Program verification for the Linux kernel: Potential costs and benefits

Julia Lawall, Keisuke Nishimura, Jean-Pierre Lozi
Inria

September 18, 2024

The Linux kernel

- Seems reliable…

The Linux kernel

- Seems reliable... But actually it is full of bugs.

The Linux kernel

- Seems reliable... But actually it is full of bugs.
- Some fit well-known patterns:
  - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
  - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)

The Linux kernel

- Seems reliable... But actually it is full of bugs.
- Some fit well-known patterns:
  - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
  - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)
- Some depend on an algorithm, and are completely context specific:

The Linux kernel

- Seems reliable... But actually it is full of bugs.
- Some fit well-known patterns:
  - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
  - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)
- Some depend on an algorithm, and are completely context specific:
  - Maybe verification can help identify these bugs?

Formal verification

Basic idea:

- Write specifications describing expected code behavior.
- Use tools to verify that the code respects the specifications.

Formal verification

Basic idea:

- Write specifications describing expected code behavior.
- Use tools to verify that the code respects the specifications.

Specifications are a form of documentation, with tool support.

Costs? Benefits?

Positive:

- Thinking about what to prove can highlight inconsistencies, bugs, and missed optimization opportunities.
- Specifications provide an unambiguous, consistent description of what the code does.

Costs? Benefits?

Positive:

- Thinking about what to prove can highlight inconsistencies, bugs, and missed optimization opportunities.
- Specifications provide an unambiguous, consistent description of what the code does.

Negative:

- Creating specifications is hard.
- Can we hope to maintain them? (cognitive overload)
- No magic bullet: The specifications could even be wrong!

This talk

A thought experiment

This talk

A thought experiment

- Not, hmmph, I don’t want to write a bunch a formulas.

This talk

A thought experiment

- Not, hmmph, I don't want to write a bunch a formulas.
- But rather, what would happen if we did?

A simple example (thanks to Krister Walfridsson)

```c
static void swap(int *p, int *q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
}
```

A simple example (thanks to Krister Walfridsson)

```
static void swap(int *p, int *q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
}
```

Properties to verify:

- p and q are readable and writeable.
- The final p value is the original q value.
- The final q value is the original p value.

The tool we use: Frama-C

Approach:

- Annotate source code with pre conditions and post conditions.
  - Pre conditions describe the states in which the function can be called.
  - Post conditions describe the state after calling the function in those states.

The tool we use: Frama-C

Approach:

- Annotate source code with pre conditions and post conditions.
  - Pre conditions describe the states in which the function can be called.
  - Post conditions describe the state after calling the function in those states.
- Frama-C analyzes the code, line by line, and determines the conditions needed to establish the post conditions based on the preconditions.

The tool we use: Frama-C

Approach:

- Annotate source code with pre conditions and post conditions.
  - Pre conditions describe the states in which the function can be called.
  - Post conditions describe the state after calling the function in those states.
- Frama-C analyzes the code, line by line, and determines the conditions needed to establish the post conditions based on the preconditions.
- A *SMT solver* automatically proves that the conditions are satisfied.

Our pre and post conditions, in Frama-C notation

Preconditions:

- p and q are readable and writeable.

```
requires \valid(p);
requires \valid(q);
```

Our pre and post conditions, in Frama-C notation

Preconditions:

- p and q are readable and writeable.

```
requires \valid(p);
requires \valid(q);
```

Postconditions:

- The final p value is the original q value.

```
ensures *p == \old(*q);
```

Our pre and post conditions, in Frama-C notation

Preconditions:

- p and q are readable and writeable.

```
requires \valid(p);
requires \valid(q);
```

Postconditions:

- The final p value is the original q value.

```
ensures *p == \old(*q);
```

- The final q value is the original p value.

```
ensures *q == \old(*p);
```

Putting it all together

```
/*@
requires \valid(p);
requires \valid(q);
assigns *p, *q;
ensures *p == \old(*q);
ensures *q == \old(*p);
*/
static void swap(int *p, int *q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
}
```

Checking it with Frama-C

```
> frama-c -wp -wp-rte -wp-prover=z3 swap.c
[kernel] Parsing swap.c (with preprocessing)
[rte:annot] annotating function swap
[wp] 8 goals scheduled
[wp] Proved goals:   10 / 10
  Terminating:      1
  Unreachable:      1
  Qed:              5
  Z3 4.12.2:        3 (20ms-40ms)
```

Attacking the Linux kernel task scheduler

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.

