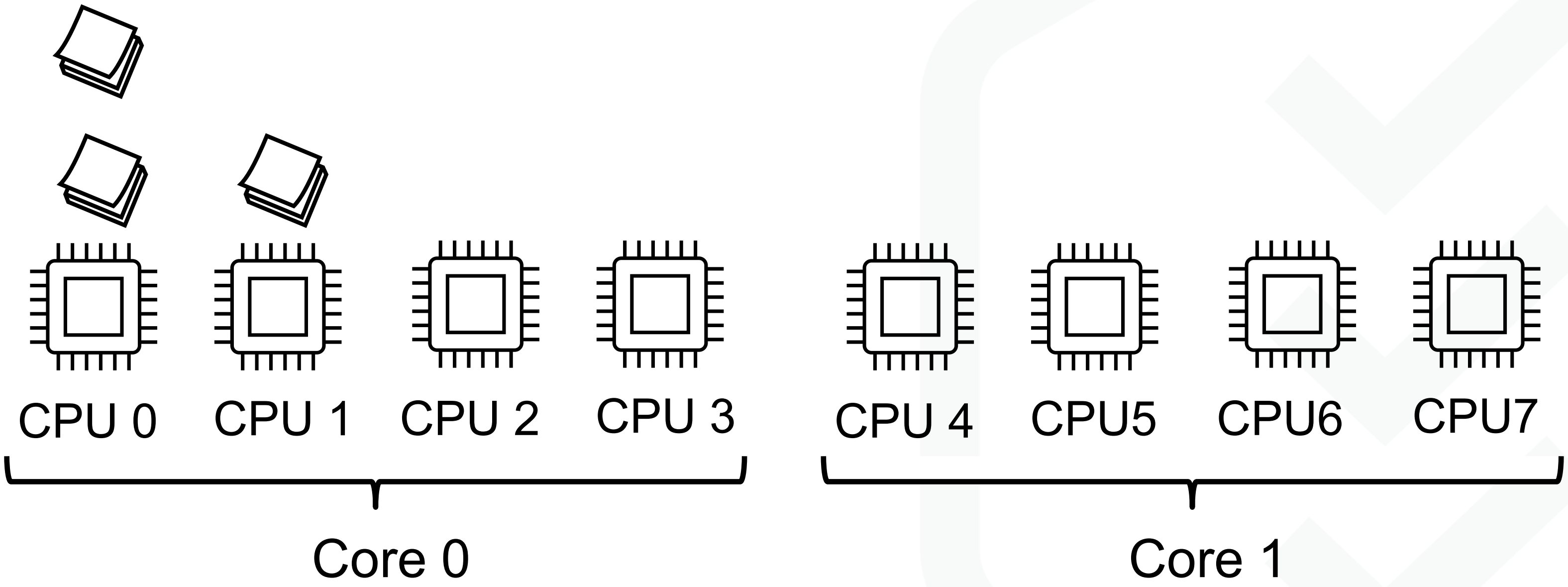
```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

ELISA Enabling Linux in Safety Applications

WORKSHOP

Define Specifications 1/3

The first step: model CPU states

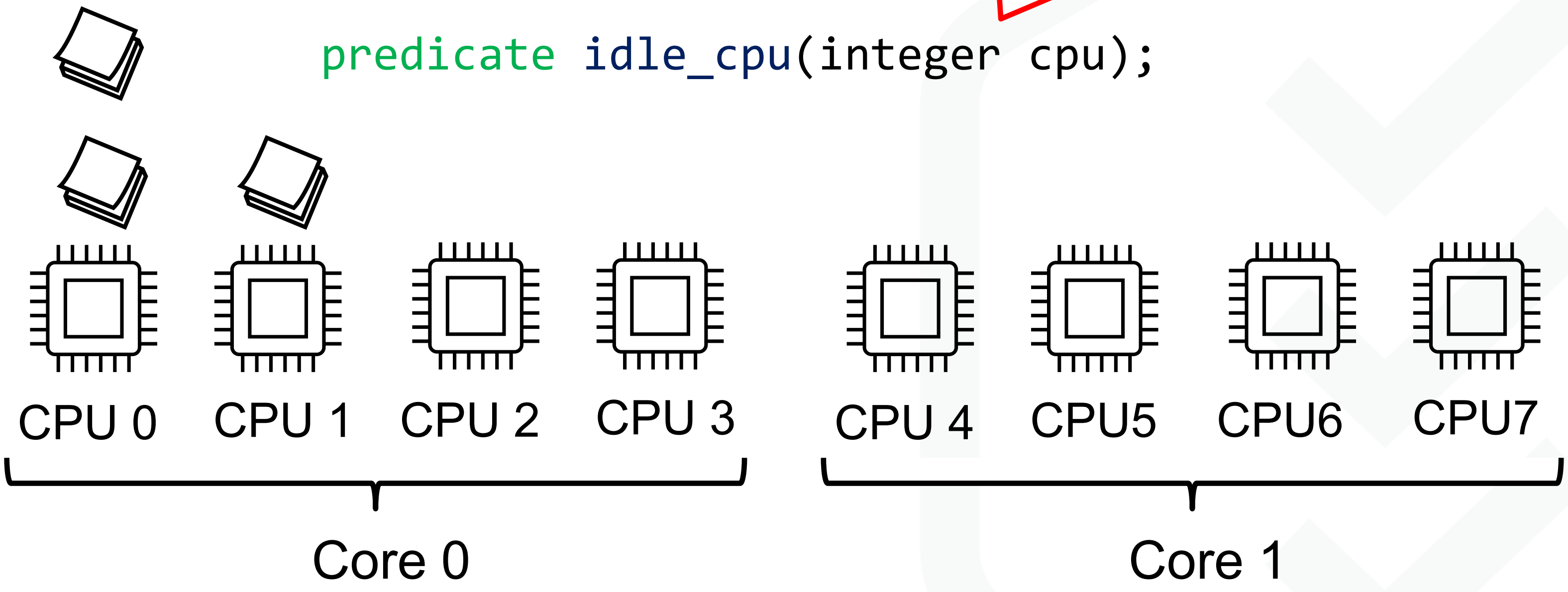


Define Specifications 1/3

Deciding if a CPU is idle.

