MultiSSE: Static Syscall Elision and Specialization for Event-Triggered Multi-Core RTOS

Gerion Entrup, Björn Fiedler, Daniel Lohmann

Leibniz Universität Hannover
{entrup, fiedler, lohmann}@sra.uni-hannover.de

***Abstract*—The implementation of static real-time control systems often allows for extensive compile-time optimizations based on RTOS-aware whole-program analyses. Previous work has shown the high optimization potential of control-flow aware static system-call tailoring, but is restricted to single-core systems due to the inherent problem of an exponentially growing analysis state in multicore settings.**

We present MultiSSE, a multi-core capable and RTOS-aware static whole-system analysis that makes such analyses also feasible on multi-core systems. MultiSSE exploits structural and optional timing information to analyze the core-level control flows as independently as possible from each other, synchronizing their states only when necessary. Thereby, MultiSSE provides means to realize compile-time deadlock detection, lock elision, and system-call optimization also on multi-core systems. We evaluate our approach with synthetic benchmarks and a real-world quadrotor application. In all cases, we were able to optimize or even completely elide costly cross-core system calls and system objects.

I. Introduction

Much real-time control systems (RTCSs) are implemented as special-purpose systems [26], [6]. This facilitates a closed-world assumption regarding their code artifacts and, thus, extensive static optimization of the underlying software utilizing whole-program analyses (WPAs) in the respective compilers and linkers. The real-time operating system (RTOS) itself is also tailored toward the specific application: All HW/SW events are mapped to a finite set of threads, interrupt service routines (ISRs), semaphores, or other system objects, which makes it possible to allocate them at compile time. This is realized, for instance, in OSEK/AUTOSAR [30], [2], which require ahead-of-time system-object definitions to keep the RAM overhead low, but also the much newer Zephyr RTOS [40] provides static initialization for the same purpose.

A. RTOS-Aware Static Analysis

WPA and the resulting optimization usually stop at system-call boundaries, as the semantics and threading model of the RTOS are typically unknown to the compiler. However, Dietrich and colleagues have shown [8], [9] that it is possible to realize RTOS-aware WPA that works *across* the RTOS boundaries and the RTOS-managed control flows (i.e. threads, alarms, ISRs). If, for instance, a higher-priority thread A triggers the activation of a lower-priority thread B, the RTOS-aware compiler could specialize the respective system call (i.e., `ActivateTask(B)` in AUTOSAR) to only mark B as ready but omit the costly scheduler invocation, as we statically know that no rescheduling is necessary at this point. Such system-call specialization can greatly improve memory footprint, latency and robustness of the resulting system [8], [9]. Technically, the possible specializations are inferred from a graph enumerating all RTOS states (running thread, ready threads, ISR status, ...) possible at the run time of this particular application. This graph is constructed by an algorithm named the system-state enumeration (SSE). However, the SSE addresses only single-core systems. While the authors state that their approach "would also work for multicore systems" [8] this holds only for the unrealistic setting of a completely partitioned system without *any* cross-core (cross-partition) interaction and dependency.

B. About this Paper

In this paper, we present the MultiSSE for partitioned fixed-priority multi-core systems with cross-core interactions, such as described by AUTOSAR 4.0 [2] (e.g., cross-core task activation or spinlocks). However, a complete state enumeration, as in the SSE, is generally unfeasible with multiple cores, as the number of possible system states rises exponentially with the number of cores. The MultiSSE addresses this challenge by exploiting the fact that cross-core interactions, even though possible, occur comparatively *rare* in partitioned systems. Thus, we can analyze each core independently until we get to a cross-core interaction. Only at *this* point, we need to synchronize the states by deriving explicit synchronization points that manifest the possible relative execution states of the involved cores. In particular, we claim the following contributions:

- We describe MultiSSE, an approach for the analysis and optimization of multi-core real-time systems at the application–RTOS interface.
- As part of MultiSSE, we provide a feasible algorithm for enumerating all possible RTOS states on a multi-core system together with an ordering of these states.
- To demonstrate the use of the algorithm, we sketch possible system call specializations that have the potential to reduce jitter and priority inversion in event-triggered multi-core real-time systems.

II. The MultiSSE Approach

The MultiSSE aims to provide detailed information about the application-RTOS interactions from its implementation

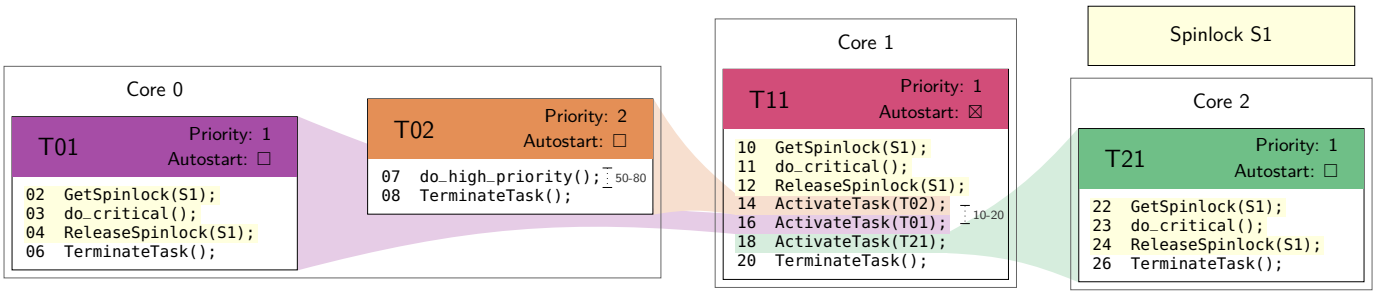


Fig. 1: Example application. Four tasks are distributed on three cores. Additionally, one spinlock exists. T11 starts automatically, the other tasks need an explicit activation which happens with the `ActivateTask` system call in T11. The yellow parts are critical sections they are guarded by protecting calls to the spinlock. The instruction numbers match the ABBs (Figure 2).

(i.e., source code). Its core assumption is that it is possible to statically detect all cross-core interactions at the system-call interface. For this, the underlying multicore RTOS and the application code have to provide the following basic properties:

(**1**) Deterministic scheduling policy, such as fixed-priority preemptive scheduling.

(**2**) Partitioned scheduling, that is, each task (software- or hardware-triggered) is pinned to exactly one core and all potentially blocking/scheduling-relevant cross-core interaction takes place via system calls.

(**3**) All system objects are known at compile-time.

(**4**) All system calls are explicit, that is, they are detectable at compile-time including their actual parameter values.

(**5**) [Optional] For each system call and each computation block (code between two system calls), the worst-case and best-case execution times (WCET, BCET) are known.

In practice, requirements 1–4 are already fulfilled or easy to achieve for event-triggered hard real-time control systems as they are basically a technical consequence of predictability and, thus, already mandated by the dominant coding and RTOS standards of the domain: Examples include ARINC 653 (avionics), which prescribes global fixed-priority scheduling within its multicore partitions [1] or multicore AUTOSAR (automotive) [2]. Without loss of generality, we therefore describe our approach in the following on the example of the system model mandated by the AUTOSAR-OS standard.

A. A Brief Example

In Figure 1, we exemplify some possible MultiSSE-based optimizations on an application using AUTOSAR-OS. We will use this example (with four tasks partitioned over three cores) throughout the paper, hence briefly introduce it here: The application employs some cross-core system calls (cross syscalls) that may cause an action on another core: `GetSpinlock` and `ReleaseSpinlock` protect a critical region across multiple cores. `ActivateTask` sets a task ready, which might have been assigned to another core.

The structure of this application facilitates a number of system-call specializations, which all could be obtained by MultiSSE:

Initially, T11 on Core 1 is the only running task (autostart is set), Cores 0, 2 are idling. Hence, T11 does not need to obtain Spinlock S1 to get exclusive access to the critical section (lines 7–9), as it is guaranteed that neither T01 nor T21 (the possible competitors) are running at this stage. The respective system calls could be safely elided.

T11 continues by activating T02 and T01. Both are cross syscalls, which in the case of `ActivateTask` requires the calling core to issue an inter-processor interrupt (IPI) to trigger a (potentially necessary) reschedule on the target core. In the case of T02 (line 10) the reschedule is necessary to wake up Core 0. In the case of T01 it depends: If T02 is still running, we could omit the reschedule (and, thus, the expensive IPI [16])[1], as T02 has priority over T01 and its `TerminateTask` in line 6 will issue a reschedule anyway. However, if T02 has already finished, Core 0 might again be idling and the IPI is required. By considering available best/worst-case execution time intervals, the MultiSSE can infer that the IPI can indeed be omitted. Additionally, in that case, the whole lock can be elided since all critical regions cannot interfere anymore.

So, in this example, the MultiSSE could detect three relatively expensive cross-core interactions (`Get/ReleaseSpinlock`, IPI in `ActivateTask`) that are infeasible by structure or timing.

