Static Race Detection for RTOS Applications

Rishi Tulsyan
Indian Institute of Science Bangalore, India
rishitulsyan@iisc.ac.in

Rekha Pai
Indian Institute of Science Bangalore, India
rekhapai@iisc.ac.in

Deepak D'Souza[1]
Indian Institute of Science Bangalore, India
deepakd@iisc.ac.in

Abstract

We present a static analysis technique for detecting data races in Real-Time Operating System (RTOS) applications. These applications are often employed in safety-critical tasks and the presence of races may lead to erroneous behaviour with serious consequences. Analyzing these applications is challenging due to the variety of non-standard synchronization mechanisms they use. We propose a technique based on the notion of an "occurs-in-between" relation between statements. This notion enables us to capture the interplay of various synchronization mechanisms. We use a pre-analysis and a small set of not-occurs-in-between patterns to detect whether two statements may race with each other. Our experimental evaluation shows that the technique is efficient and effective in identifying races with high precision.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Static analysis, concurrency, data-race detection, RTOS

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2020.57

Supplementary Material `https://bitbucket.org/rishi2289/static_race_detect/`

Acknowledgements The second author want to thank University Grants Commission (UGC) India for the Dr. DS Kothari Post Doctoral Fellowship EN/17-18/0039.

1 Introduction

Real-Time Operating Systems (RTOSs) are small operating systems or microkernels that an application programmer uses as a library to create and manage the execution of multiple tasks or threads. The programs written by the application programmer are called RTOS applications and are programs typically written in C or C++ that are compiled along with the RTOS kernel library and run on bare metal processors. Much of embedded software today, ranging from home appliances to safety-critical systems like industrial automation systems and flight controller software, are implemented as such programs.

An RTOS application comprises multiple threads (even if these are typically run on a single core) and hence they need to protect against concurrency issues like data races. Two statements are involved in a data race if they are conflicting accesses to a shared memory location and can happen "simultaneously" or one after another. Data races can lead to unexpected and erroneous program behaviours, with serious consequence in safety-critical applications.

[1] corresponding author

While detecting data races is important, doing this for RTOS applications is a challenging problem. This is because these programs use a variety of non-standard synchronization mechanisms like dynamically raising and lowering proirities, suspending other tasks and the scheduler, flag-based synchronization, disabling and enabling interrupts, in addition to the more standard locks and semaphores. A look at the ArduPilot flight control software [1] which is written in C++ and runs on the ChibiOS RTOS shows several instances of *each* of these synchronization mechanisms being used. Standard techniques for race detection like lockset analysis [28] or for priority-ceiling based scheduling and flag-based synchronization [23, 22], or the disjoint-block approach of [8] for disabling interrupts, would not be precise enough as they do not handle the first two mechanisms mentioned above. Extending the disjoint-block approach for these synchronization mechanisms seems difficult.

Instead, in this work we adapt the disjoint-block approach of [8] to focus on a weaker notion of "not occuring in between". Essentially, a statement s_2 does not *occur in between* a statement s_1 if it is not possible for a thread running s_2 to preempt a thread while it is running s_1. If s_1 and s_2 cannot occur in between each other they also cannot race. We identify six patterns or rules that ensure that a statement cannot occur in between another. We take the help of a pre-analysis to identify dynamic priority ranges as well as task suspension information, for each statement in an application. Then for each pair of conflicting statements we check if the rules tell us that they cannot occur in between each other.

We have implemented our analysis for FreeRTOS applications (FreeRTOS [5] is a popular open source RTOS), and analyse several small benchmarks from the literature as well as a fragment of the ArduPilot [1] code, which we translate as a FreeRTOS application. Our analysis runs in fractions of a second with an overall precision rate of 73%.

2 Overview

We begin with an overview of our technique with an illustrative example adapted from a FreeRTOS demo application. The application, shown in Fig. 1, begins by creating two task threads `t1` and `t2` that run the task functions `prod` and `cons` respectively, both at

```
void main(...) {                Prio  Susp
1. item = count = 0;
2. xTaskCreate(prod,...,1, t1);
3. xTaskCreate(cons,...,1, t2);
4. vTaskStartScheduler();
}

void prod(...) {
10. for( ; ; ) {                 1,1   -
11.   vTaskSuspend(t2);          1,1   -
12.   item = 5;                  1,1   cons
13.   count = count+1;           1,1   cons
14.   vTaskResume(t2);           1,1   cons
15. }                            1,1   -
}

void cons(...) {
20. for( ; ; ) {                 1,1   -
21.   temp = item;               1,1   -
22.   vTaskPrioritySet(NULL, 2); 1,1   -
23.   count = count-1;           2,2   -
24.   vTaskPrioritySet(NULL, 1); 2,2   -
25. }                            1,1   -
}
```

Figure 1 A producer-consumer FreeRTOS app.

priority 1. Once the scheduler is started in line 4 of `main`, the two threads begin executing in a round-robin manner, preempting each other whenever the time slice is over (unless one thread is suspended, or the running thread has raised its priority above the other thread). The `prod` thread protects its accesses to the shared variables `item` and `count` by suspending the `cons` thread in line 11, and resuming it in line 14 after the access. Similarly, the `cons` thread protects its access to `count` by temporarily raising its priority to 2 in line 22.

We are interested in statically detecting potential data races in this application. We give a more precise definition of a race in Sec. 4, but for now we can take it to mean that two statements access a shared variable with at least one writing to it (we call these "conflicting" accesses), and these statements happen one after the other in some execution of the application.

Our analysis begins by first performing a data-flow analysis to identify the minimum and maximum dynamic priorities that each statement can run at. The computed values are shown on the second column from the right in the figure, and represent the priorities just before the statement. Thus at line 23 in `cons` the min and max priorities are both 2. We also perform a "suspended" analysis to find out at each point, which are the tasks that are guaranteed to be suspended. These values are shown in the rightmost column.

Next, for each conflicting pair of accesses s_1 and s_2, we check whether s_2 can "occur in between" s_1. Essentially, s_2 can occur in between s_1 if there is an execution in which while s_1 is executing, a context-switch may happen and s_2 eventually executes before the context switches back to s_1. If s_2 cannot occur in between s_1, and vice-versa, then one can *rule out* s_1 and s_2 being involved in a race. To check the "occur in between" relation we use a small set of rules (see Fig. 4 in Sec. 5) which tell us when s_2 *cannot* occur in between s_1. Thus, by the "Suspend" rule (C1), we can conclude that statements in line 21 and 23 cannot occur in between the statements in line 12 and 13 (since the `cons` task is suspended here). Similarly, by the "Priority" rule (C2), it follows that line 13 cannot occur in between line 23 (since it runs at a higher priority). This allows us to conclude that the accesses to `count` in lines 13 and 23 cannot race. However for the accesses to `item` in lines 12 and 21, we are unable to show that line 12 cannot occur in between 21, and hence our analysis declares them as potentially racy. Indeed, these two accesses are racy.

We note that analyses like [23, 8] do not handle these synchronization mechanisms and would be unable to declare the accesses in line 13 and 23 to be non-racy.

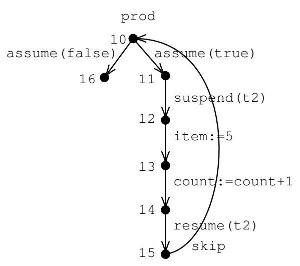
3 Interrupt-Driven Applications

In this section, we describe the syntax and semantics of an Interrupt-Driven Application (IDA). An IDA program is essentially a set of thread functions, which are run by dynamically created threads during execution. The functions are of two types: *task* functions which will be run by threads that are created dynamically at different priorities, and *ISR* functions which are run as Interrupt Service Routines triggered by hardware interrupts, at fixed priorities above that of task threads. There is a designated *main* function which is run by the *main* thread which is the only thread running initially. The *main* thread may create other task threads and then "`start`" the scheduler, at which point the created threads and ISR threads are enabled. The scheduler runs the task threads according to a highest-priority-first basis and time-slices within threads of the same priority. ISR threads can be triggered at any point of time, preempting task threads or lower priority ISR threads.