The first step: model CPU states

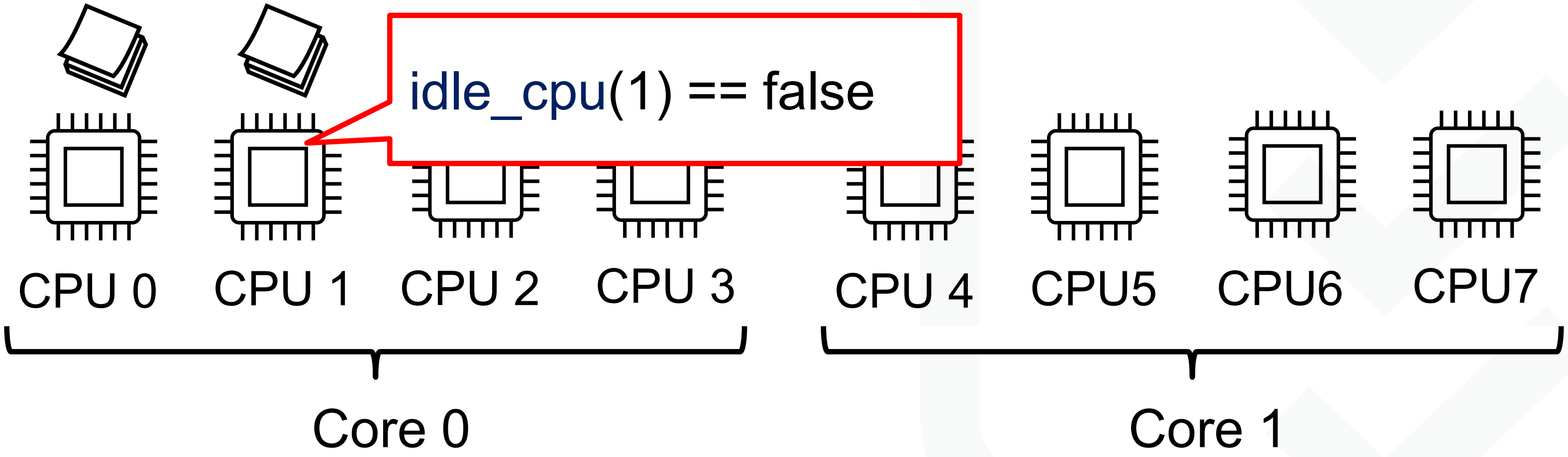
```
predicate idle_cpu(integer cpu);
```



Define Specifications 1/3

The first step: model CPU states

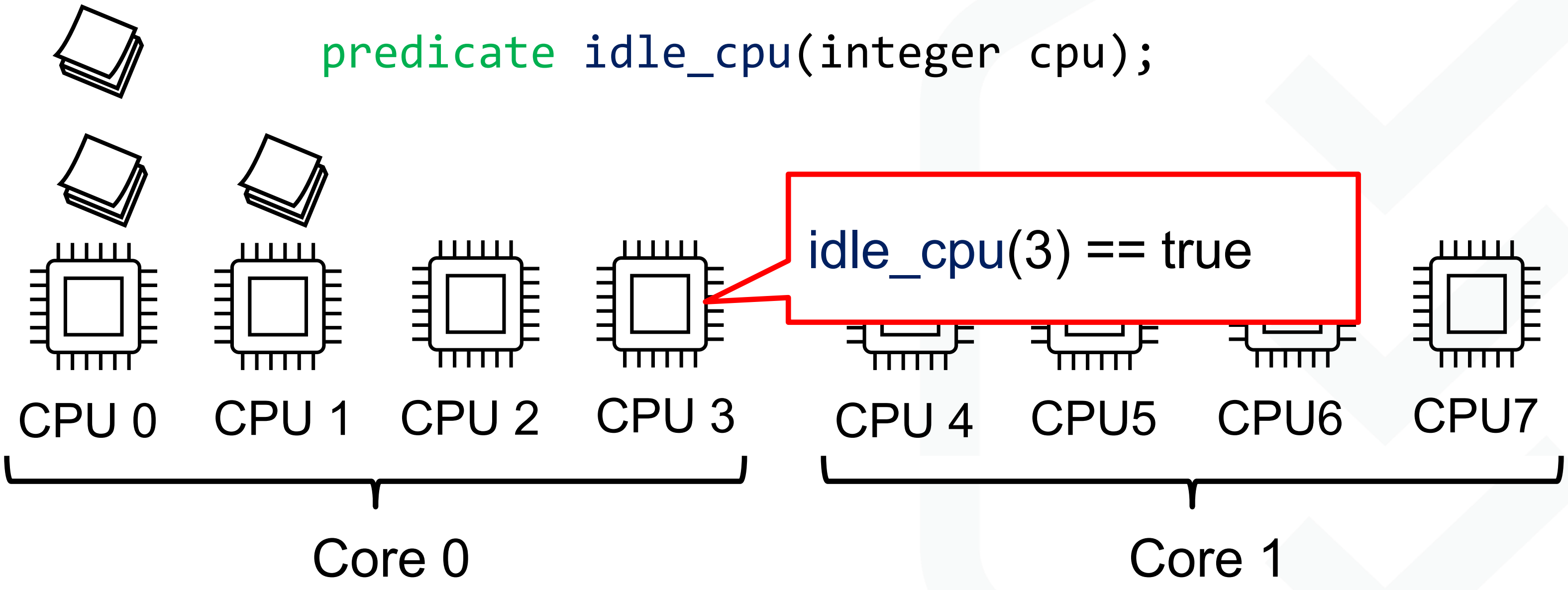
```
predicate idle_cpu(integer cpu);
```



Define Specifications 1/3

The first step: model CPU states