B. MultiSSE: A Brief Overview

To provide detailed information about all application-RTOS interactions, the MultiSSE calculates the graph of all possible multi-core abstract system states (MAbSSs). A MAbSS represents the state of all CPU cores and RTOS objects at a specific point in time.

With existing techniques, it is possible to enumerate all core-local states [8]. However, to use this in a multi-core setting, we need to respect the relative timing of the cores to each other. The naive approach to get all MAbSSs is to enumerate all single-core states and then build the cross product between them. To avoid this combinatorial explosion, the MultiSSE calculates a reduced graph based on the fact that the cores do not interact arbitrarily with each other, but only at system-call sites that target other cores, which we call *cross syscalls*. Only at *these points*, the relative timings of only the *affected cores* are

[1]Depending on the CPU architecture, an IPI may have a cost tag of up to several hundred clock cycles on the sending core and up to a thousand clock cycles on the receiving core. The (unnecessary) interruption on behalf of a lower-priority task is a case of rate-monotonic priority inversion [7], [19].

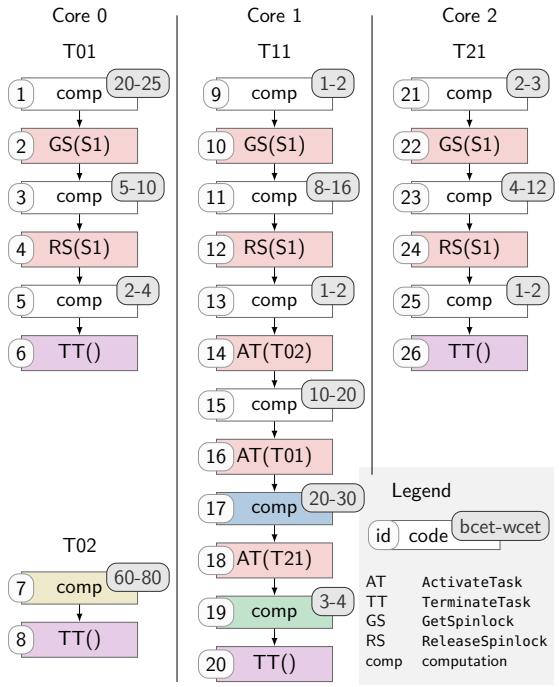


Fig. 2: ABBs of the example application. We can see a computation block (white) between each system call (in purple) or cross syscall (in red). The yellow, green, and blue blocks correspond to the equally colored blocks in Figure 4.

relevant. The MultiSSE analysis, therefore, construct a smaller graph, the multi-core state transition graph (MSTG), that makes these interaction points explicit. It consists of two node types (both special forms of a MAbSS): Local-core abstract system states (LAbSSs) describe the current state of a single core, while synchronization points (SPs) describe the current state of a set of cores together, which determines their relative position to each other. The main idea for the algorithm is to construct these two node types alternately. This happens by executing the following four steps:

(1) Initialization: The system start represents the first SP. Since all cores start, the state of them relative to each other is known. (Example from Figure 1: Evaluation starts with all cores in idle, except Core 1, which executes line 7 `GetSpinlock`.)

(2) Core-local analysis: Follow the control flow on each core independently to calculate a set of LAbSS until a cross syscall occurs. (Example: Core 0 and 2 are idling, Core 1 is at the execution of `GetSpinlock`, a cross syscall.)

(3) Pairing-partner search: For each cross syscall, calculate the list of all possible pairing partners, that is, all states in which all other affected cores may be at the same time. (Example: The lock is shared between all cores, so Core 1 needs to synchronize with Core 0 and 2. Therefore, the analysis has to find all states of Core 0 and 2 that may happen in parallel with Core 1. Normally, multiple pairing partners exist, in this case it is just the set of idle states of Core 0 and 2.)

(4) New SP construction: These states form a new entry SP, which the analysis evaluates into one or more exit SPs by calculating the effects of the cross syscall that leads to the SP. Starting at the introduced exit SPs, continue with step (2). (Example: The acquisition of the lock just goes through and does not change the idle status of the other cores, hence results in one exit SP, which is the next start for core-local analysis.)

III. Detailed MSTG Construction

In the subsequent sections, we describe the MultiSSE construction in further detail.

A. Control-Flow Graph Preparation

To ease the actual analysis, we preprocess the control-flow graph by partitioning it into atomic basic blocks (ABBs) [34]. An ABB is a sequence of instructions that fulfills the following properties:

(1) Every ABB has a single entry and a single exit instruction.

(2) Every ABB is of the type *syscall*, *call*, or *computation*.

(3) A *computation* ABB groups a sequence of instructions that are neither a system call nor a call to function that eventually issues a system call.

(4) A *syscall* ABB contains a single system call instruction.

(5) A *call* ABB contains a single call instruction.

(6) Two sys/call ABBs do not follow each other, but always have a (potentially empty/NOP) computation ABB in between.

Hence, ABBs point out the system calls and summarize all other instructions as far as possible. For the analysis, each ABB represents a valid position in the code, that is the conceptual program counter (PC) of the abstract machine.

The MultiSSE can optionally leverage timing information if the program's basic blocks come with a BCET and WCET. We map these times onto the ABBs. To further ease the analysis, sys/call ABBs are considered as happening atomically; we account their times to the surrounding computation ABBs. Figure 2 displays the ABB-Graph of our example (Figure 1).

B. Data Structures and Terms

The core data structure of the MSTG is the *multi-core abstract system state* (MAbSS):

$$\begin{aligned}\text{MAbSS} = (C, O) \quad & C \subseteq \mathcal{C}, O \subseteq \mathcal{O}_g \\ \mathcal{C} = \{c_0, \ldots, c_{\max}\} \quad & \text{set of all core contexts} \\ \mathcal{O}_g = \{o_0, \ldots, o_{\max}\} \quad & \text{set of all global OS-object contexts}\end{aligned}$$

Figure 3 visualizes one example. The context of an OS-object is object dependent (e.g., the spinning state of a spinlock). An OS-object is global if it belongs to more than one core (i.e., involved cross syscalls). A core context is a six tuple:

$$\begin{aligned}c \in \mathcal{C} = (\text{ABB}, \tau, \text{IRQ}, S, E, O) \\ \text{ABB} &= \text{the currently executed ABB (PC)} \\ \tau &= \text{the currently executed thread} \\ \text{IRQ} &= \text{interrupt enable bit (hardware state)} \\ S &= \text{call stack (path of called functions)} \\ E &\in \{\text{idle}, \text{waiting}, \text{normal}\} \text{ (execution state)} \\ O &= \text{set of core local OS-object contexts}\end{aligned}$$

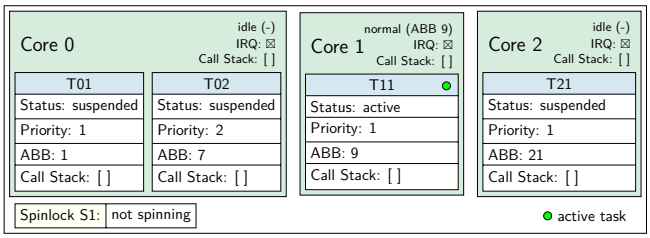


Fig. 3: The root SP of Figure 1 as an example of a MAbSS

```
1  def system_semantic(mabss, cpu_id):
2      active_abb = mabss.cpus[cpu_id].abb
3      if type(active_abb) == computation:
4          next_mabsss = follow_abb_chain(mabss)
5          return next_mabsss
6      if type(active_abb) == call:
7          return follow_call_chain(mabss)
8      if type(active_abb) == syscall:
9          if is_cross_syscall(active_abb) and not
10             mabss.contains_all_affected_cores(active_abb):
11             return "cross_syscall"
12         return interpret(active_abb)  # syscall specific
```

List. 1: Pseudo code of the system_semantic function.

So, a core context consists of its execution context (ABB, τ, and S), which defines its position in the control flow, its execution and IRQ state, which specifies the RTOS internal status of the core, and the set of core-local OS-object contexts, which describe OS objects bound to a specific core. The number of core contexts is variable: A MAbSS with exactly one core context forms a *local-core abstract system state* (LAbSS) that ignores the current state of all other cores:

$$\begin{aligned}\text{LAbSS} &= \text{MAbSS}_{|C|=1 \wedge O=\emptyset} \\ &= (C, O) \quad \text{with } |C| = 1 \text{ and } O = \emptyset \\ \cup_{\text{LAbSS}} &= \text{set of all LAbSSs}\end{aligned}$$

A MAbSS with multiple core contexts forms a *synchronization point* (SP), which determines the relative position of multiple cores to each other within the control flow:

$$\begin{aligned}\text{SP} &= \text{MAbSS}_{C \neq \emptyset} = (C, O) \quad \text{with } |C| \geq 2 \\ \cup_{\text{SP}} &= \text{set of all SPs}\end{aligned}$$

SPs are caused by a cross-core system call (cross syscall), a syscall ABB that interacts with another core. We call a LAbSS, which ABB is a cross syscall, a *cross-syscall LAbSS*. Since a cross syscall has a (potential) effect on each affected core, the originating LAbSS needs to be synchronized with LAbSSs on the affected cores, the *pairing partners*:

> A *pairing partner* to a cross-syscall LAbSS is a LAbSS of an affected core that can possibly coexist in time with the originating LAbSS.