Attacking the Linux kernel task scheduler

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.

Focus on the algorithm:

- No consideration of concurrency.
- No consideration of hidden memory issues
  (aliasing, null pointers, use after free, etc.).

Attacking the Linux kernel task scheduler

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.

Focus on the algorithm:

- No consideration of concurrency.
- No consideration of hidden memory issues (aliasing, null pointers, use after free, etc.).
- These are hard issues, but developers can make mistakes without them.

A case study: should_we_balance

Goal:

- Should a core should try to steal tasks during load balancing?

Starting point:

- Patch first proposed in August 2013.
- Extracted from scattered existing code.
- First patch was buggy.
- First released in Linux v3.12.

Subsequent history:

- 10 variants over time (+1 proposed as a result of this work).
- Several recent optimizations.

Overview of the should_we_balance code

Input:

- The CPU trying to steal.
- Some information about the set of CPUs participating in load balancing.

Overview of the should_we_balance code

Input:

- The CPU trying to steal.
- Some information about the set of CPUs participating in load balancing.

Action:

1. Elect an idle CPU that is allowed to steal.
2. If none, elect a default CPU.
3. Return true if and only if the elected CPU is the one trying to steal.

Overview of the should_we_balance code

Input:

- The CPU trying to steal.
- Some information about the set of CPUs participating in load balancing.

Action:

1. Elect an idle CPU that is allowed to steal.
2. If none, elect a default CPU.
3. Return true if and only if the elected CPU is the one trying to steal.

Goal: Only one non newly idle CPU steals at a time.

The original definition

```
static int should_we_balance(struct lb_env *env) {
        struct sched_group *sg = env->sd->groups;
        struct cpumask *sg_cpus, *sg_mask;
        int cpu, balance_cpu = -1;

        if (env->idle == CPU_NEWLY_IDLE)
                return 1;
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;
                balance_cpu = cpu;
                break;
        }
        if (balance_cpu == -1)
                balance_cpu = group_balance_cpu(sg);
        return balance_cpu != env->dst_cpu; // != should be ==
}
```

Input: `env` describes the core that wants to steal tasks

```
static int should_we_balance(struct lb_env *env) {
        struct sched_group *sg = env->sd->groups;
        struct cpumask *sg_cpus, *sg_mask;
        int cpu, balance_cpu = -1;

        if (env->idle == CPU_NEWLY_IDLE)
                return 1;
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;
                balance_cpu = cpu;
                break;
        }
        if (balance_cpu == -1)
                balance_cpu = group_balance_cpu(sg);
        return balance_cpu != env->dst_cpu; // != should be ==
}
```

If the core is newly idle, it can always steal

```
static int should_we_balance(struct lb_env *env) {
        struct sched_group *sg = env->sd->groups;
        struct cpumask *sg_cpus, *sg_mask;
        int cpu, balance_cpu = -1;

        if (env->idle == CPU_NEWLY_IDLE)
                return 1;
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;
                balance_cpu = cpu;
                break;
        }
        if (balance_cpu == -1)
                balance_cpu = group_balance_cpu(sg);
        return balance_cpu != env->dst_cpu; // != should be ==
}
```

Otherwise, find the core that is allowed to steal

```
static int should_we_balance(struct lb_env *env) {
        struct sched_group *sg = env->sd->groups;
        struct cpumask *sg_cpus, *sg_mask;
        int cpu, balance_cpu = -1;

        if (env->idle == CPU_NEWLY_IDLE)
                return 1;
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;
                balance_cpu = cpu;
                break;
        }
        if (balance_cpu == -1)
                balance_cpu = group_balance_cpu(sg);
        return balance_cpu != env->dst_cpu; // != should be ==
}
```

Is the core that is allowed to steal the current one?

```c
static int should_we_balance(struct lb_env *env) {
        struct sched_group *sg = env->sd->groups;
        struct cpumask *sg_cpus, *sg_mask;
        int cpu, balance_cpu = -1;

        if (env->idle == CPU_NEWLY_IDLE)
                return 1;
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;
                balance_cpu = cpu;
                break;
        }
        if (balance_cpu == -1)
                balance_cpu = group_balance_cpu(sg);
        return balance_cpu != env->dst_cpu; // != should be ==
}
```