```
predicate idle_cpu(integer cpu);
```



Define Specifications 1/3

Mapping CPU IDs and Cores

The first step: model CPU states

```
logic integer get_core(integer cpu);
```

A function for specifications that returns an integer (Core ID).

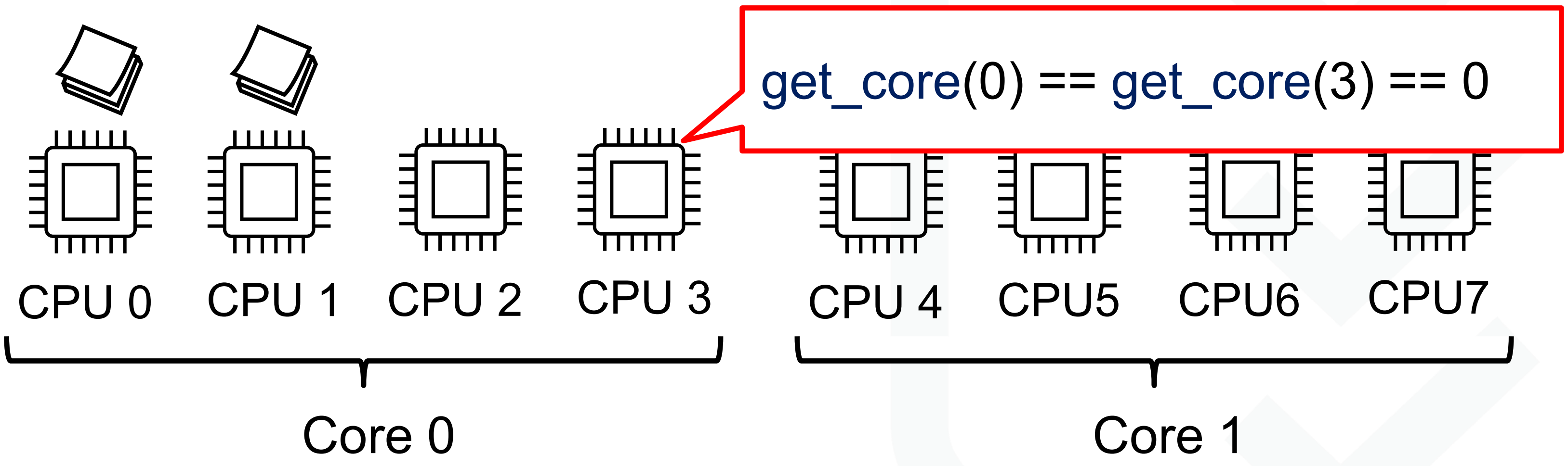
CPU 0 CPU 1 CPU 2 CPU 3 CPU 4 CPU5 CPU6 CPU7

Core 0 Core 1

Define Specifications 1/3

The first step: model CPU states