The analysis needs to determine the set of paring partners for each affected core. The trivial upper bound for each of these sets is the set of *all* LAbSSs of the affected core. However, since the cross product of the paring partner sets determines the number of inserted SPs, it is crucial to further constrain coexistence and, thereby, the number of paring partners. We will discuss this later, but now define the MSTG:

$$\begin{aligned}\text{MSTG} =& (V, E) \\ T_E &= \{\text{local}, \text{global}, \text{fork}, \text{join}\} \\ &\quad \text{edge type} \\ V &= \{\text{MAbSS}_0, \ldots, \text{MAbSS}_n\} \\ E &= \{(x, y, f) | (x, y) \in V^2, f = V \times V \to T_E\} \\ f(x,y) &= \begin{cases} \text{local} & x \in \cup_{\text{LAbSS}}, y \in \cup_{\text{LAbSS}} \\ \text{global} & x \in \cup_{\text{SP}}, y \in \cup_{\text{SP}} \\ \text{fork} & x \in \cup_{\text{SP}}, y \in \cup_{\text{LAbSS}} \\ \text{join} & x \in \cup_{\text{LAbSS}}, y \in \cup_{\text{SP}} \end{cases}\end{aligned}$$

The MSTG is a graph consisting of MAbSSs which are connected by edges. Each edge has a specific type depending on the connected node types. Local edges connect two LAbSSs and capture a core-local state transition. Global edges connect an entry SP with an exit SP and capture the transition caused by a cross syscall. Join edges connect a set of LAbSSs with the resulting entry SP, thus, capturing the pairing of a cross syscall state with its pairing partners. Fork edges connect an exit SP with the resulting set of LAbSSs when the analysis continues to calculate the effects of all cores individually.

For a better understanding, Figure 4 illustrates an excerpt of the MSTG that results from the example (Figure 1). The figure does not contain the initial state. It starts at SP *a*, which is based on the evaluation of `ReleaseSpinlock(S1)` (ABB 12). Core 0 and 2 are idling at this point. We will come back in detail to that example later.

C. The Algorithm

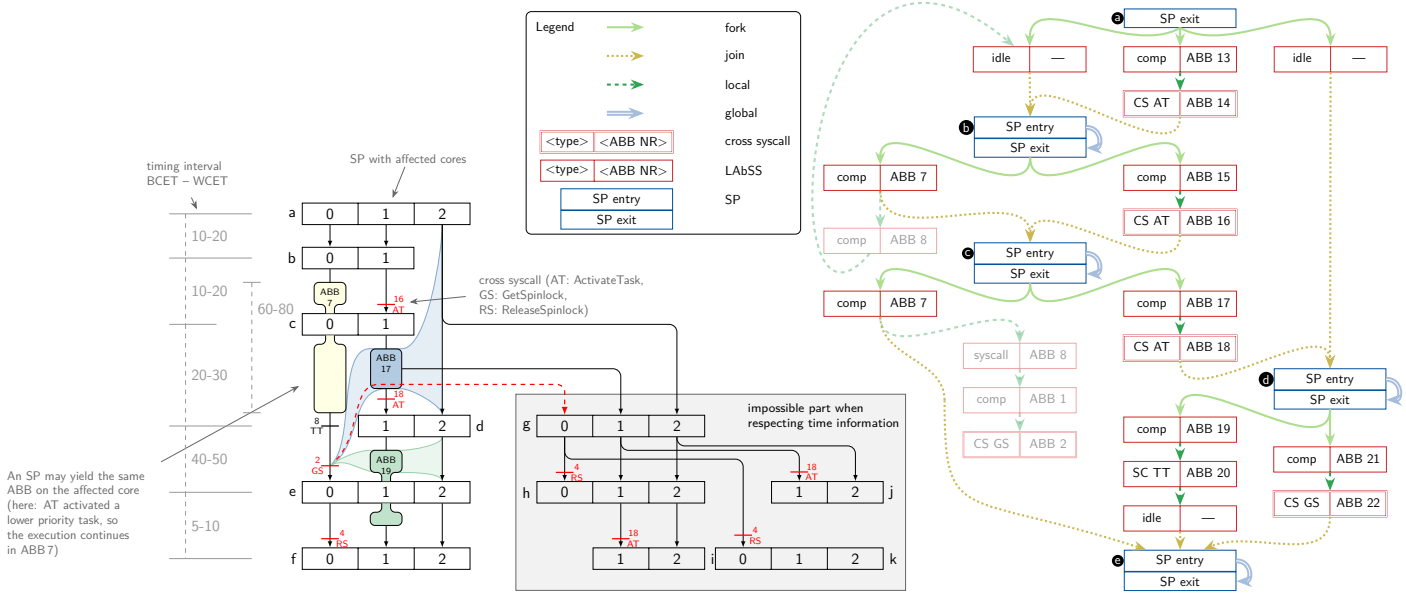
1) Initialization: The MultiSSE starts with the construction of the first SP, the *root SP*. It symbolizes the starting point of the scheduler on all cores and is therefore deterministic. In this state, all cores are in idle state except the ones that should autostart a task according to the system specification. Additionally, the algorithm initializes all RTOS objects according to the system specification. Figure 3 shows the root SP of the example.

2) Single-Core Analysis: From the previous SP, the algorithm calculates the further processing of each core individually. It, therefore, splits the SP into LAbSS by extracting the specific local core context, adds fork edges from the SP to those LAbSSs, and omits all global object contexts.

The MultiSSE merges MAbSSs where possible. Therefore, if it has already evaluated the LAbSS in a previous iteration, it skips the state and its successors. Otherwise, it follows the ABB graph on each path by calculating a transition between two states for every ABB. For that, it needs a transition function which it also uses for an SP-SP transition:

$$\begin{aligned}(\text{MAbSS}_{g+1,0}, \ldots, \text{MAbSS}_{g+1,n}) &= \texttt{system\_semantic}(\text{MAbSS}_g, c) \\ g &= \text{generation} \\ c &= \text{the core to be evaluated}\end{aligned}$$

In the local-core case, c is always the only core context that is present in the LAbSS.



(a) MSTG (simplified) (b) MSTG (raw)

Fig. 4: An extract of the MSTG of the example (Figure 1). In (b) the MSTG as described is illustrated, while (a) shows a simplified version. The simplified version summarizes the entry and exit SP and merges LAbSSs, fork, local, and join edges. All red lines mark a state where a cross syscall happens. The yellow, blue, and green parts mark states where the core resides in the same ABB. Sometimes (for the yellow and green states), they are interrupted by an SP. The grey numbers describe the time interval between two SPs or the length of one ABB execution.

The `system_semantic` function works dependent on the type of the ABB of the active core context c in the MAbSS (Listing 1): In the case of a computation ABB, it follows the control-flow and constructs – for each follow-up ABB – a new LAbSS with the updated ABB for the active core. In the case of a call ABB, the analysis follows it by updating the core's call stack and set the new ABB of the called function (function return will pop the call stack again). For a syscall ABB, the analysis divides further: In the case of a cross syscall, the analysis stops the core-local analysis (here follows an SP). Otherwise, it interprets the system-call-specific effects on the state and emits one or multiple follow-up LAbSSs.

A single system call may produce multiple follow-up states. An example is `ReleaseSpinlock()`: When synchronizing 3 cores while 2 of them are spinning, the MultiSSE emits two follow-up MAbSSs, one for each of the two cores receiving the lock.

After executing `system_semantic`, the analysis merges the new LAbSSs with existing ones, if possible. For all remaining states, it iterates and applies the `system_semantic` function again. This step terminates if either all emitted new states are merged with previous ones (for example, the idle state has a transition to itself) or ends in a cross syscall. When terminated, the analysis continues with the next step.

3) Pairing-Partner Search: Once the MultiSSE detects a cross syscall, it searches for possible pairing partners. First, it determines the affected cores, which are all cores the cross syscall synchronizes. Second, the analysis searches for LAbSSs on the affected cores that can exist at the same time. To not have to consider *all* LAbSSs as paring partners, we take the last SPs into account. Every to-be-synchronized core set has preceding SPs that synchronize at least the respective cores (that is, a superset).[2] We call an SP that precedes the cross-syscall LAbSS and synchronizes a superset of cores a *parent SP*:

> A *parent SP* of an SP is an SP that synchronizes a superset of cores and for which exists at least one path in the MSTG between parent SP and SP that does not contain another parent SP.