The thread functions can use a variety of commands, listed in Tab. 1, to perform computation or influence the way they are scheduled. Task threads are created using the `create` command. The command creates a new thread, which runs the specified task

```
main:                   prod:                   cons:
1. item:=0;             10. for(; ;) {          20. for(; ;) {
2. count:=0;            11.   suspend(t2);      21.   temp:=item;
3. create(prod,1,t1);   12.   item:=5;          22.   set_priority(t2,2);
4. create(cons,1,t2);   13.   count:=count+1;   23.   count:=count-1;
5. start;               14.   resume(t2);       24.   set_priority(t2,1);
6.                      15. }                   25. }
                        16.                     26.
```

(a) Example IDA program.



(b) CFG of `prod`.

Figure 2 Example program and the CFG representation of `prod`.

function at the specified priority. High priority threads share execution time with low priority threads using the `set_priority`, `suspend`, and `block` commands. These commands can lead to re-scheduling of the threads, thereby giving other threads a chance to execute. The `set_priority` command sets the priority of a task thread, `suspend` suspends the execution of a task thread, and `block` (representing blocking commands like "delay" or "receive message") blocks the execution of the current task thread. A suspended task thread can be resumed with the `resume` command. A blocked task thread resumes after a non-deterministic amount of time. Task threads can suspend and resume the scheduler with `suspendsched` and `resumesched`, respectively. When the scheduler is suspended the currently running task thread can be preempted only by an ISR thread, and not by other task threads. Threads can also disable and enable interrupts with `disableint` and `enableint`, respectively. When interrupts are disabled, no preemption can occur. Tasks can synchronize accesses to shared variables by acquiring and releasing locks with `lock` and `unlock` commands, respectively.

More formally, an IDA program P is a triple $\langle V, M, F\rangle$ where V is a finite set of integer-valued global variables, M is a finite set of locks, and F is a finite set of thread function names, with a designated one called *main*. Each function A in F has an associated Control Flow Graph (CFG) $G_A = (L_A, ent_A, ext_A, inst_A)$, where L_A is the set of *locations*, ent_A and ext_A are respectively the *entry* and *exit* locations in L_A, and $inst_A \subseteq L_A \times cmd(V, M) \times L_A$ is the set of *instructions* of the CFG. Here $cmd(V, M)$ is the set of commands in Tab. 1 over the variables V and locks M. Each function A in F also has an associated *type*, $type(A)$, which is one of *task* or *ISR*. While task threads are created during execution at priorities specified in the `create` command, ISR threads run at a fixed static priority. We assume that during execution task threads can have priorities upto a constant value $m \in \mathbb{N}$ (which we fix for all IDA programs), while ISR threads have distinct priorities which are greater than m. If $\{f_1, \ldots, f_k\}$ are the functions of type *ISR*, then without loss of generality we assume their priorities to be $m + 1, \ldots, m + k$ respectively. The IDA version of the FreeRTOS application from Fig. 1 is shown in Fig. 2.

Some notation will be useful going forward. For a program P, the instructions of P, denoted $inst_P$, is the union of instructions in the thread functions of P, and locations in P, denoted L_P, is the union of the locations in the thread functions of P. An IDA program allows standard integer and Boolean expressions over V. For an integer expression e, Boolean expression b, and an environment ϕ for V, $[\![e]\!]_\phi$ denotes the integer value that e evaluates to in ϕ, and $[\![b]\!]_\phi$ denotes the Boolean value that b evaluates to in ϕ. For a map $f : X \to Y$ and elements x, y which may or may not be in X or Y, we use the notation $f[x \mapsto y]$ to denote the function $f' : X \cup \{x\} \to Y \cup \{y\}$ given by $f'(x) = y$ and for all z different from x, $f'(z) = f(z)$.

Table 1 IDA Basic Commands.

Command	Description
`skip`	Do nothing.
$x := e$	Assign the value of expression e to variable x.
`assume`(b)	Enabled only if expression b evaluates to *true*; does nothing.
`create`(A, p, t)	Create task thread with func A, prio p, and store thread id in variable t.
`set_priority`(t, p)	Set priority of task thread t to p. When the first parameter is NULL, set priority of current thread. Allowed only in task function.
`suspend`(t)	Suspend task thread t. When the parameter is NULL, suspend current thread. Allowed only in task function.
`resume`(t)	Resume task thread t. Allowed only in task function.
`suspendsched`	Suspend scheduler. Disables switching to other task threads.
`resumesched`	Resume the scheduler. Enables switching to other task threads.
`disableint`	Disable interrupts and suspend the scheduler.
`enableint`	Enable interrupts and resume the scheduler.
`lock`(l)	Acquire lock l. Blocks if l is not available.
`unlock`(l)	Release lock l.
`block`	Block the current task thread. Re-enable after non-deterministic delay.
`start`	Start scheduler and enable interrupts. Called only by *main*.

We can now define the semantics of an IDA program $P = \langle V, M, F \rangle$ as a labeled transition system $\langle S, \Sigma, \Rightarrow, s_0 \rangle$, whose components are defined as follows. Let $f_1, \ldots, f_k$ be the thread functions of type *ISR*, with priorities $m+1, \ldots, m+k$ respectively.

The set of states S contains tuples of the form $s = \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc, \phi, r, i, ss, id \rangle$, where

- $\mathcal{B}$, $\mathcal{S}$, and $\mathcal{R}$ are sets of thread ids (which we assume to be simply integers) representing the set of *blocked* task threads, *suspended* task threads, and *ready* task threads, respectively. The sets $\mathcal{B}$, $\mathcal{S}$, and $\mathcal{R}$ are pairwise disjoint. We denote the set of threads created so far by $\mathcal{T} = \mathcal{B} \cup \mathcal{S} \cup \mathcal{R}$.
- $\mathcal{P} : \mathcal{T} \to \mathbb{N}$ gives the current priority of each thread.
- $\mathcal{A} : M \rightharpoonup \mathcal{T}$ is a partial map giving us the thread that has acquired a particular lock.
- $\mathcal{F} : \mathcal{T} \to F$ gives the function associated with a thread.
- $pc : \mathcal{T} \to L_P$ gives the current location of a thread t in the CFG of $\mathcal{F}(t)$.
- $\phi \in V \to \mathbb{Z}$ is a valuation for the variables.
- $r \in \mathcal{R}$ is the currently running thread, while $i \in \mathcal{R}$ is the interrupted task thread.
- ss is a Boolean value indicating whether the scheduler is suspended ($ss = true$) or not, while id is a Boolean value indicating whether interrupts are disabled ($id = true$) or not.

The initial state s_0 is $\langle \emptyset,\ \{1, \ldots, k\},\ \{0\},\ \{0 \mapsto 0, 1 \mapsto m+1, \ldots, k \mapsto m+k\},\ \emptyset,$ $\{0 \mapsto main, 1 \mapsto f_1, \ldots, k \mapsto f_k\}, \lambda t \in \mathcal{T}.ent_{\mathcal{F}(t)},\ \lambda x \in V.0,\ 0,\ 0,\ true,\ true \rangle$. Thus initially, no threads are blocked, ISR threads $1, \ldots, k$ (with priorities $k+1, \ldots, k+n$ respectively) are disabled, and the main thread 0 with priority 0 is ready and also running. No locks are acquired. The threads $0, 1, \ldots, k$ are associated with their functions. All the threads are at their entry locations and all variables are initialized to zero. The interrupted thread is taken to be 0, the scheduler is suspended, and interrupts are disabled.