Initial version (verification expert?): pre and post conditions

```
/*@
... // data validity, no side effects

behavior newly_idle:
  assumes env->idle == CPU_NEWLY_IDLE;
  ensures \result;

behavior not_newly_idle1:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);
  ensures \forall integer i;
    relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (\result <==> env->dst_cpu != i);

behavior not_newly_idle2:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes \forall integer i; relevant(i, env) ==> !idle_cpu(i);
  ensures \result <==> group_balance_cpu(env->sd->groups) != env->dst_cpu;

complete behaviors;
disjoint behaviors;
*/
```

Initial version (verification expert?): loop invariants

```
static int should_we_balance(struct lb_env *env)
{
        ...
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        /*@
          loop invariant 0 <= cpu <= small_cpumask_bits;
          loop invariant \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
          loop assigns cpu;
          loop variant small_cpumask_bits - cpu;
        */
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;

                balance_cpu = cpu;
                break;
        }
        ...
}
```

Initial version (verification expert?): loop invariants

```
static int should_we_balance(struct lb_env *env)
{
        ...
        sg_cpus = sched_group_cpus(sg);
        sg_mask = sched_group_mask(sg);
        /*@
          loop invariant 0 <= cpu <= small_cpumask_bits;
          loop invariant \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
          loop assigns cpu;
          loop variant small_cpumask_bits - cpu;
        */
        for_each_cpu_and(cpu, sg_cpus, env->cpus) {
                if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
                        continue;

                balance_cpu = cpu;
                break;
        }
        ...
}
```

On our test machine, Frama-C proves this in under 1 minute.

Problem: The code evolves over time

#	Commit id	Date	Release	Impact
0	23f0d2093c78	Aug. 2013	–	create the function
1	b0cff9d88ce2	Sep. 2013	v3.12	replace != by ==
2	af218122b103	May 2017	–	eliminate a redundant function call
3	e5c14b1fb892	May 2017	v4.13	rename a function
4	024c9d2faebd	Oct. 2017	v4.14	check validity of the stealing CPU
5	97fb7a0a8944	Mar. 2018	v4.17	improve comments
6	64297f2b03cc	Apr. 2020	v5.8	return early on finding an idle core
7	792b9f65a568	Jun. 2022	v6.0	abort if tasks are detected on a newly idle CPU
8	b1bfeab9b002	Jul. 2023	–	prefer fully idle cores
9	f8858d96061f	Sep. 2023	v6.6	remove non-idle hyperthreads from the CPU mask
10	6d7e4782bcf5	Oct. 2023	v6.8	change a condition of the selection algorithm

Red versions contain bugs.

Question:

As the code changes,
can developers update the specifications accordingly?

An idea

- For optimizations, the overall input-output behavior should not change.

An idea

- For optimizations, the overall input-output behavior should not change.
- Maybe we could define pre and post conditions for one version and reuse them on new versions?

Change types and proof impact: No impact

Changes (mostly capitalization) in comments clearly have no impact on the proof.

Change types and proof impact: No impact

Changes (mostly capitalization) in comments clearly have no impact on the proof.

Code changes may also have no impact on the proof.

```
 static int should_we_balance(struct lb_env *env)
 {
        struct sched_group *sg = env->sd->groups;
-       int cpu, balance_cpu = -1;
+       int cpu;

        ...
        for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
-               balance_cpu = cpu;
-               break;
+               return cpu == env->dst_cpu;
        }
-       if (balance_cpu == -1)
-               balance_cpu = group_balance_cpu(sg);
-       return balance_cpu == env->dst_cpu;
+       return group_balance_cpu(sg) == env->dst_cpu;
 }
```

Change types and proof impact: new conditions

```
 static int should_we_balance(struct lb_env *env)
 {
         struct sched_group *sg = env->sd->groups;
         int cpu, balance_cpu = -1;

+        if (!cpumask_test_cpu(env->dst_cpu, env->cpus))
+                return 0;
         if (env->idle == CPU_NEWLY_IDLE)
                 return 1;
         for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
                 if (!idle_cpu(cpu))
                         continue;
                 balance_cpu = cpu;
                 break;
         }
         if (balance_cpu == -1)
                 balance_cpu = group_balance_cpu(sg);
         return balance_cpu == env->dst_cpu;
 }
```