Conceptually, the parent SP acts as a barrier in the control flow: The cross syscall in this context must happen after it, so the analysis can limit its search space to the graph part after a parent SP. Given the graph structure (caused by the state merging), multiple such parent SPs can exist that are reachable via disjoint paths. Because of that, the analysis also stores the chain of intermediate SPs that lead to each parent and performs every following step for each path. For this search, the analysis uses an adapted breadth-first search (BFS).

For a better understanding, we want to use Figure 4 as an example for such a search. In detail, we want to focus on the pairing process of `GetSpinlock(S1)` (ABB 2). This system call affects all three cores. Therefore, the analysis needs to find possible pairing partners for Core 1 and 2. The search for parent SPs leads to SP a which synchronizes Core 0, 1 and 2 with one path over SP c and SP b.

Since the MultiSSE merges MAbSS whenever possible, there may be loops to the parent SP or other paths that contradict

[2]Ultimately, the root SP synchronizes all cores.

the cross syscall (for example, an SP that synchronizes the same core). Therefore, the search first filters the graph: It removes all ingoing edges to the parent SP, thus, eliminates circles. It also throws away every SP that either synchronizes the originating core and is not part of the path to the parent SP, or synchronizes only cores that are not affected by the cross syscall.

All SPs on the path to the parent SP form a relation in time between specific cores. Therefore, for each core, a later executed SP that synchronizes a subset of affected cores in the chain may mask the parent SP (it acts as a closer barrier for this reduced core set). It is enough to start the partner search from this point on. We name these SPs *core-local parent SPs*.

> A *core-local parent SPs* of a pairing partner is the last SP on the path from the parent SP to the pairing partner that is also an element of the path from the parent SP to the cross-syscall LAbSS.

For example, SP *c* lies on the path from the parent SP *a* to ABB 2 and already synchronizes Core 0 with Core 1, so the search for pairing partners of Core 1 can start at SP *c*. For Core 2, the search must start directly at the parent SP, since all other SPs on the path do not synchronize Core 2.

When the analysis has calculated all core-local parent SPs, it continues to find pairing corridors: For every core, it iterates over all belonging states starting at the core-local parent SP. Each state can be restricted by following SPs, which may synchronize with another affected core and act as a barrier in its search area. As a consequence, the decision for a pairing area on one core can restrict the search space on the other cores, thus forming pairing corridors.

For example, by starting the search on Core 1, the algorithm finds the blue area first. All states in this area are possible pairing partners (here it is just ABB 17). The blue area is restricted by SP *d*, which synchronizes with Core 2, another core for which we want to find pairing partners, thus, restricting the search for Core 2 to the blue corridor. The SP *d* is a valid follow-up SP relative to SP *a* and SP *c*, so the green corridor also needs to be searched.

After determining a corridor, the analysis builds the product of each LAbSS in it. For that, it only uses states that execute in a time interval, so computation or idling states.

The example contains just one state for each core on the corridor, so the cross syscall triggers two SPs: SP *e* (for the green corridor) and SP *g* (for the blue corridor).

4) SP Construction: With the cross syscall and the list of all pairing partners, the MultiSSE can create a new entry SP in the graph and connect it with all preceding LAbSSs (the cross syscall and its partners) via join edges. It therefore combines all core contexts of the preceding LAbSSs, which are disjunct by design, and takes the global contexts from the parent SP. Afterward, the analysis interprets the new entry SP with the `system_semantic` function, which results in a set of new exit SPs. For example, the interpretation of the `ActivateTask(T02)` (ABB 14) in SP *b* (entry) leads to SP *b* (exit), in which the state of T02 changes to "active" and the core executes ABB 7, the first ABB of this task.

The algorithm puts each exit SP in the work queue and starts the whole process again. Additionally, it determines the need for reevaluations, that is, existing SPs that need another evaluation:

- If the new SP has a predecessor in time that synchronizes another set or a superset of cores, the newly created state may add additional pairing partners for cross syscalls of this predecessor.
- If the to-be-created SP is already present and the algorithm just needs to add an edge between the old and the new SP, this adds a new system call order to the graph, so the SP needs to be reevaluated concerning this new edge.

IV. Using Timing Information

The SPs in the MSTG describe an ordering of cross syscalls, since the graph connects them with edges that model an order in time. Each SP connects a cross syscall with a set of pairing partners, that is, other core states that can potentially run simultaneously. Naively, this would be all LAbSS of the respective core, which we already have reduced by the described pairing partner search to those states that can actually coexist by structure (because they happen in the control flow after a previous SP).

In the following, we go further and also take execution times into account: At run time, many structurally possible combinations of pairing partners are actually impossible because of the times the cores spent in their different computation ABBs. To make use of this, the MultiSSE can leverage timing information already during construction: While searching for pairing partners, the algorithm immediately excludes combinations that are impossible because of timing constraints, that is, it neither creates the respective SPs nor evaluates them. This can drastically reduce the resulting number of SPs and, thus, analysis time.

In general, the algorithm works by comparing intervals on the different cores. For that, it assumes that each computation ABB has a BCET and WCET assigned. The idea of the MSTG is to synchronize a core only when needed (in an SP) and let it "run" independently until the next cross syscall. So, in an SP we know the exact point in time of different cores relative to each other. We can use that by just comparing the time interval of the cross syscall relative to its preceding SP with the times of the potential pairing partners on the other cores (relative to the same SP). To compare the intervals, we build a linear inequality system, which is an equation system with additional interval constraints for each variable. Such a system forms a linear programming problem and can be solved efficiently with an LP-solver [22]. For its functioning, the algorithm uses four types of time intervals:

ABB: The execution time interval of an ABB. This is a property of the ABB (Section III-A).

L–L: A core-local execution time interval, given by a starting and ending LAbSS. The analysis calculates this interval type based on the underlying ABB intervals.

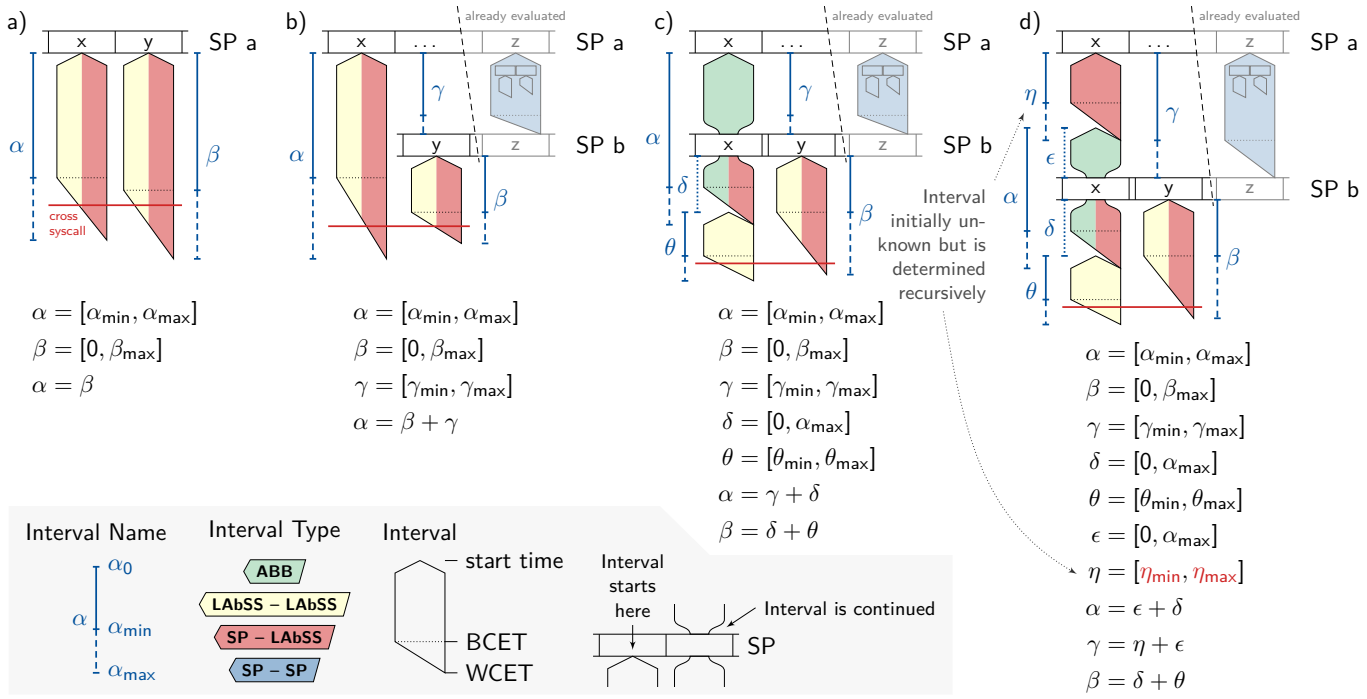


Fig. 5: All different building blocks for building the inequality system, sorted by their rising complexity. The to-be-compared intervals as well as the resulting inequality systems are shown. Some intervals have multiple types (indicated by their two colors). The requested interval is always that of the cross syscall to the last SP (the interval of β given all side constraints).