The transition relation $\Rightarrow$ is given as follows. Consider a state s expressed as the tuple $s = \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc, \phi, r, i, ss, id \rangle$, a thread $t \in \mathcal{T}$, and an instruction $\iota = (l, c, l')$ in $\mathcal{F}(t)$. Then we have $s \Rightarrow_\iota s'$ iff one of the following rules is satisfied. Each rule says that if the conditions on command c and state s, specified in the antecedent of a rule (above the line), hold then $s \Rightarrow_\iota s'$, specified in the consequent of the rule (below the line), holds. We use $task(t)$ to indicate that t is a task thread (i.e. $(type(t) = task)$ and $ISR(t)$ to indicate that t is an ISR thread.

In the interest of space, only few rules are shown here. The full semantics can be found in Arxiv. The ASSIGN is a simple rule on assignment statement. The ASSIGN-INT rule shows how interrupts are handled while CREATE-CS and CREATE-NS rules show how the execution of a statement can lead to context switch and no switch, respectively, and the START rule shows how the threads get running. For the ASSIGN-INT rule given below, the condition $pc(t) = l = ent_{\mathcal{F}(t)}$ should hold while for others $pc(t) = l$ needs to be true.

$$\frac{c = x := e \qquad t = r}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi[x \mapsto [\![e]\!]_\phi], r, i, ss, id\rangle} \ \textsc{assign}$$

$$\frac{c = x := e \qquad t \in \mathcal{R} \qquad ISR(t) \qquad t \neq r \qquad \mathcal{P}(t) > \mathcal{P}(r) \qquad id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi[x \mapsto [\![e]\!]_\phi], t, r, ss, id\rangle} \ \textsc{assign-int}$$

$$\frac{c = \texttt{create}(A, p, v) \ \ t = r \ \ task(t) \ \ A \in F \ \ type(A) = task \ \ ts \notin \mathcal{T} \ \ (p \leq \mathcal{P}(r) \vee (ss \vee id) = true)}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R} \cup \{ts\}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}[ts \mapsto A], pc[t \mapsto l', ts \mapsto ent_A], \phi[v \mapsto ts], r, i, ss, id\rangle} \ \textsc{create-ns}$$

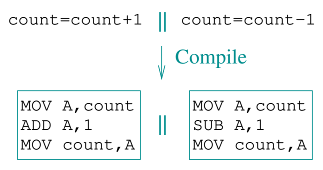
$$\frac{c = \texttt{create}(A, p, v) \ \ t = r \ \ task(t) \ \ A \in F \ \ type(A) = task \ \ ts \notin \mathcal{T} \ \ p > \mathcal{P}(r) \ \ (ss \vee id) = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R} \cup \{ts\}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}[ts \mapsto A], pc[t \mapsto l', ts \mapsto ent_A], \phi[v \mapsto ts], ts, i, ss, id\rangle} \ \textsc{create-cs}$$

$$\frac{c = \texttt{start} \ \ t = r = 0 \ \ (ss \vee id) = false \ \ \exists ts \in (\mathcal{S} \cup \mathcal{R}).task(ts) \ \wedge \ \mathcal{P}(ts) = max(\{\mathcal{P}(u) | u \in \mathcal{S} \cup \mathcal{R} \ \wedge \ task(u)\})}{s \Rightarrow_\iota \langle \mathcal{B}, \emptyset, \mathcal{S} \cup \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, false, false\rangle} \ \textsc{start}$$

An *execution* σ of P is a finite sequence of transitions in the transition system defined. $\sigma = \tau_0, \tau_1, \cdots, \tau_n$, where $n \geq 0$ and there exists a finite sequence of states $s_0, s_1, \cdots, s_{n+1}$ in S such that s_0 is the initial state and $\tau_i = s_i \Rightarrow s_{i+1}$ for each $0 \leq i \leq n$.

4 Data Races and the Occur-in-Between Relation

We use the notion of data races introduced by Chopra et al [8], which is a general notion that applies to programs with non-standard synchronization mechanisms. The definition essentially says that two statements in a program race if (a) they are conflicting accesses to a memory location and (b) they may happen in parallel, in that notional "`skip` blocks" around these statements overlap with each other in some execution of the program. The definition is meant to capture the fact that when these two statements are compiled down to


```
count=count+1  ||  count=count-1
```

```
MOV A,count        MOV A,count
ADD A,1       ||   SUB A,1
MOV count,A        MOV count,A
```

instructions of a processor, the interleaving of these instructions may lead to undesirable behaviours of the program which don't correspond to any sequential execution of the two statements. For example in the figure alongside, the conflicting accesses to `count` may get compiled to the instructions shown, and the interleaving of these two blocks of instructions may lead to unexpected results like `count` getting decreased by 1 despite both blocks having completed.

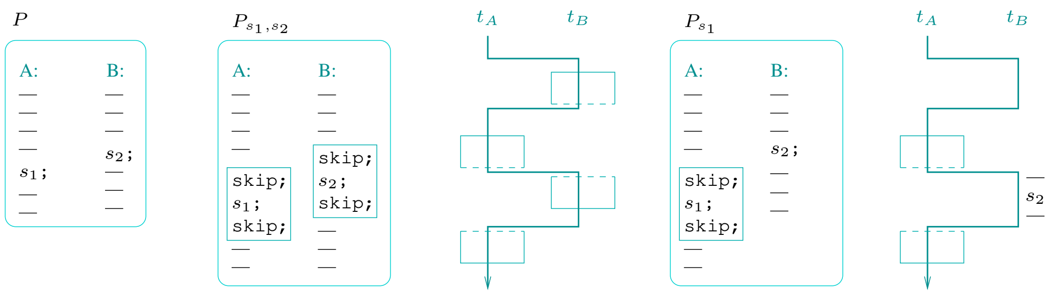


Figure 3 A program P; its transformation P_{s_1,s_2}; an execution of P_{s_1,s_2} in which the skip blocks overlap and witnesses that s_1 and s_2 MHP in P; the program P_{s_1}; and an execution of P_{s_1} which witnesses occurrence of s_2 in between s_1.

We now define these notions more formally in our setting. Let us fix an IDA program P. Let s_1 and s_2 be two instructions in P, with associated commands c_1 and c_2 respectively. We restrict ourselves to the case where c_1 and c_2 are assignment or assume statements. We say s_1 and s_2 are *conflicting accesses* to a variable x if they both access x and at least one of them writes x. Let P_{s_1} denote the program obtained from P by inserting `skip` statements immediately before and after s_1. Similarly, let P_{s_1,s_2} denote the program obtained from P by inserting `skip` statements immediately before and after both s_1 and s_2. We say s_1 and s_2 *may happen in parallel* (MHP) in P if there is an execution of P_{s_1,s_2} in which the two `skip`-blocks interleave (i.e. one block begins in between the other). These terms are illustrated in Fig. 3. We use the convention that A and B represent the static thread functions, while t_A and t_B represent dynamic threads that run the functions A and B respectively, with an optional subscript indicating the priority at which the thread was created. Finally we say s_1 and s_2 are involved in a *data-race* (or simply are *racy*) in P, if they are conflicting accesses that may happen in parallel in P.

It will be convenient for us to use a stronger notion than MHP called "occurs-in-between" while reasoning about IDA programs. Once again, if s_1 and s_2 are statements in P, we say that s_2 can *occur-in-between* s_1 if there is an execution of P_{s_1} in which s_2 occurs sometime between the first `skip` and the second `skip` around s_1. In this case we write $s_1 \lhd s_2$, and $s_1 \not\lhd s_2$ otherwise. The definition of $s_1 \lhd s_2$ is illustrated in the right side of Fig. 3. While it is immediate that if s_2 occurs in between s_1 then they also MHP, a weaker version of the converse is also true:

▶ **Proposition 1.** *Let s_1 and s_2 be two statements in an IDA program P. Then s_1 MHP s_2 iff either s_1 occurs in between s_2 or s_2 occurs in between s_1.* ◀

Thus to conclude that s_1 and s_2 cannot MHP (and hence not race) it is enough to show that s_1 and s_2 cannot occur in between each other.