```
logic integer get_core(integer cpu);
```

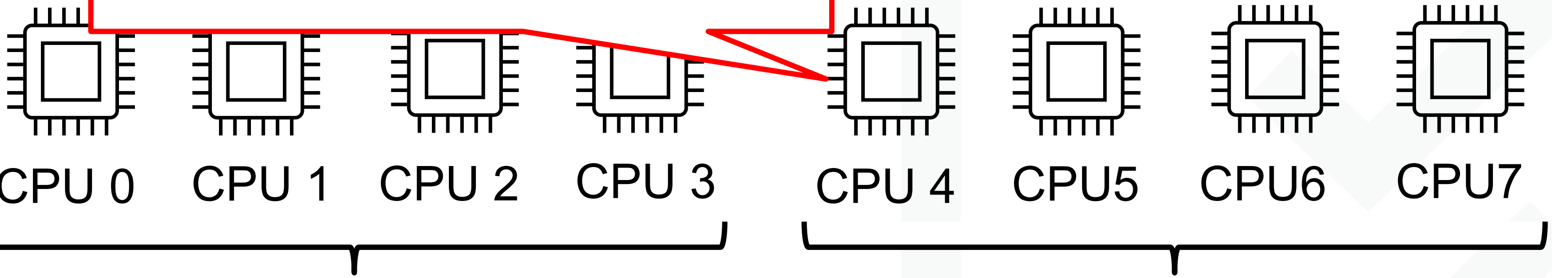


Define Specifications 1/3

The first step: model CPU states

```
logic integer get_core(integer cpu);
```

get_core(0) != get_core(4)



Define Specifications 1/3

The first step: model CPU states

```
axiomatic schedule_cpumask {
  predicate idle_cpu(integer cpu);
  logic integer get_core(integer cpu);

  predicate idle_core(integer cpu) =
    \forall integer i; 0 <= i < NR_CPUS ==>
    get_core(i) == get_core(cpu) ==> idle_cpu(i);
}
```

```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

ELISA Enabling Linux in Safety Applications

WORKSHOP

```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;

	true;
```

Define Specifications 1/3

The first step: model CPU states

```
axiomatic sc
  predicate i
  logic integ

  predicate idle_core(integer cpu) =
    \forall integer i; 0 <= i < NR_CPUS ==>
    get_core(i) == get_core(cpu) ==> idle_cpu(i);
}
```

This predicate defines the idleness of a core that a CPU (cpu) belongs to.