S–S: A time interval between two SPs. It describes all possible points in time in which the successor SP can exist relative to its predecessor.

S–L: A time interval starting at an SP and ending at a LAbSS that indicates an execution chain of one core relative to an SP. The pairing-partner search compares exactly these types of intervals, which the algorithm has to calculate.

For each interval, we will denote its starting time with t_0, its minimum execution time with t_{min} and its maximum execution time with t_{max}. To build the inequality system, we can differentiate four different building blocks of rising complexity, which we illustrate in Figure 5.

A. The Inequality System

1) Simple intervals: Figure 5a shows the straightforward case. We see two different intervals on Core x and y, which are synchronized by SP *a*. Core x triggers a cross syscall, and we are interested in the information on whether the pairing partner on Core y can execute at the same time. We, therefore, want to compare the time interval α of SP *a* to the cross syscall with the time interval β of the LAbSS on core y starting at SP *a* (both of type **S–L**), which we need to calculate. In this case, Core x starts a LAbSS with a new ABB directly after SP *a* that eventually leads to the cross syscall (with possible LAbSSs in between). As a consequence, we need to calculate the interval α that starts and ends at a LAbSS (type **L–L** with inclusive start and exclusive end). Since all LAbSSs execute a single ABB which has a given BCET and WCET (type **ABB**), the problem reduces to a classical BCET or WCET analysis for which we chose a path-based approach [20], [13].

We need to compare α with the **S–L** interval β of the LAbSS of Core y that also starts at SP *a*. It denotes the time in which the LAbSS can take place. Therefore, β starts directly at SP *a* (time 0) and going to the WCET of the LAbSS.

With the correct limits, we are now able to formulate the equation $\alpha = \beta$ and solve the inequality system. Thus, we obtain a new time interval that specifies the limits wherein the cross syscall with exactly this combination of pairing partners (and hence the resulting SP) can take place relative to SP *a*. To avoid later recalculations, we store the resulting **S–S** interval as an attribute of a new edge between the SPs. If the inequality system has no solution, the SP is impossible.

2) Intervals with different SPs as a starting point: Figure 5b is mostly equivalent to Figure 5a except that the pairing partner does not start at the same SP. So, to compare the intervals, we also need the **S–S** interval γ that describes the time between SP *a* and SP *b*. The algorithm has calculated this time interval already in a previous step: During the construction of SP *b* it has built an inequality system whose solution specifies the interval γ. Therefore, it knows all needed interval limits and can compare them with $\alpha = \beta + \gamma$.

3) SPs that interrupt ABBs: In Figure 5a and Figure 5b all LAbSSs with their respective executing ABBs start directly at the SPs. However, often an SP interrupts the execution of an ABB and, thus, its **ABB** execution interval. For example, assume a cross syscall `GetSpinlock` that pairs with another core that does not hold the lock. In this case, the cross syscall

does not change the LAbSS of the other core. Or assume an `ActivateTask` that activates a task on another core with a lower priority than the currently executing one. In that case, the LAbSS changes (the state context is different) but not the executing ABB. In these cases, some time of the ABB interval may have passed before the SP which the algorithm has to respect.

Figure 5c models exactly that. The ABB interval α is the time interval of an ABB whose execution is interrupted by SP *b*. We are interested in the S–L interval between SP *b* and the cross syscall which is a combination of the L–L interval θ and δ, the remaining part of α. In this case, the ABB execution starts directly at SP *a*, so we can calculate δ as $\alpha - \gamma$ which is the S–S interval between SP *a* and SP *b* and already calculated.

4) Interrupted ABBs that do not start at an SP: Figure 5d illustrate the most complex case. Here, the execution of the interrupted ABB does not start directly at an SP but is preceded by other LAbSSs. So, to be able to calculate δ, we need to know the length of the first part of α (ϵ) which, in turn, is the result of $\gamma - \eta$. γ is already calculated η is unknown. However, this maps to the same problem again but with another (prior) SP set and can be calculated recursively. The algorithm stops eventually (when it reaches the root SP). But, we can optionally terminate earlier by falling back to the previous execution time of an interrupted ABB as zero or its WCET (the ABB was executed in complete or not at all).

B. Challenges

For the ABB timings, we assume atomic system calls by accounting the actual execution time to the surrounding computation ABBs. However, this assumption does not hold for ABBs of the affected cores. Since the cross syscall triggers an IPI, the affected core is interrupted in an unknown computation ABB, which the SP makes explicit. However, the IPI handling time on the affected core needs to be accounted as well. For this, our current implementation simply uses an additional constant term (i.e., the WCET of all IPI handlers) in the inequality system for every affected core. In future work, we want to support syscall-specific timing overheads.

Another challenge are loops. Given a cross syscall that executes within a loop, the affected cores could be interrupted multiple times, maybe even in the same ABB. This forms a loop in the MSTG, which the pairing partner search has to respect. When using the simple paring-partner search (without timing information), we can just ignore the loop: We are only interested in the fact, *whether* a synchronization is possible, not *when*. But for the paring-partner search with timing information, we need to determine the number of traversals. To remain sound, we therefore disable timing calculations for this particular search space whenever we detect an unbounded loop, that is, we set the time frame to $[0, \infty]$. However, commonly, loops are bounded in hard real-time system. A bounded loop can be virtually unrolled, thus, omitting the problem altogether.

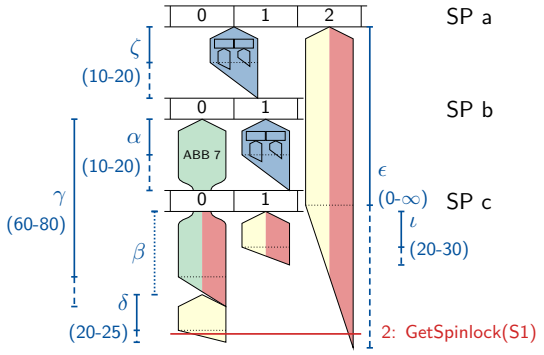


Fig. 6: Relevant parts of Figure 4 for interval calculation of the blue corridor.

C. A Concrete Example

To illustrate the technique, we want to build a full inequality system of one part of the example (Figure 6). We are using `GetSpinlock(S1)` (ABB 2) with the pairing partner of the blue corridor (ABB 17). The MultiSSE first calculates the time interval to the last SPs. This results in the interval $\delta = [20, 25]$ for the time between the end of the state that executes ABB 7 and the state that executes ABB 2, the interval $\iota = [0, 30]$ for the execution of the state that executes ABB 17 relative to SP *c*, and the interval $\epsilon = [0, \infty]$ for the time of the idling state of Core 2 starting from SP *a*. The time intervals between SP *a*, SP *b*, and SP *c* are already evaluated in previous iterations (the times are displayed in grey in Figure 4). We further need the remaining time of the state that executes ABB 7 relative to SP *c* (case c). ABB 7 (interval γ) starts its execution directly after SP *b*, so we can build the equation by comparing it with the interval between SP *b* and SP *c* (α). This results in the following inequality system:

$$
\begin{aligned}
\alpha + \beta &= \gamma \\
\beta + \delta &= \theta && \text{(time between SP } b \text{ and ABB 2)} \\
\theta &= \iota && \text{(compare Core 0 with Core 1)} \\
\theta + \zeta + \alpha &= \epsilon && \text{(compare Core 0 with Core 2)} \\
\alpha &= [10, 20] && \text{(time between SP } b \text{ and SP } c\text{)} \\
\beta &= [0, 80] && \text{(remaining execution interval)} \\
\gamma &= [60, 80] && \text{(BCET and WCET of ABB 7)} \\
\delta &= [20, 25] && \text{(ABB 7 to ABB 2)} \\
\iota &= [0, 30] && \text{(execution of ABB 17)} \\
\epsilon &= [0, \infty] && \text{(idling state on Core 2)} \\
\zeta &= [10, 20] && \text{(interval from SP } a \text{ to SP } b\text{)}
\end{aligned}
$$

This equation system has no solution: β evaluates to $[40, 70]$, leading to $[60, 95]$ for θ, which renders $\theta = \iota$ impossible. As a consequence, it eliminates the whole right part of the MSTG (the marked grey area), which leads to all cases where the lock spins which – as a result – is useless and can be elided. For the whole MSTG resulting from the presented example, the analysis generates 45 SPs without considering timing information and 12 SPs including timing information.