5 Occur-In-Between Rules

In this section we focus on statically computing a conservative (i.e. under-) approximation of the *cannot*-occur-in-between relation for an IDA program, by giving rules for identifying this relation. To illustrate the typical issues we need to keep in mind while framing these rules, consider the example program alongside. Task threads $A4$, $B2$, and $C4$ are created at priority 4,2, and 4 respectively. In the normal course statement s_2 in $B2$ would not be able to occur in between s_1 in $A4$ as $A4$ runs at a higher priority than $B2$. However, (the thread that runs) $C4$ may suspend $A4$ just before it executes s_1, block itself, and allow $B2$ to run. Thus s_2 can occur in between s_1.

```
A4:      B2:      C4:
—        —        —
—        —        // t runs A4 at prio 4
s1;      s2;      suspend(t);
—        —        block;
—        —        ...
                  resume(t);
```

We will make use of the following terminology for an IDA program P. Let s be a statement in thread function A in P. We say s may run at priority p if there is an execution of P in which a thread t runs A and executes statement s at a priority of p. We say (p, q) is the dynamic priority of s if p and q are respectively the minimum and maximum priorities that s can run at. Similarly, we say that the dynamic priority of a thread function A (or a block of code in A) is (p, q) if p and q are respectively the minimum and maximum priorities at which any statement in A (or the block of A) can run. Finally, we say that a task function A *may suspend* another task function B in P, if A contains a statement of the form `suspend`(t), and there is an execution of P in which the statement is executed when the thread id t points to task function B (that is t runs task B). We say the statement `suspend`(t) in P *must suspend* (or simply *suspends*) a task B if t takes on a unique thread id at this point along any execution of P, and this thread id is the only thread that runs B. In this case, we will denote such a statement by `suspend`(B).

We now proceed to propose sufficient conditions under which one statement in an IDA program cannot occur-in-between another statement in the program. Let us fix an IDA program P. Let s_1 and s_2 be statements in thread functions A and B respectively (A and B could be the same thread function). The following conditions (C1)–(C6) below are meant to be sufficient conditions that ensure that s_2 cannot occur in between s_1. In the rules below, by a statement s in a thread function A being enclosed in a `suspend-resume` block we mean there is a path in the CFG of A which contains s, begins with a `suspend`, ends with a `resume`, and has *no* intervening `resume` statement; and similarly for other kinds of blocks. Each of these rules is illustrated in Fig. 4.

- **C1** (Suspend Task): Each of the following conditions must hold:
  - s_1 is enclosed in a `suspend`(B)-`resume`(B) block with dynamic priority (p, q);
  - there is no task with maximum dynamic priority greater than or equal to p, that can resume B;
  - Either no blocking statement in the `suspend`(B)-`resume`(B) block, or no other task that can resume B.
- **C2** (Priority): Each of the conditions below must hold:
  - The dynamic priorities of s_1 and s_2 are (p_1, q_1) and (p_2, q_2) respectively, with $p_1 > q_2$.
  - There is no thread body with maximum dynamic priority greater than or equal to p_1 that can suspend A.
- **C3** (Flag): Each of the conditions below must hold:
  - s_1 is enclosed in a block F begining with setting the variable `flag` to 1 and ending with resetting it to 0, with dynamic priority of the block being (p_1, q_1).
  - The block F is either in the scope of a `suspendsched` command or there is no thread of priority $\geq p_1$ that resets `flag`.
  - Either there is no blocking command before s_1 in F, or no thread that can reset flag.
  - s_2 is in an `if-then` block which checks that `flag` is not set, with the block having dynamic priority (p_2, q_2).
  - $q_1 < p_2$.
- **C4** (Lock): Each of s_1 and s_2 are within a `lock`(l)-`unlock`(l) block, for some common lock l.

- **C5** (Disable Interrupts): s_1 is within a `disableint-enableint` block.
- **C6** (Suspend Scheduler): s_1 is within a `suspendsched-resumesched` block in a task function, and s_2 is in a task function.

▶ **Theorem 2.** *Let P be an IDA program, and let s_1 and s_2 be statements in P that satisfy one of the conditions (C1) to (C6) above. Then $s_1 \not\triangleleft s_2$ in P.*

Proof. We sketch here a proof of Thm. 2 on the soundness of the conditions C1–C6. Let P be an IDA program with statements s_1 and s_2 satisfying one of the conditions C1–C6. We need to argue that in each case $s_1 \not\triangleleft s_2$. We focus on the first three rules C1–C3 since the remaining are more standard and their soundness is easy to see.

C1: Suppose s_1 and s_2 satisfy the condition C1, and suppose there is an execution ρ of P in which s_2 occurs in between s_1. Let us say s_1 is executed by thread t_1 and s_2 by thread t_2. Then s_2 must happen some time after t_2 was suspended by t_1, and before s_1 takes place. The only way this can happen is if:

- Some thread t_3 with priority *greater than or equal to* p_1 resumes t_2. But this is not possible since the condition says that there is *no* other task with dynamic priority greater than or equal to p_1 which can resume B.
- t_1 makes a blocking call and another task runs and resumes t_2. However this is ruled out by the requirement that [there is no `block` command before s_1] OR [there is no task other than A which can resume B].

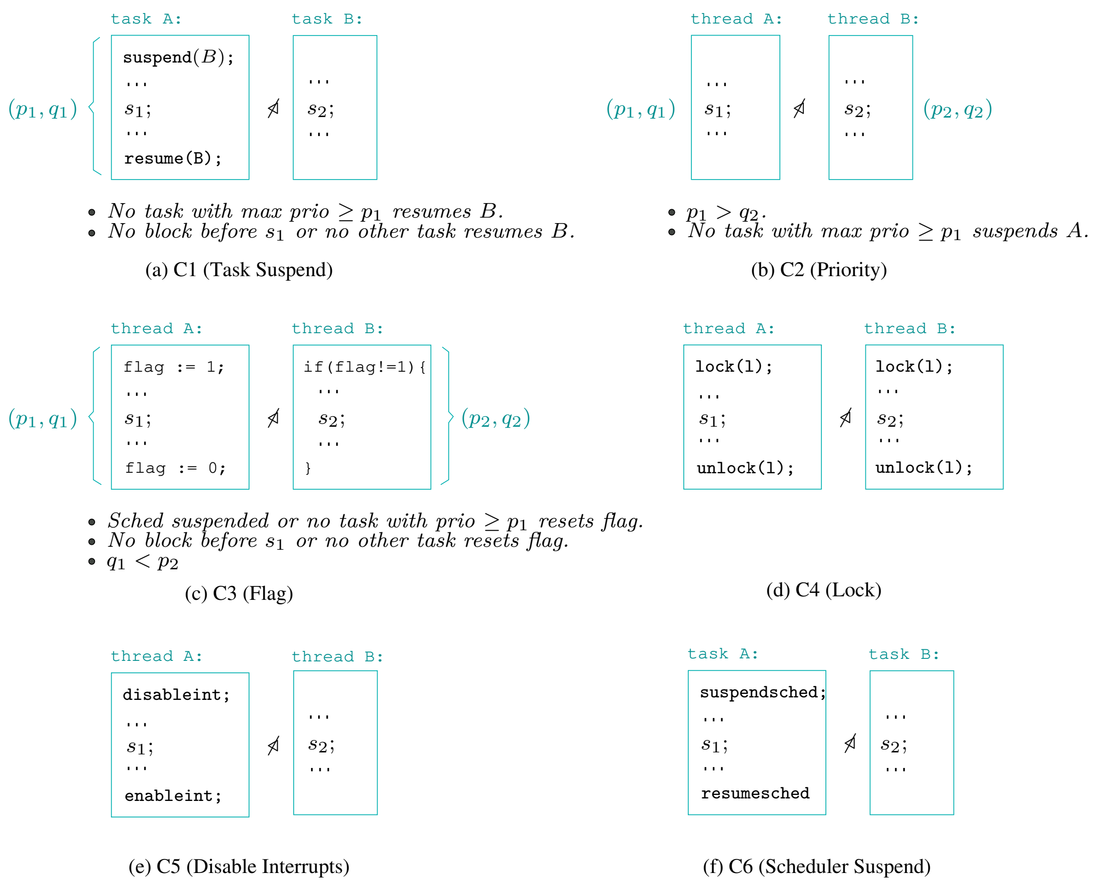
C2: Suppose s_1 and s_2 satisfy the condition C2, and suppose there is an execution ρ of P in which s_2 occurs in between s_1. Let us say s_1 is executed by thread t_1 and s_2 by thread t_2. Then thread t_2 must preempt thread t_1 during the execution of s_1. The only way this can happen is if:

- t_2 with priority *greater than* p_1 was blocked. It runs and preempts t_1. But this is not possible since the condition says that $p_1 >$ maximum dynamic priority of t_2.
- t_2 has a priority *equal to* p_1 and t_1's time slice expires and it gets preempted by t_2. Again, this is not possible since the condition says that $p_1 >$ maximum dynamic priority of t_2.
- Some thread t_3 with priority *greater than or equal to* p_1 was blocked. It runs and suspends t_1. However this is ruled out by the requirement that there is no task other than t_1 with maximum dynamic priority $\geq p_1$, which can suspend t_1.

C3: Suppose s_1 and s_2 satisfy the condition C3, and suppose there is an execution ρ of P in which s_2 occurs in between s_1. Let us say s_1 is executed by thread t_1 and s_2 by thread t_2. Then s_2 must happen some time after $flag_1$ is set to 1 by t_1, and before s_1 takes place. The only way this can happen is if:

- Some thread t_3 with priority *greater than or equal to* p_1 was blocked. It runs and resets $flag_1$ to 0. But this is not possible since the condition says that s_1 is either in the scope of a `suspendsched` command or there is no thread of priority $\geq p_1$ that resets $flag_1$.
- t_1 makes a blocking call and another task runs and resets $flag_1$ to 0. Again, this is not possible because of the requirement that [there is no `block` command before s_1] OR [there is no task other than t_1 which can reset $flag_1$ to 0].
- Both t_1 and t_2 run at the same priority. Before t_1 sets $flag_1$ to 1, t_2 checks $flag_1$ and finds that it is 0, and enters the block containing s_2. Before t_2 executes s_2, it's time slice expires. It gets preempted by t_1 which sets $flag_1$ to 1 and starts s_1. However this is ruled out by the requirement that $p_2 > p_1$.

This completes the argument. ◀



```
task A:                task B:
suspend(B);            ...
...                    s2;
s1;                    ...
...
resume(B);
```

- *No task with max prio $\geq p_1$ resumes B.*
- *No block before s_1 or no other task resumes B.*

(a) C1 (Task Suspend)

```
thread A:              thread B:
...                    ...
s1;                    s2;
...                    ...
```

- $p_1 > q_2$.
- *No task with max prio $\geq p_1$ suspends A.*

(b) C2 (Priority)

```
thread A:              thread B:
flag := 1;             if(flag!=1){
...                      ...
s1;                      s2;
...                      ...
flag := 0;             }
```

- *Sched suspended or no task with prio $\geq p_1$ resets flag.*
- *No block before s_1 or no other task resets flag.*
- $q_1 < p_2$

(c) C3 (Flag)

```
thread A:              thread B:
lock(l);               lock(l);
...                    ...
s1;                    s2;
...                    ...
unlock(l);             unlock(l);
```

(d) C4 (Lock)

```
thread A:              thread B:
disableint;            ...
...                    s2;
s1;                    ...
...
enableint;
```

(e) C5 (Disable Interrupts)

```
task A:                task B:
suspendsched;          ...
...                    s2;
s1;                    ...
...
resumesched
```

(f) C6 (Scheduler Suspend)

Figure 4 Rules that guarantee s_2 cannot occur in between s_1 (i.e. $s_1 \not\triangleleft s_2$).

6 Implementation and Evaluation

We have implemented our analysis for FreeRTOS [5] applications in a tool called RAPR (for "RTOS App Racer"). Our IDA language is closely modelled on the syntax and semantics of FreeRTOS apps, and hence we continue to use the IDA commands in place of the FreeRTOS commands in this section. Our analysis is implemented in the CIL static analysis framework [20] using OCaml, and comprises a few pre-analyses followed by the main analysis for checking the conditions. For convenience we assume that each `create` statement uses a different thread identifier.

Priority Analysis. The priority analysis determines the min and max dynamic priority at each statement in each thread function. This is done in 2 passes as follows. We first perform an interval-based analysis for each function A, maintaining an interval $[p, q]$ of possible priority values at each statement. The initial interval is $[p_0, q_0]$, given by the min and max priorities among threads that run A. The transfer function for a `set_priority`$(\texttt{NULL}, p)$ statement returns the interval $[p, p]$, and is identity for other statements. The join is the standard join on the interval lattice. In the second pass, for each statement `set_priority`(t, p') where t may run A, we update the interval $[p, q]$ at each statement in A to $[\min(p, p'), \max(q, p')]$.

Suspend/Resume Analysis. This analysis determines the set of tasks which can suspend or resume a given task. We maintain a set of task functions *susplist*(A) and *reslist*(A) for each task function A, containing the set of tasks that can suspend and resume A respectively. For each task B with a `suspend`(A) or `resume`(A) statement, we add B to *susplist*(A) or *reslist*(A) respectively.

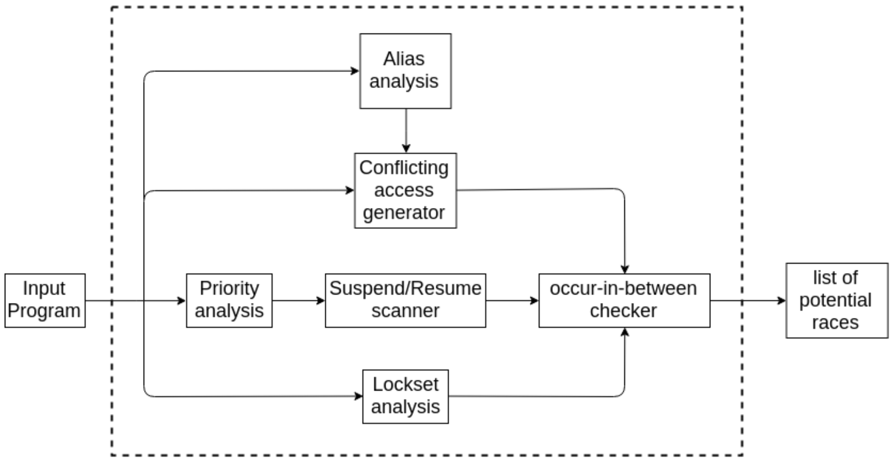


Figure 5 Architecture of RAPR.

Lockset Analysis. A standard lockset analysis aims to compute the set of locks that are *must* held at each program point. On a `lock`(l) statement, the transfer function adds l to the set of locks held after this statement, while for an `unlock`(l) statement we remove l from the set of locks held. The join operation is simply the intersection of the locksets at the incoming points. In our setting, apart from the standard locks, we use notional locks that correspond to the different kinds of code blocks used in the rules of Fig. 4. Thus, for each `suspend`(A)-`resume`(A) block (rule C1) we use a notional lock S_A that we "acquire" at the `suspend`(A) statement and "release" at the `resume`(A) statement. The lockset analysis would then let us identify a `suspend`(A)-`resume`(A) block by the fact that the lock S_A is held throughout these statements. In a similar way we use locks F^{set}_{flag} and F^{chk}_{flag} for each flag variable *flag*, corresponding to the two blocks in rule C3; lock D for `disableint-enableint` (rule C5); and lock S for `suspendsched-resumesched` (rule C6).