Define Specifications 1/3

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

The first step: model CPU states

```
axiomatic schedule_cpumask {
  predicate idle_cpu(integer cpu);
  logic integer get_core(integer cpu);

  predicate idle_core(integer cpu) =
    \forall integer i; 0 <= i < NR_CPUS ==>
    get_core(i) == get_core(cpu) ==> idle_cpu(i);
}
```

If CPU i belongs to the same core...

Define Specifications 1/3

The first step: model CPU states

```
axiomatic schedule_cpumask {
  predicate idle_cpu(integer cpu);
  logic integer get_core(integer cpu);

  predicate idle_core(integer cpu) =
    \forall integer i; 0 <= i < NR_CPUS ==>
    get_core(i) == get_core(cpu) ==> idle_cpu(i);
}
```

```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

Such CPUs must be idle

ELISA Enabling Linux in Safety Applications WORKSHOP

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPUS;
assigns \nothing;
ensures \result <==> idle_cpu(cpu);
*/
bool idle_cpu(int cpu);
```

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPUS;
assigns \nothing;
ensures \result <==> idle_cpu(cpu);
*/
bool idle_cpu(int cpu);
```

“requires” represents the preconditions

“ensures” represents the postconditions

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu <          s;
assigns \nothing;
ensures \result <==> idle_cpu(cpu);
*/
bool idle_cpu(int cpu);
```

means no side effects.

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPUS;
assigns \nothing;
ensures \result == cpu_smt_mask(cpu);
ensures cpu_smt_mask(cpu)->bits[cpu];
ensures \forall integer i; 0 <= i < NR_CPUS ==>
  (cpu_smt_mask(cpu)->bits[i] <==> get_core(i) == get_core(cpu));
*/
static inline struct cpumask *cpu_smt_mask(int cpu);
```

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPU
assigns \nothing;
ensures \result == cpu_sm
ensures cpu_smt_mask(cpu)
ensures \forall integer i
  (cpu_smt_mask(cpu)->bits[i] <=              e(i) == get_core(cpu));
*/
static inline struct cpumask *cpu_smt_mask(int cpu);
```

This function returns a cpumask that represents the CPU set contained in a core, where the CPU *cpu* belongs.

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPUS;
assigns \nothing;
ensures \result == cpu_smt_mask(cpu);
ensures cpu_smt_mask(cpu)->bits[cpu];
ensures \forall integer i; 0 <= i < NR_CPUS ==>
  (cpu_smt_mask(cpu)->bits[i] <==> get_core(i) == get_core(cpu));
*/
static inline struct cpumask *cpu_smt_mask(int cpu);
```

The CPU *cpu* is included in the cpumask.

Define Specifications 2/3

The second step: model the dependencies (the callee functions)

```
/*@
requires 0 <= cpu < NR_CPUS;
assigns \nothing;
ensures \result == cpu_smt_mask(cpu);
ensures cpu_smt_mask(cpu)->bits[cpu];
ensures \forall integer i; 0 <= i < NR_CPUS ==>
  (cpu_smt_mask(cpu)->bits[i] <==> get_core(i) == get_core(cpu));
*/
static inline struct cpumask *cpu_smt_mask(int cpu);
```

The other CPUs sharing the same core are also included.

Define Specifications 3/3

The third step: model the function behavior.

```
requires 0 <= cpu < NR_CPUS;
requires idle_cpu(cpu);

ensures \result <==> idle_core(cpu);
```

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

ELISA Enabling Linux in Safety Applications

WORKSHOP

Define Specifications 3/3

The third step: model the function behavior.

```
requires 0 <= cpu < NR_CPUS;
requires idle_cpu(cpu);

ensures \result <==> idle_core(cpu);
```

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

Applying the Verifier

```
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

Recall: a loop requires its invariants

- Iterates over the CPU IDs in the given core.
- Skips the idleness check for the given CPU that is supposed to be idle.
- Otherwise, it checks the idleness. If it is not idle, return early.