V. External Interrupts

Even though not mentioned explicitly, the algorithm is already prepared for dealing with external interrupt sources: From the perspective of the affected core, a cross syscall is an external interrupt (technically received as an IPI).

This allows us to model other external interrupts sources in exactly the same way: Each external interrupt source gets a virtual core assigned that runs a virtual interrupt program, a set of virtual ABBs that trigger the interrupt in form of a cross syscall according to its specification. If the interrupt has inter-arrival times specified (e.g., as in AUTOSAR [2]), we map this to an ABB with respective BCET and WCET interleaving the interrupt cross syscalls. The virtual interrupt cores start their execution at system start or after an activating system call (e.g., for alarms). The actual interrupt handling routine is done as part of the `system_semantic` function, which schedules the core context to the correct entry point. Periodic interrupts lead to a loop in the virtual interrupt program, which is unbounded without further information, resulting in a similar problem when respecting timing information as with other loops (discussed above). However, giving the interrupt's inter-arrival times, we can constrain the maximum number of triggers within a hyper period, thus bounding the loop again.

Interrupts are a significant SP source, since, in theory, every interrupt could arrive at "any time". Modeling them all can result in a state explosion. In practice, however, plenty of them are likely to be impossible because of logical or timing constraints. The complexity of our algorithm scales with the number of SPs, so omitting impossible ones is desirable. We therefore optionally support specifying a task set belonging to an interrupt and omitting the interrupt while any of such tasks is executed to support the interrupt's "logic of action": An interrupt should not interrupt its own handling routine.

Even though not yet completely supported in our implementation, our approach (to model interrupt sources as normal programs and interrupt triggering as cross syscalls) makes it possible to integrate additional (arbitrary complex) trigger conditions into the analysis to further reduce the number of SPs: For instance, it would be relatively easy to specify by such program that the send-buffer-empty interrupt of a communication device may only occur 10-20 msec after a `SendBuffer` invocation or that a button-released IRQ may not happen before a button-pressed. The same holds for platform-specific hardware behavior: If the interrupt controller supports advanced features, such as IRQ multicasts or round-robin or priority-based routing of IRQs, this could be incorporated into the implementation of the virtual cross syscall and its set of affected cores.

VI. Possible Applications

The MSTG represents all possible system states, which gives us the knowledge to perform partial system-call specialization. We have not yet implemented this or explored this in depth – this is a topic of future work –, but want to provide some first applications to demonstrate the potential of the MSTG. For the single-core case, the MSTG matches the graph produced by related work, namely the SSE, so all system-call specializations are possible that are already detected by it [8], [9]. In concrete, that are pre-calculating core local scheduler decisions up to a static lookup table [10], applying fault detection mechanisms to prevent transient hardware faults and lay the base for more advanced techniques like tightening the worst-case energy consumption or improving the memory footprint through efficient stack sharing [11], [39]. While not implementing them directly, we see no difficulties in applying the optimizations based on the knowledge of the MSTG instead of the single core equivalent. Pre-calculating core local scheduler decisions and efficient stack sharing is not meaningfully adaptable to work across cores but remains functional in a multi-core environment. Applying additional information for fault detection and tightening the worst-case energy consumption should be extendable to work across cores and may give additional benefits. Additionally, we can employ the MultiSSE to detect further multi-core specific cases for specialization.

1) Unnecessary IPIs: If an `ActivateTask` activates a task on another core, the standard RTOS implementation will set the target task state to "ready" and trigger an IPI to cause a reschedule on the affected core. However, the costly IPI is not necessary if we know for sure that the affected core will be executing a task with a higher priority. The Trampoline OS [4], an open source AUTOSAR implementation, mitigates that that problem by sending the IPI only when necessary by having a global scheduler with the high cost of a global kernel lock, which also complicates a real-time analysis. The MSTG gives us exactly this knowledge: If no SP originating from an `ActivateTask` leads to the execution of this task in the resulting MAbSS, we can safely omit the IPI.

For example, the `ActivateTask(S01)` (ABB 15) builds a single SP. This SP only interrupts the execution of ABB 7, thus rendering the IPI here useless.

The usage of `SetEvent` may also trigger an unnecessary IPI: If a task waits for an event that has a higher priority than the currently running one and that event is set from another core, an IPI is necessary to trigger a reschedule. In all other cases, the event just sets a flag in the task context. In the MSTG we are able to detect these cases as well: If an SP that originates from a `SetEvent` constructs a MAbSS with a waiting state in a task and this specific task is running and not waiting in the following MAbSSs anymore, we cannot avoid the IPI.

2) Lock Elision and Deadlock Detection: For locks, we are able to detect several patterns: A deadlock happens, if a `GetSpinlock` is executed on a core while already holding the lock (recursive locking). The analysis catches this already at construction time, since the `system_semantic` function has to handle this case. Another source of deadlocks are nested locks that are taken in different orders. This is forbidden by the AUTOSAR specification, but not easy to check at compile time, thus can happen in real implementations. However, with the MSTG, all system-call orders are statically detected, so we are able to check for incorrect lock nesting.

Taking a lock with `GetSpinlock` is unnecessary if we

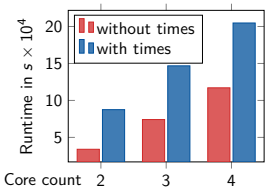


Fig. 7: The algorithm's runtime in dependence of the amount of cores. All other parameters are hold constant. For every core 150 systems are generated.

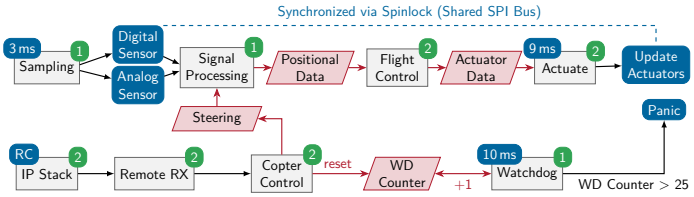


Fig. 8: *I4Copter* flight-controller application (adapted from [37]). Eight tasks (grey) are mapped on two cores (green).

can prove that `GetSpinlock` does not spin at this point. If all invocations of `GetSpinlock` on this lock never spin, the complete lock could be elided. The MSTG contains the necessary information to detect such a situation: If no MAbSS after one call of `GetSpinlock` contains the lock in the spinning state, the call is unnecessary. If no MAbSS contains the lock in the spinning state, the complete lock is unnecessary.[3] Dropping the entire lock additionally frees the lock's data and related control structures.

In the example, the `GetSpinlock(S1)` (ABB 10) cannot result in a spinning state. If we additionally consider timing information, all locking operations cannot result in a spinning state, which renders the whole lock useless.

VII. Evaluation

To evaluate the findings of our approach and the resulting freedom for system-call specialization, we implemented the MultiSSE in ARA[4], a whole system optimizer with a focus on RTOS specialization based on LLVM [14], [25]. We implemented the algorithm in more then 5000 LoC of Python/C++ (including 349 for the SSE core, 1681 specific for the MultiSSE, 1236 for the AUTOSAR model).

We applied the MultiSSE onto several different systems. Thereby, we analyze the reduction of possible spinning states for the spinlocks, avoidable IPIs and surely set events. We compare the outcome of the MultiSSE with and without timing information against the plain information from the system specification. First, we analyze the conformance test cases from the Trampoline [4] project. Second, we investigate the example application and several handcrafted applications to demonstrate its principle workings. Third, we analyze the *I4Copter* [37], a safety-critical embedded real-time control system (quadrotor helicopter) adapted to a dual-core platform. Fourth, we validate the wider applicability by generating synthetic benchmarks with varying characteristics.

A. Trampoline conformance tests

Trampoline is an open-source OSEK/VDX and AUTOSAR compliant RTOS implementation [4]. To test the conformance of the MultiSSE, we analyzed the provided multi-core related test cases and their applications (without additional timing information). These applications use a broad set of the AUTOSAR functionalities including task activations across cores, spinlocks, alarms, interrupts, timers, and events. Thereby, we are able to produce an MSTG for each of those 12 applications, which may be used for further optimizations. As a first result, the analysis shows, that there are neither superfluous locks nor situations resulting in an unexpected deadlock.

B. The Example Application

Our example application of Figure 1 uses the spinlock `S1` for mutual exclusion during a critical section. The system specification permits three tasks to use the spinlock. Therefore, all those tasks may possibly spin on this lock. Applying MultiSSE without timing information results in an MSTG consisting of 137 vertices connected via 816 edges. Using this graph, we are able to detect that Task `T11` never gets in conflict with the other tasks acquiring the lock. Hence, those calls may be elided from `T11`. As it is unclear how long the computation phases of `T02` and the delta between the activations of `T02` and `T01` are, no prediction about the need of an IPI for `T01`'s activation is possible.