Main Analysis. The overall analysis computes conflicting accesses by scanning for global variables having shared accesses in different threads with at least one access being a write access. We use CIL's inbuilt alias analysis to resolve pointers to shared global variables. The information obtained from priority and lockset analysis is used to check for the cannot occur-in-between relation between the pair of conflicting accesses, using rules C1–C6. If both accesses in the pair cannot occur-in-between each other, they are declared to be non-racy; else they are declared to be potentially racy. A schematic of our implementation is shown in Fig 5.

Finally, the analysis allows a couple of command-line switches to handle some of the configuration options of FreeRTOS. Certain locks (called "mutex" locks) have a priority inheritance mechanism associated with them: when a higher priority thread is waiting on a mutex already acquired by a lower priority thread, the lower priority thread has its priority bumped up to the priority of the higher priority thread. Anticipating a need while translating nxtOSEK applications, we also allow a ceiling priority mechanism for mutexes which immediately increases the priority of the acquiring thread to the max priority of all threads that might ever acquire the mutex. To handle this we adapt the transfer function of our priority analysis for a `lock`(l) statement, when l is a mutex, to return $[p, \max(q, p')]$ in the case of priority inheritance, and $[p', p']$ in the case of ceiling priority, where $[p, q]$ is the incoming priority interval and p' is the max priority of any task that might acquire l. We also provide a switch to disallow round-robin scheduling within threads of the same priority, and handle it by modifying the cannot occur-in-between conditions for C1 and C2 by replacing "$>$" by "$\geq$" for the dynamic priorities.

Table 2 Experimental results.

Program	LoC	Conf. acc.	True Races	RAPR Time (in s)	RAPR Pot. Races	RAPR % Elim.	RAPR % Prec.	Pot. Races [23]	Pot. Races [8]
bipedrobot.c	338	3	0	1.39	2	33.33	0.00	2	10
pe_test.c	95	4	3	0.01	3	25.00	100.00	3	7
res_test.c	110	40	8	0.03	9	77.50	88.88	9	11
tt_test.c	105	5	3	0.01	3	40.00	100.00	3	6
usb_test.c	169	0	0	0.02	0	0.00	100.00	0	52
example.c	87	13	1	0.03	1	92.30	100.00	1	61
example_fun.c	102	13	1	0.05	4	69.23	25.00	1	61
pingpong.c	112	3	0	0.01	0	100.00	100.00	0	7
counter.c	96	6	6	0.01	6	0.00	100.00	6	9
dynamic.c	429	20	2	0.13	6	70.00	33.33	16*	23
IntQueue.c	747	42	5	0.97	16	61.90	31.25	10*	24
rangefinder.c	394	16	10	0.23	10	37.50	100.00	16*	18

Experimental Evaluation. We tested our analysis on 12 FreeRTOS applications, shown in Tab. 2. The first 9 are nxtOSEK test programs [6] analysed in [23], which were converted to FreeRTOS programs taking care to preserve the nxtOSEK semantics which these programs use. nxtOSEK uses a priority ceiling protocol for mutex locks and no round-robin scheduling between same priority tasks. The next 2 are demo applications in FreeRTOS and finally, rangefinder.c is the converted version of an ArduPilot subsystem [1] originally in ChibiOS/C++.

The examples used by Schwarz et al. [23] consist of bipedrobot.c which is part of the control software of a self-balancing robot, pe_test.c which tests preemptive scheduling, res_test.c which tests resource based synchronization, tt_test.c where tasks are time-triggered, usb_test.c which tests usb communication, pingpong.c where two tasks set a variable to "ping" and "pong" using a mutex and counter.c where one task increments the fields of a structure and the other task resets and prints these fields. The programs example.c and example_fun.c are examples from [23]. The FreeRTOS demo dynamic.c consists of three tasks which use different mechanisms to access a shared global counter. IntQueue.c is another FreeRTOS demo where tasks share global arrays and counters. Finally rangefinder.c is an ArduPilot subsystem with three task threads and one ISR thread which share access to a state variable and ring and bounce buffers.

Table 2 shows the results of our analysis on these programs. We ran these experiments on a Intel Core i5-8250U 1.60GHz Quad CPU machine under Ubuntu 18.04.4. Conf. acc. denotes the total number of pairs of conflicting accesses to shared global variables in the program. True races denotes the number of actual races existing in the program. RAPR "Pot. Races" denotes the number of conflicting accesses flagged to be potentially racy by the analysis. %Elim. denotes the fraction of conflicting accesses declared to be non-racy and %Prec. denotes the fraction of potential races which are actual races. Pot. Races from [23] and [8] denote the number of potential races flagged using their respective techniques.

In bipedrobot.c, the Task_Init only runs once and hence the potential races are false positives. The decrement to digits in LowTask races with the read and write access in HighTask in pe_test.c. In res_test.c, the read accesses to digits are unprotected due to which it can be an actual race. The decrement of digits in LowTask is unprotected from HighTask and hence it is racy in tt_test.c. In usb_test.c, there are no shared accesses between tasks and hence it is trivially race-free. The races in example.c and example_fun.c are shown in [23]. The ping and pong tasks use a mutex to access the shared variable in pingpong.c and it is race-free. In counter.c, the fields of the global structure are accessed without any protection and hence race with each other. The first initialization of the counter by the controller task in dynamic.c is an actual race with the increment in the continuous increment task because both are created at the same priority and the continuous increment task can preempt the controller when it is initializing the counter. In Intqueue.c some accesses to the shared arrays are real races. In rangefinder.c, the I2C bus thread's access to the state variable is not protected from the main thread and the main thread's access to the ring buffer is not protected from the UART thread which results in a high number of actual races.

The potential races from [23] value is obtained by manually estimating the working of the idea in [23]. This is marked with a * for the last 3 programs as their technique does not handle constructs like dynamically changing the priority of a task and hence is potentially unsound for these programs. It also results in more false positives for the dynamic.c and IntQueue.c examples as protection from synchronization mechanisms like suspending another task, disabling interrupts and suspending the scheduler is not considered. The number of potential races from [8] is obtained using their tool. The tool does not consider priorities for synchronization. Moreover it considers each task function to be run by multiple threads even if only one thread runs it in the application. These factors add to its imprecision.

In dynamic.c the conflicts in the continuous increment task and the limited increment task seem to be racy because they occur at the same priority but the controller task actually ensures that these two can never be in the ready state at the same time keeping one of these two suspended at all times. But this is unknown to the analysis when it encounters the conflicts as this dynamic information about the controller task cannot be made available at these points. This is the reason behind the false positives. The analysis and the test programs with the results can be found in the repository `bitbucket.org/rishi2289/static_race_detect/`.

7 Related Work

We discuss related work grouped according to the three categories below.

Static Race Detection. The most closely related work is that of Schwarz *et al.* [22, 23] and Chopra *et al.* [8]. In [22, 23] Schwarz *et al.* provide a precise interprocedural data-flow analysis for checking races in OSEK-like applications that use the priority ceiling semantics. Chopra *et al.* [8] propose the notion of disjoint-blocks to detect data races and carry out data-flow analysis for FreeRTOS-like interrupt-driven *kernel APIs*. In contrast to both these works, our work handles a comprehensive variety of synchronization mechanisms, including suspend-resume and setting priorities dynamically. In addition we handle dynamic thread creation which both these works do not.