Change types and proof impact: new conditions

```
+behavior race_condition:
+  assumes !env->cpus->bits[env->dst_cpu];
+  ensures !\result;
+
 behavior newly_idle:
   assumes env->idle == CPU_NEWLY_IDLE;
+  assumes env->cpus->bits[env->dst_cpu];
   ensures \result;

 behavior not_newly_idle1:
   assumes env->idle != CPU_NEWLY_IDLE;
+  assumes env->cpus->bits[env->dst_cpu];
   assumes \exists integer i; relevant(i, env) && idle_cpu(i);
   ensures \forall integer i;
     relevant(i, env) ==> idle_cpu(i) ==>
     (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
     (\result <==> env->dst_cpu == i);

 behavior not_newly_idle2:
   assumes env->idle != CPU_NEWLY_IDLE;
+  assumes env->cpus->bits[env->dst_cpu];
   assumes \forall integer i; relevant(i, env) ==> !idle_cpu(i);
   ensures \result <==> group_balance_cpu(env->sd->groups) == env->dst_cpu;
```

Change types and proof impact: more invasive changes

```
        for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
+               if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
+                       if (idle_smt == -1)
+                               idle_smt = cpu;
+                       continue;
+               }
                return cpu == env->dst_cpu;
        }
```

- Sensitive to hyperthreads.
- Avoid a core whose hyperthread is occupied, but keep it as a fallback.

Change types and proof impact: more invasive changes

```
        for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
+               if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
+                       if (idle_smt == -1)
+                               idle_smt = cpu;
+                       continue;
+               }
                return cpu == env->dst_cpu;
        }
```

Specification change:

```
  /*@
   loop invariant 0 <= cpu <= small_cpumask_bits;
-  loop invariant \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
-  loop assigns cpu;
+  loop invariant env->sd->flags & SD_SHARE_CPUCAPACITY ==> idle_smt == -1;
+  loop invariant idle_smt == -1 ==> \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
+  loop invariant idle_smt != -1 ==> 0 <= idle_smt < cpu && relevant(idle_smt, env) && idle_cpu(idle_smt);
+  loop invariant idle_smt != -1 ==> \forall integer j; 0 <= j < idle_smt ==> relevant(j, env) ==> !idle_cpu(j);
+  loop invariant idle_smt != -1 ==> \forall integer j; idle_smt <= j < cpu ==> relevant(j, env) ==> !idle_core(j);
+  loop assigns cpu, idle_smt;
   loop variant small_cpumask_bits - cpu;
  */
```

Change types and proof impact: invasive changes

```
+       cpumask_copy(swb_cpus, group_balance_mask(sg));
-       for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+       for_each_cpu_and(cpu, swb_cpus, env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
                if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
                        if (idle_smt == -1)
                                idle_smt = cpu;
+#ifdef CONFIG_SCHED_SMT
+                       cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
+#endif
                        continue;
                }
                return cpu == env->dst_cpu;
        }
```

Change types and proof impact: invasive changes

```
+       cpumask_copy(swb_cpus, group_balance_mask(sg));
-       for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+       for_each_cpu_and(cpu, swb_cpus, env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
                if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
                        if (idle_smt == -1)
                                idle_smt = cpu;
+#ifdef CONFIG_SCHED_SMT
+                       cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
+#endif
                        continue;
                }
                return cpu == env->dst_cpu;
        }
```

- cpumask_andnot writes into its first argument.
  - Such side effects impact the loop invariants.

Change types and proof impact: invasive changes

```
+       cpumask_copy(swb_cpus, group_balance_mask(sg));
-       for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+       for_each_cpu_and(cpu, swb_cpus, env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
                if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
                        if (idle_smt == -1)
                                idle_smt = cpu;
+#ifdef CONFIG_SCHED_SMT
+                       cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
+#endif
                        continue;
                }
                return cpu == env->dst_cpu;
        }
```

- cpumask_andnot writes into its first argument.
    - Such side effects impact the loop invariants.
- The first two arguments to cpumask_andnot are aliases.

Change types and proof impact: invasive changes

```
+       cpumask_copy(swb_cpus, group_balance_mask(sg));
-       for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+       for_each_cpu_and(cpu, swb_cpus, env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
                if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
                        if (idle_smt == -1)
                                idle_smt = cpu;
+#ifdef CONFIG_SCHED_SMT
+                       cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
+#endif
                        continue;
                }
                return cpu == env->dst_cpu;
        }
```

- cpumask_andnot writes into its first argument.
  - Such side effects impact the loop invariants.
- The first two arguments to cpumask_andnot are aliases.