Taking timing information into account (Figure 2), further reduction of the MSTG is possible. The resulting graph consists of 70 fewer states and 603 fewer transitions. The analysis shows that the activation of `T01` always occurs during the computation phase of the higher priority `T02`. Hence, the task activation could be reduced to setting the ready bit. Sending an IPI to trigger the scheduler could be omitted. This results in less priority- inverting activity on Core 0 and less interrupt-introduced jitter for the execution time of the high priority task `T02`. Furthermore, no other task acquiring `S1` results in no spinning state, and hence, the spin-lock calls are useless and can be removed.

C. The I4Copter

The *I4Copter* is an embedded real-time flight controller software formerly designed for a single-core platform, that was kindly provided to us by Ulbrich et al. [37]. For this evaluation, we adapted the task set to distribute the two major components *sensor gathering* and *actuation* onto two cores. Those two parts work mostly without interference. The only shared resource is the SPI bus. Therefore, we introduced a spinlock to mutually exclude the access from the two cores. Figure 8 depicts the components and their interworking.

[3]One may think that such unnecessary and, hence, guaranteed contention-free lock taking is basically for free, but depending on the RTOS-internal implementation and the target architecture, it may trigger costly cache-coherence messages between cores [15]. On Linux, taking a lock on a single core system results in a measurable performance impact [33].

[4]https://github.com/luhsra/ara

Execution of the MultiSSE on an Intel i5-6400 quad-core system with 32 GiB of main memory takes 12.79 seconds. Applying MultiSSE generates an MSTG with 294 vertices and 2143 edges. Applying timing information reduces it by 140 vertices (48%) and 1496 edges (70%). Since we assume that BCET and WCET times are given by an additional external analysis we generated times that represent the logical application structure[5]. Additionally, the graph shows that the spinlock, introduced to guard the SPI bus shared between the two cores, will never result in a spinning state.

D. Synthetic Benchmarks

To validate the wider applicability and usefulness of our approach and implementation, we systematically generated synthetic applications. As a benchmark generator, we use the (publicly available) generator presented in [11] and extended it for multi-core systems. There are six parameters characterizing the generated systems: #cores (2,4,6), #threads (up to 15 per core), #spinlocks (1, 3), #lock-users (2, 6), #cross-core-task-activations (10%), #events (0, 5). We use a pseudo-random number generator with varying seed to get different but reproducible results. The system generation consists of the following steps: First, we create a directed acyclic thread-dependency graph with #threads nodes. Those threads are mapped onto the specified number of cores. We assign each thread a unique random priority for priority-based scheduling. Then, thread activation is distributed between core-local and cross-core activation following the given percentage. Those threads neither activated by a parent thread nor via a cross-core activation are set to be automatically started at system startup. For each spinlock, #lock-users using threads are chosen such that all threads using a lock originate from different cores. A single thread may use multiple locks and there may be multiple threads on a single core accessing different locks. Furthermore, we add additional #events dependencies resulting in tasks waiting for events to be set from other tasks.

We generated 872 applications with varying parameters. Executing the MultiSSE results in graphs with 464 nodes and 2837 edges on average. Thereby, in total 1890 IPIs were detected as avoidable. From the total of 7143 attempts to acquire a spinlock, 4695 are detected to never result in a spinning lock. The run-time of the MultiSSE for these examples range from 23 s to 7825 s with a mean of 1893 s.

Applying randomly generated timing information regarding best-case and worst-case execution times for each ABB, results in an average reduction of the MSTG by 30 nodes (6%) and 476 edges (17%). The MultiSSE with timing information has a run-time of 23 s to 7825 s with a mean of 1908 s. Additionally, in Figure 7 we show the runtime dependent on the amount of cores. For this test, we only varied the amount of cores while holding every other variable constant.

[5]We selected the times based on an educated guess. Especially signal processing and flight control calculation have a high payload and thus get higher times assigned.

VIII. Discussion

The biggest caveat of the MultiSSE is an exponential growing of calculating every possible system state. We do mitigate this by aggressively merging or omitting states whenever possible:

- We use ABBs as underlying control-flow representation. They effectively reduce the control-flow states in contrast to basic blocks or plain instruction by subsuming everything in one block that is not relevant for the analysis.
- We merge single-core states, whenever possible. If the analysis determines that an SP does not modify the local-core state, it merges it with the prior state (and skips further evaluations). For example, a `GetSpinlock` system call modifies a global context and, thus, never a LAbSS.
- We build an SP only when it is of interest. An essential part of the analysis consists of the calculation of the minimal set of SPs required to still capture the effects of all cross syscalls.

Since the analysis belongs to the building process and thus is executed only a few times, we see from the evaluation part that the analysis overhead seems reasonable. Further reduction will be possible by incorporating more knowledge about the RTCSs into our analysis. The MultiSSE already utilizes worst/best-case execution times to determine impossible SPs. In future work, additional sources should be exploited to further reduce the SPs amount: One example is to further explicit ordering of semantically dependent system calls on different cores by incorporating their temporal logic of actions [24]. Another possibility is that application developers provide annotations or explicit barriers as SP to bring all cores in sync at suitable points within the application's logic.

For timing calculations, the MultiSSE requires a stricter set of real-time properties like bounded loops or falls back to disabling them. Note, however, that having timing information is optional – it just improves analysis results and time. Furthermore, the availability of precise timing information is not an all-or-nothing requirement. If the timing analysis fails for some states, it can still be employed for the states after the next SP.

While we have not formally proven MultiSSE's soundness, we are convinced that it holds true. The analysis result is an over-approximation in all cases, making too optimistic optimizations impossible. Assuming the SSE's soundness, we only have to demonstrate soundness for cross-core interactions added by MultiSSE. The OS specification and our system model clearly restrict cross-core interactions to a finite set of system calls and interrupts. Under the assumption of known system-call arguments (already needed for the SSE), the MultiSSE is sound by calculating the superset of all possible effects of all system calls and interrupts. The MultiSSE aims not to be correct, in doubt it always chooses the pessimistic approach, which – in the worst case – may result in missing optimizations.

IX. Related Work

System-call specialization based on static analysis is not new. For dynamic embedded systems, Bertran et al. introduce a global view of the interactions between application and

operating system by constructing a global control-flow graph (CFG) [5] that connects the system-call entry point to their corresponding application call sites. With that, they can eliminate dead code of uncalled system call functions, which reduces code size. Rajagopalan et al. [32] try to detect and remove unused kernel parts by statically analyzing and rewriting the kernel binary. In contrast to this work, both analyses respect neither OS semantics nor multiple cores.

The first respecting OSEK semantics are Barthelmann et al. [3]. They use it to reduce the costs of task context switching. A static analyzer constructs an inference graph describing the mutual preemptions of basic blocks, which allows to facilitate an optimized inter-task register allocation and static context switch code generation. In contrast to this work, the analysis is flow insensitive. Hence, the influence graph includes superfluous combinations which considering task activations could avoid. Also, since the work uses OSEK semantics, it targets only single-core systems.

Also, specializations exist for general-purpose operating systems. Pu et al. developed the Synthesis kernel, which includes a code synthesizer that produces application-dependent optimized code paths at run time for frequently called system calls [31]. They provide manually optimized templates which are filled out during run-time which leads to an overall speed-up. In contrast to the dynamic Synthesis system, our approach takes place ahead of time, which also allows system call removals.

With SWAN [35], Schuster et al. developed a whole-system WCET analyzer that builds a state transition graph to achieve tighter bounds for the WCET. However, SWAN is not able to model multi-core systems and does not aim for specialization.

Dietrich et al. introduced the SSE [8] which we used as a base for the MultiSSE. While we share the algorithms for the single-core analysis, we adapted the data structures and functions to work for multiple cores. We additionally introduced the SPs mechanism and completely changed the interrupt handling. As a follow-up analysis for the SSE, Dietrich et al. further introduced another method – the System-State Flow – to construct (much faster) a more course-grained state graph that can be used for similar specializations [9]. While this greatly reduces the algorithm's execution time, it does this at the cost of state precision. For the MultiSSE, we need the full set of possible states, from only which the full set of SPs can be constructed, which is why we chose to build upon the SSE.

Regarding the use of spinlocks in embedded real-time systems, Wieder et al demonstrate the problematic semantics of spinlocks for schedulability analyses of multi-core real-time applications [38]. With our approach, we are able to elide the locks from the application or derive the order of lock acquirements to further improve those analyses.