In other work in this category Chen *et al.* [7] develop a static analysis tool for race detection in binaries of interrupt-driven programs with interrupt priorities of upto two levels. The tool considers only disable-enable of interrupts as a synchronization mechanism and does not consider interleavings of task threads. Regehr and Cooprider [21] describe a source-to-source translation of an interrupt-driven program to a standard multi-threaded program, and analyze the translated program for data races. However their translation is inadequate in our setting and we refer the reader to [8] for the inherent problems with such an approach. Sung *et al.* [26] propose a modular technique for static verification of interrupt-driven programs with nesting and priorities. However, the algorithm does not consider interrupt-related synchronization mechanisms nor does it consider interleavings of task threads or interaction with the ISRs. Wang *et al.* [29] present SDRacer, an automated framework that detects and validates race conditions in interrupt-driven embedded software. The tool combines static analysis, symbolic execution, and dynamic simulation. However, it is unsound as their static analysis does not iterate to fixpoint. Mine *et al.* [18] extend Astree by employing a thread-modular static analyzer to soundly report data races in embedded C programs with mutex locks and dynamic priorities. However they do not consider interrupts and synchronization mechanisms like flag-based and suspend-resume. Finally, several papers do lockset-based static analysis for data races in classical concurrent programs [25, 11, 28, 2]. Flanagan *et al.* [12, 13] uses type system to track the lockset at each program point. However none of these techniques apply to interrupt-driven programs with non-standard synchronization mechanisms and switching semantics.

Model-Checking. Several researchers have used model-checking tools like Slam, Blast, and Spin to precisely model various kinds of synchronization mechanisms and detect errors exhaustively [16, 10, 15, 14, 30, 3, 19]. These technique cannot handle dynamic thread creation, and even with a small bound on the number of threads suffer from state-space explosion. Liang *et al.* [17] present an effective method to verify interrupt-driven software with nested interrupts, based on symbolic execution. The method translates a concurrent program into atomic memory read/write *events*, and then describe the interleavings of these events as a symbolic partial order expressed by a SAT/SMT formula. It is able to verify only a bounded number of interrupts.

High-Level Race Detection. A "high-level" race occurs when two blocks of code representing critical accesses overlap in an execution. Our definition of a data race between statements s_1 and s_2 in program P can thus be phrased as a high-level race on the `skip`-blocks in the augmented program P_{s_1,s_2}. Artho *et al.* [4], von Praun and Gross [27], and Pessanha *et al.* [9] study a "view"-based notion of high-level races and carry out lockset based static analysis to detect high-level races. Singh *et al.* [24] use the disjoint-block notion of [8] to detect high-level races in several RTOS *kernels.* They consider some non-standard synchronization mechanisms and also the relative scheduling priorities of specialized threads like callbacks and software interrupts. However none of these techniques handle the full gamut of synchronization mechanisms we address, and hence would be very imprecise for our applications.

8 Conclusions and Future Work

We have presented an efficient and precise way to detect data-races in RTOS applications that use a variety of non-standard synchronization constructs and idioms. Going forward we would like to extend our tool to be able to handle large real-life applications like ArduPilot which are written in C++ and run on the ChibiOS RTOS. We would also like to extend our technique to identify disjoint-block patterns so that we can carry out efficient data-flow analysis [8] for such applications.

References

1 ArduPilot: Open source drone software. versatile, trusted, open. `https://ardupilot.org/`, 2020.

2 Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

3 Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1):167–188, 2000.

4 Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

5 Richard Barry. The FreeRTOS kernel, v10.0.0. `https://freertos.org`, 2017.

6 Takashi C. The NxtOSEK project. `https://sourceforge.net/projects/lejos-osek/`, 2014.

7 Rui Chen, Xiangying Guo, Yonghao Duan, Bin Gu, and Mengfei Yang. Static data race detection for interrupt-driven embedded software. In *Proceedings of the 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion*, SSIRI-C '11, page 47–52, USA, 2011. IEEE Computer Society.

8 Nikita Chopra, Rekha Pai, and Deepak D'Souza. Data races and static analysis for interrupt-driven kernels. In *Proceedings of the 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019*, volume 11423 of *Lecture Notes in Computer Science*, pages 697–723, Prague, Czech Republic, 2019. Springer. `doi:10.1007/978-3-030-17184-1_25`.

9 Ricardo J. Dias, Vasco Pessanha, and João Lourenço. Precise detection of atomicity violations. In *Proceedings of the 8th International Haifa Verification Conference, HVC 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 8–23, Haifa, Israel, 2012. Springer. `doi:10.1007/978-3-642-39611-3_8`.

10 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research, 2005.

11 Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, 37(5):237–252, October 2003.

12 Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000*, pages 219–232, Vancouver, Britith Columbia, Canada, 2000. ACM. `doi:10.1145/349299.349328`.

13 Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE, 2001*, pages 90–96, Snowbird, Utah, USA, 2001. ACM. `doi:10.1145/379605.379687`.

14 Klaus Havelund, Michael R. Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

15 Klaus Havelund and Jens U. Skakkebæk. Applying model checking in java verification. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, page 216–231, Berlin, Heidelberg, 1999. Springer-Verlag. `doi:10.1007/3-540-48234-2_17`.

16 Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 1–13, New York, NY, USA, 2004. Association for Computing Machinery.

17 Lihao Liang, Tom Melham, Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Effective verification for low-level software with competing interrupts. *ACM Transactions on Embedded Computing Systems*, 17(2):36:1–36:26, December 2017.

18 Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016.

19 Suvam Mukherjee, Arun Kumar, and Deepak D'Souza. Detecting all high-level dataraces in an RTOS kernel. In *Proceedings of the 18th International Conference on VMCAI 2017*, volume 10145 of *Lecture Notes in Computer Science*, pages 405–423, Paris, France, 2017. Springer. `doi:10.1007/978-3-319-52234-0_22`.

20 George Necula. CIL – infrastructure for C Program Analysis and Transformation (v. 1.3.7). `http://people.eecs.berkeley.edu/~necula/cil/`, 2002.

21 John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, 2007.

22 Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, and Kalmer Apinis. Precise analysis of value-dependent synchronization in priority scheduled programs. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2014*, volume 8318 of *Lecture Notes in Computer Science*, pages 21–38, San Diego, CA, USA, 2014. Springer. doi:10.1007/978-3-642-54013-4_2.

23 Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 93–104, Austin, TX, USA, 2011. ACM. doi:10.1145/1926385.1926398.

24 Abhishek Singh, Rekha Pai, Deepak D'Souza, and Meenakshi D'Souza. Static analysis for detecting high-level races in RTOS kernels. In *Proceedings of the Formal Methods - The Next 30 Years - Third World Congress, FM 2019*, volume 11800 of *Lecture Notes in Computer Science*, pages 337–353, Porto, Portugal, 2019. Springer. doi:10.1007/978-3-030-30942-8_21.

25 Nicholas Sterling. WARLOCK - A static data race analysis tool. In *Proc. Usenix Winter Technical Conference*, pages 97–106, 1993.

26 Chungha Sung, Markus Kusano, and Chao Wang. Modular verification of interrupt-driven software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 206–216, Urbana, IL, USA, 2017. IEEE Computer Society. doi:10.1109/ASE.2017.8115634.

27 Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

28 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007*, pages 205–214, Dubrovnik, Croatia, 2007. ACM. doi:10.1145/1287624.1287654.

29 Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017*, pages 113–124, Santa Barbara, CA, USA, 2017. ACM. doi:10.1145/3092703.3092724.

30 Reng Zeng, Zhuo Sun, Su Liu, and Xudong He. Mcpatom: A predictive analysis tool for atomicity violation using model checking. In *Proceedings of the 19th International Workshop on Model Checking Software SPIN 2012*, volume 7385 of *Lecture Notes in Computer Science*, pages 191–207, Oxford, UK, 2012. Springer. doi:10.1007/978-3-642-31759-0_14.