Months of work...

Bug found

An older behavior:

```
behavior not_newly_idle1:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes env->cpus->bits[env->dst_cpu];
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (\result <==> env->dst_cpu == i);
```

A newer behavior: (bug introduced)

```
behavior not_newly_idle1b:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes env->cpus->bits[env->dst_cpu];
  assumes !(env->sd->flags & SD_SHARE_CPUCAPACITY);
  assumes \forall integer i; relevant(i, env) ==> !idle_core(i);
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (\result ==> (env->dst_cpu == i || env->dst_cpu == group_balance_cpu(env->sd->groups)));
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (env->dst_cpu == i ==> \result);
```

The buggy code

```c
static int should_we_balance(struct lb_env *env)
{
        struct cpumask *swb_cpus = this_cpu_cpumask_var_ptr(should_we_balance_tmpmask);
        struct sched_group *sg = env->sd->groups;
        int cpu, idle_smt = -1;

        [...]
        if (env->idle == CPU_NEWLY_IDLE) {
                [...]
                return 1;
        }
        cpumask_copy(swb_cpus, group_balance_mask(sg));
        for_each_cpu_and(cpu, swb_cpus, env->cpus) {
                if (!idle_cpu(cpu))
                        continue;
                if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
                        if (idle_smt == -1)
                                idle_smt = cpu;
                        cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
                        continue;
                }
                return cpu == env->dst_cpu;
        }
        if (idle_smt == env->dst_cpu)
                return true;
        return group_balance_cpu(sg) == env->dst_cpu;
}
```

Assessment

#	Commit id	Date	Release	Impact
0	23f0d2093c78	Aug. 2013	–	create the function
1	b0cff9d88ce2	Sep. 2013	v3.12	replace != by ==
2	af218122b103	May 2017	–	eliminate a redundant function call
3	e5c14b1fb892	May 2017	v4.13	rename a function
4	024c9d2faebd	Oct. 2017	v4.14	check validity of the stealing CPU
5	97fb7a0a8944	Mar. 2018	v4.17	improve comments
6	64297f2b03cc	Apr. 2020	v5.8	return early on finding an idle core
7	792b9f65a568	Jun. 2022	v6.0	abort if tasks are detected on a newly idle CPU
8	b1bfeab9b002	Jul. 2023	–	prefer fully idle cores
9	f8858d96061f	Sep. 2023	v6.6	remove non-idle hyperthreads from the CPU mask
10	6d7e4782bcf5	Oct. 2023	v6.8	change a condition of the selection algorithm

- Changes 1-7 easy to verify.

Assessment

#	Commit id	Date	Release	Impact
0	23f0d2093c78	Aug. 2013	–	create the function
1	b0cff9d88ce2	Sep. 2013	v3.12	replace != by ==
2	af218122b103	May 2017	–	eliminate a redundant function call
3	e5c14b1fb892	May 2017	v4.13	rename a function
4	024c9d2faebd	Oct. 2017	v4.14	check validity of the stealing CPU
5	97fb7a0a8944	Mar. 2018	v4.17	improve comments
6	64297f2b03cc	Apr. 2020	v5.8	return early on finding an idle core
7	792b9f65a568	Jun. 2022	v6.0	abort if tasks are detected on a newly idle CPU
8	b1bfeab9b002	Jul. 2023	–	prefer fully idle cores
9	f8858d96061f	Sep. 2023	v6.6	remove non-idle hyperthreads from the CPU mask
10	6d7e4782bcf5	Oct. 2023	v6.8	change a condition of the selection algorithm

- Changes 1-7 easy to verify.
- Changes 8 and 9 introduced challenges, but revealed a bug and a missed optimization opportunity

Going forward

Key observation so far:

- Complexities in the code are magnified in the specifications, exploding the proof time and effort.

Some tools that might help:

- Tools to isolate relevant code:
  - Collect dependencies.
- Tools to facilitate writing specifications:
  - Collect aliases.
  - Construct invariants for specific loop types.
- Tools to help react to code changes:
  - Distinguish easy and challenging code changes.
  - Identify and interpret source code bugs.

https://gitlab.inria.fr/lawall/swb_artifact