With special regard to multi-core analysis, Mittermayr et al. modeled multi-core synchronization patterns with the help of Kronecker-algebra, a method to model parallel executing automata in form of a matrix [28], [29]. It represents all possible intersections, which, with knowledge of barriers (semaphores), is reducible to actual possible paths and computable in a lazy way. While the resulting graph models cross-core interactions in a flow-sensitive manner, it does concentrate on barriers only and is not modeling other RTOS primitives. Furthermore, the approach is not used for system-call specialization, but for deadlock detection and WCET analysis. Haur et al. use high-level colored time Petri nets to model a multi-core application for AUTOSAR including the RTOS code [17], [18]. With their approach, they were able to find incorrect locking patterns within the RTOS code by formal analysis. However, in contrast to our approach, they do not focus on unnecessary locks and need to map manually from the application code to the model. Miné et al. extended Astreé, an abstract interpreter to support multi-core programs to which Kästner et al. later added support for AUTOSAR applications [27], [21]. While they are doing a flow-sensitive analysis and taking locks into account, they collect all cross-core interactions in a flow-insensitive fact pool and iterate the analysis until it becomes stable. While their approach scales better and does not concentrate on operating system facts only, they are neither flow sensitive between cores nor take their possible orders into account. Schwarz et al. transform multi-core programs into a set of constraints with side effect to reason over all local and global flow paths [36]. In contrast to the MultiSSE their approach detects run-time errors regarding global data usage and works for POSIX threads.

Regarding deadlock detection, Engler et al. searched for them via an interprocedural flow-sensitive but unsound static analysis [12], which – in contrast to our approach – provides false positives and aims application development. Kroening et al. present a method for sound deadlock detection for the C language using pthreads [23]. For that, they build a similar control flow graph but construct a special lock graph, which does not model arbitrary thread interactions.

X. Conclusions

With MultiSSE, we presented an algorithm that creates a multi-core state transition graph of all possible cross-core interactions in a partitioned real-time system. It works by traversing the application's CFG and iteratively building a graph of all core-local RTOS states which it connects with explicit synchronization points for a cross-core interaction.

To reduce the number of synchronization points, the algorithm can additionally make use of externally provided timing information. With that, we could reduce the MSTG vertices up to 48% and edges up to 70%.

With help of the MSTG, we were able to extract the necessary information for cross-core system-call specializations: We showed the possibility of removing superfluous (not spinning) locks and locking system calls and the possibility to avoid IPIs that trigger an unnecessary reschedule. In addition to this, the MSTG provides all necessary information for single-core system-call specializations as described in prior works.

We have validated our approach with unit tests as well as 872 synthetically generated applications, showing 66% of attempts to take a spinlock as useless and a total of 1890 of IPIs as avoidable. Additionally, we applied our approach to an adapted version of the *I4Copter*, an embedded real-time flight controller, and were able to classify the existing lock as useless.

XI. Acknowledgments

We like to thank the anonymous reviewers and our shepherd for their feedback and fruitful comments. This work has been supported by the German Research Foundation (DFG) under the grant no. LO 1719/4-1.

The source code and evaluation artifacts are available at:
`https://www.sra.uni-hannover.de//p/multisse-rtas23`

References

[1] AEEC. Avionics application software standard interface (ARINC specification 653p1-4), 2015.

[2] AUTOSAR. Specification of operating system (version 5.1.0). Technical report, Automotive Open System Architecture GbR, February 2013.

[3] Volker Barthelmann. Inter-task register-allocation for static operating systems. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES '02)*, pages 149–154, New York, NY, USA, 2002. ACM Press. doi:10.1145/513829.513855.

[4] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline: An OpenSource implementation of the OSEK/VDX RTOS specification. In *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.*, pages 62–69, Washington, DC, USA, September 2006. IEEE Computer Society Press. doi:10.1109/ETFA.2006.355432.

[5] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*, Washington, DC, USA, June 2006. IEEE Computer Society Press.

[6] Jim Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.

[7] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press. doi:10.1109/RTAS.2006.34.

[8] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, New York, NY, USA, June 2015. ACM Press. doi:10.1145/2670529.2754963.

[9] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM Transactions on Embedded Computing Systems*, 16(2):35:1–35:25, 2017. doi:10.1145/2950053.

[10] Christian Dietrich and Daniel Lohmann. OSEK-V: Application-specific RTOS instantiation in hardware. In *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '17)*, New York, NY, USA, June 2017. ACM Press. doi:10.1145/3078633.3078637.

[11] Christian Dietrich and Daniel Lohmann. Semi-extended tasks: Efficient stack sharing among blocking threads. In Sebastian Altmeyer, editor, *Proceedings of the 39th IEEE Real-Time Systems Symposium 2018*, Nashville, Tennessee, USA, 2018. IEEE Computer Society Press. doi:10.1109/RTSS.2018.00049.

[12] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, New York, NY, USA, 2003. ACM Press. doi:10.1145/945445.945468.

[13] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving wcet bounds by abstract execution. In *ECRTS 2011*, 2011.

[14] Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. ARA: Static initialization of dynamically-created system objects. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*, pages 400–412, May 2021. doi:10.1109/RTAS52030.2021.00039.

[15] Andy Glew and Wen-Mei Hwu. Snoopy cache test-and-test-and-set without execessive bus contention. *SIGARCH Comput. Archit. News*, 18(2):25–32, may 1990. doi:10.1145/88237.88240.

[16] Per Hammarlund, James B. Crossland, Shivnandan D. Kaushik, and Anil Aggarwal. Inter-processor interrupts, 2003. US Patent 8,984,199 B2. URL: https://patents.google.com/patent/US8984199B2/en.

[17] Imane Haur, Jean-Luc Béchennec, and Olivier Henri Roux. Formal schedulability analysis based on multi-core RTOS model. In *29th International Conference on Real-Time Networks and Systems*, RTNS'2021, page 216–225, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453417.3453437.

[18] Imane Haur, Jean-Luc Béchennec, and Olivier H. Roux. Formal verification of the inter-core synchronization of a multi-core rtos kernel. In Adrian Riesco and Min Zhang, editors, *Formal Methods and Software Engineering*, page 140–155. Springer International Publishing, 2022.

[19] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*, pages 204–213. IEEE Computer Society Press, December 2009. doi:10.1109/RTSS.2009.18.

[20] Niklas Holsti. Computing time as a program variable: a way around infeasible paths. In Raimund Kirner, editor, *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1660, doi:10.4230/OASIcs.WCET.2008.1660.

[21] Daniel Kaestner, Antoine Miné, Andrew Schmidt, Heinz Hille, Laurent Mauborgne, Stephan Wilhelm, Xavier Rival, Jérôme Feret, Patrick Cousot, and Christian Ferdinand. Finding all potential run-time errors and data races in automotive software. volume 2017-March. SAE International, 2017. doi:10.4271/2017-01-0054.

[22] L. G. Khachiyan. *A polynomial algorithm in linear programming*, volume 244. 1979.

[23] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 379–390, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970309.

[24] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, may 1994. doi:10.1145/177492.177726.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Washington, DC, USA, March 2004. IEEE Computer Society Press.

[26] Peter Marwedel. *Embedded System Design*. Springer-Verlag, Heidelberg, Germany, 2006.

[27] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel c programs. *Logical Methods in Computer Science*, 8(1), mar 2012. URL: https://doi.org/10.2168%2Flmcs-8%281%3A26%292012, doi:10.2168/lmcs-8(1:26)2012.

[28] Robert Mittermayr and Johann Blieberger. A generic graph model for wcet analysis of multi-core concurrent applications. *Journal of Software Engineering and Applications*, 9:182–198, 01 2016. doi:10.4236/jsea.2016.95015.

[29] Robert Mittermayr and Johann Blieberger. Deadlock and wcet analysis of barrier-synchronized concurrent programs. *Computing*, 103:749–770, 2021. doi:10.1007/s00607-017-0555-8.

[30] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29.

[31] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[32] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. Automatic operating system specialization via binary rewriting. 2005.

[33] Florian Rommel, Christian Dietrich, Michael Rodin, and Daniel Lohmann. Multiverse: Compiler-assisted management of dynamic variability in low-level system software. In *Fourteenth EuroSys Conference 2019 (EuroSys '19)*, New York, NY, USA, 2019. ACM Press. doi:10.1145/3302424.3303959.

[34] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)*, pages 34–41, Washington, DC, USA, May 2010. IEEE Computer Society Press. doi:10.1109/ISORC.2010.11.

[35] Simon Schuster, Peter Wägemann, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Proving real-time capability of generic operating systems by system-aware timing analysis. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 318–330, 2019. doi:10.1109/RTAS.2019.00034.

[36] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving thread-modular abstract interpretation. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 359–383. Springer International Publishing, 2021.

[37] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM Press.

[38] Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS '13)*, pages 45–56. IEEE Computer Society Press, December 2013. doi:10.1109/RTSS.2013.13.

[39] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems. In Sebastian Altmeyer, editor, *Proceedings of the 30th Euromicro Conference on Real-Time Systems 2018*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECRTS.2018.24.

[40] Zephyr Project homepage. https://www.zephyrproject.org/.