Applying the Verifier

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

```
loop invariant 0 <= sibling <= NR_CPUS;
loop invariant \forall integer j; 0 <= j < sibling ==>
 cpu_smt_mask(cpu)->bits[j] ==> idle_cpu(j);
```

ELISA Enabling Linux in Safety Applications WORKSHOP

Applying the Verifier

```c
static inline bool is_core_idle(int cpu)
{
#ifdef CONFIG_SCHED_SMT
	int sibling;

	for_each_cpu(sibling, cpu_smt_mask(cpu)) {
		if (cpu == sibling)
			continue;

		if (!idle_cpu(sibling))
			return false;
	}
#endif

	return true;
}
```

```
loop invariant 0 <= sibling <= NR_CPUS;
loop invariant \forall integer j; 0 <= j < sibling ==>
 cpu_smt_mask(cpu)->bits[j] ==> idle_cpu(j);
```

Some degree of automation is possible?

Outline

- Overview: What is verification?
- Case Study: Applying verification to a small function, is_core_idle()
- Towards practical verification of the Linux kernel

Inferring Loop Invariants for the Kernel

- Adapting to the Linux kernel may allow for a practical solution to infer invariants.
- Insight: there are idiomatic usage patterns in the kernel.
    - In the case of *for_each_cpu()*, there are more than 400 loop instances that perform similar operations to a certain extent.
    - Our current work: automatic invariant inference for specific contexts in the kernel.

```
for_each_cpu(sibling, cpu_smt_mask(cpu)) {
        if (cpu == sibling)
                continue;

        if (!idle_cpu(sibling))
                return false;
}
```

ELISA Enabling Linux in Safety Applications

WORKSHOP

Ideal Verification Workflow for the Kernel

1. Select a function of interest
2. Write a specification
3. Write loop invariants
4. Verify your code

Ideal Verification Workflow for the Kernel

1. Select a function of interest
2. Write a specification

   This is not a simple task…
3. Write loop invariants
4. Verify your code

Preparation of Verifiable Code

- What should an input file of a verifier be?
- One can use the source file by adding spec. directly, but:
  - The file contains a lot of irrelevant code.

```
$frama-c -wp kernel/sched/fair.i
[kernel] Parsing kernel/sched/fair.i (no preprocessing)
[kernel] include/uapi/linux/types.h:17: User Error:
  __int128 is currently unsupported by Frama-C.
[kernel] Frama-C aborted: invalid user input.
```

  - There are many dependencies located in other files.
  - Some dependencies irrelevant to the specifications hinder verification (e.g., pr_info() has side effects and may be semantically irrelevant).

Preparation of Verifiable Code

- Our solution: Automatic preparation of verification code
    - Extracting target functions and dependencies into a minimal but self-contained (compilable) file.
    - Replacing dependencies (types, macros, etc.) with ones suitable for verification based on user-defined instructions.

Preparation of Verifiable Code

- Our solution: Automatic preparation of verification code

- Example: is_core_idle() defined in kernel/sched/fair.c
  - Code Size:
    - fair.c (original code): 13742 lines of code
    - fair.i (macro extended code): 103053 lines of code
    - Our extracted code: 66 lines of code

Preparation of Verifiable Code

- Our solution: Automatic preparation of verification code
- Example
  - Code S
    - fai
    - fai.i (macro e ... lines of code
    - Our extracted code: 66 lines of code

- 9 macros, 4 callees, etc.
- The dependencies are declared/defined across 10 files

Manual preparation is time-consuming and error-prone.

ELISA Enabling Linux in Safety Applications WORKSHOP

Other Challenges Towards Practical Usages

- Lack of formalized specifications
  - We do not know the specifications of most functions in the kernel.
  - How strict should specifications be…?

- Limitations of automation (SMT solvers)
  - Even if a program is correct, there is no guarantee that the verification succeeds, especially due to quantifiers (forall and exists).
  - One can guide SMT solvers with hints, but doing so requires expertise.

Summary

- Deductive Program Verification
  - Pre/Post conditions-based specifications
  - Requirement of manual annotation of loop invariants
  - SMT solvers-based automation

- Towards practical applications:
  - Automating the preparation of verification code and inference of loop invariants **(our current work)**
  - Good specifications and better automation

Licensing of Workshop Results

All work created during the workshop is licensed under Creative Commons Attribution 4.0 International (CC-BY-4.0) [https://creativecommons.org/licenses/by/4.0/] by default, or under another suitable open-source license, e.g., GPL-2.0 for kernel code contributions.

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.