A Semantics

$$\frac{c = \texttt{skip}t = rpc(t) = l}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id\rangle}\ \textsc{skip}$$

$$\frac{c = \texttt{skip}t \in \mathcal{R}ISR(t)t \neq rpc(t) = l = ent_{\mathcal{F}(t)}\mathcal{P}(t) > \mathcal{P}(r)id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, id\rangle}\ \textsc{skip-int}$$

$$\frac{c = x := et = rpc(t) = l}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi[x \mapsto [\![e]\!]_\phi], r, i, ss, id\rangle}\ \textsc{assign}$$

$$\frac{c = x := et \in \mathcal{R}ISR(t)t \neq rpc(t) = l = ent_{\mathcal{F}(t)}\mathcal{P}(t) > \mathcal{P}(r)id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi[x \mapsto [\![e]\!]_\phi], t, r, ss, id\rangle}\ \textsc{assign-int}$$

$$\frac{c = \texttt{assume}(b)t = rpc(t) = l[\![b]\!]_\phi = true}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id\rangle}\ \textsc{assume}$$

$$\frac{c = \texttt{assume}(b) \quad t \in \mathcal{R} \quad ISR(t) \quad t \neq r \quad pc(t) = l = ent_{\mathcal{F}(t)} \quad \mathcal{P}(t) > \mathcal{P}(r) \quad [\![b]\!]_\phi = true \quad id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, id\rangle}\ \textsc{assume-int}$$

$$\frac{c = \texttt{create}(A, p, v) \quad t = r \quad task(t) \quad A \in F \quad type(A) = task \quad ts \notin \mathcal{T} \quad (p \leq \mathcal{P}(r) \vee (ss \vee id) = true)}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R} \cup \{ts\}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}[ts \mapsto A], pc[t \mapsto l', ts \mapsto ent_A], \phi[v \mapsto ts], r, i, ss, id\rangle}\ \textsc{create-ns}$$

$$\frac{c = \texttt{create}(A, p, v) \quad t = r \quad task(t) \quad A \in F \quad type(A) = task \quad ts \notin \mathcal{T} \quad p > \mathcal{P}(r) \quad (ss \vee id) = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R} \cup \{ts\}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}[ts \mapsto A], pc[t \mapsto l', ts \mapsto ent_A], \phi[v \mapsto ts], ts, i, ss, id\rangle}\ \textsc{create-cs}$$

$$\frac{c = \texttt{set\_priority}(ts, p) \quad t = r \quad task(t) \quad pc(t) = l \quad p \in \mathbb{N} \quad task(ts) \quad ts \in \mathcal{T} \quad (\ (\mathcal{P}(r) \geq p) \ \vee \ (\mathcal{P}(r) < p \ \wedge \ ts \in (\mathcal{B} \cup \mathcal{S})) \ \vee \ (ss \vee id) = true\ )}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id\rangle}\ \textsc{setp-ns}$$

$$\frac{c = \texttt{set\_priority}(ts, p) \quad t = r \quad task(t) \quad pc(t) = l \quad p \in \mathbb{N} \quad task(ts) \quad ts \in \mathcal{R} \quad p > \mathcal{P}(r) \quad (ss \vee id) = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}[ts \mapsto p], \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, ss, id\rangle}\ \textsc{setp-cs}$$

$$\frac{c = \texttt{suspend}(ts) task(t) t = r \neq ts ts \in \mathcal{T} pc(t) = l}{s \Rightarrow_\iota \langle \mathcal{B} - \{ts\}, \mathcal{S} \cup \{ts\}, \mathcal{R} - \{ts\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id\rangle} \text{ SUS-NS}$$

$$\frac{c = \texttt{suspend}(ts) \;\; task(t) \;\; t = r = ts \;\; pc(t) = l \;\; (ss \vee id) = false \;\; \exists ts' \in \mathcal{R}.task(ts') \;\wedge\; ts' \neq r \;\wedge\; \mathcal{P}(ts') = max(\{\mathcal{P}(u) | u \in \mathcal{R} - \{r\} \;\wedge\; task(u)\})}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S} \cup \{r\}, \mathcal{R} - \{r\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts', i, ss, id\rangle} \text{ SUS-CS}$$

$$\frac{c = \texttt{resume}(ts) \;\; task(t) \;\; t = r \neq ts \;\; pc(t) = l \;\; ts \in (\mathcal{S} \cup \mathcal{R}) \;\; ((ss \;\vee\; id) = true \;\vee\; \mathcal{P}(r) \geq \mathcal{P}(ts))}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S} - \{ts\}, \mathcal{R} \cup \{ts\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id\rangle} \text{ RES-NS}$$

$$\frac{c = \texttt{resume}(ts) \;\; task(t) \;\; t = r \neq ts \;\; pc(t) = l \;\; ts \in \mathcal{S} \;\; (ss \;\vee\; id) = false \;\; \mathcal{P}(ts) > \mathcal{P}(r)}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S} - \{ts\}, \mathcal{R} \cup \{ts\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, ss, id\rangle} \text{ RES-CS}$$

$$\frac{c = \texttt{suspendsched} t = r task(t) pc(t) = l}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, true, id\rangle} \text{ SUSSCH}$$

$$\frac{c = \texttt{resumesched} t = r task(t) pc(t) = l (\forall ts \in \mathcal{R}.task(ts) \wedge \mathcal{P}(r) \geq \mathcal{P}(ts) \;\vee\; id = true)}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, false, id\rangle} \text{ RESSCH-NS}$$

$$\frac{c = \texttt{resumesched} \;\; t = r \;\; task(t) \;\; pc(t) = l \;\; \exists ts \in \mathcal{R}.task(ts) \wedge \mathcal{P}(ts) = max(\{\mathcal{P}(u) | u \in \mathcal{R} \;\wedge\; task(u)\}) \;\; id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, false, id\rangle} \text{ RESSCH-CS}$$

$$\frac{c = \texttt{disableint} t = r pc(t) = l}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, true\rangle} \text{ DISINT}$$

$$\frac{c = \texttt{disableint} t \in \mathcal{R} ISR(t) t \neq r pc(t) = l = ent_{\mathcal{F}(t)} \mathcal{P}(t) > \mathcal{P}(r) id = false}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, true\rangle} \text{ DISINT-INT}$$

$$\frac{c = \texttt{enableint} t = r pc(t) = l (\forall ts \in \mathcal{R}.task(ts) \wedge \mathcal{P}(r) \geq \mathcal{P}(ts) \;\vee\; ss = true \;\vee\; ISR(r))}{s \Rightarrow_\iota \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, false\rangle} \text{ ENINT-NS}$$

$$\frac{c = \texttt{enableint}\ \ t = r\ \ task(t)\ \ pc(t) = l\ \ \exists ts \in \mathcal{R}.task(ts) \wedge \mathcal{P}(ts) = max(\{\mathcal{P}(u) | u \in \mathcal{R} \wedge task(u)\})\ \ ss = false}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, ss, false \rangle}\ \textsc{enint-cs}$$

$$\frac{c = \texttt{enableint} t \in \mathcal{R} ISR(t) t \neq r pc(t) = l = ent_{\mathcal{F}(t)} \mathcal{P}(t) > \mathcal{P}(r) id = false}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, false \rangle}\ \textsc{enint-int}$$

$$\frac{c = \texttt{lock}(m) t = r pc(t) = l (\mathcal{A}(m) = undef\ \vee\ \mathcal{A}(m) = r)}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{A}[m \mapsto r], \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id \rangle}\ \textsc{lock-aq}$$

$$\frac{c = \texttt{lock}(m)\ \ t = r\ \ task(t)\ \ pc(t) = l\ \ \mathcal{A}(m) = ts \neq r\ \ (ss \vee id) = false\ \ \exists ts' \in \mathcal{R}.ts' \neq r\ \wedge\ task(ts')\ \wedge\ \mathcal{P}(ts') = max(\{\mathcal{P}(u) | u \in \mathcal{R} - \{r\} \wedge task(u)\})}{s \Rightarrow_{\iota} \langle \mathcal{B} \cup \{r\}, \mathcal{S}, \mathcal{R} - \{r\}, \mathcal{A}, \mathcal{F}, pc, \phi, ts', i, ss, id \rangle}\ \textsc{lock-cs}$$

$$\frac{c = \texttt{lock}(m)\ \ t \in \mathcal{R}\ \ ISR(t)\ \ t \neq r\ \ pc(t) = ent_{\mathcal{F}(t)}\ \ \mathcal{P}(t) > \mathcal{P}(r)\ \ \mathcal{A}(m) = undef\ \ id = false}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}[m \mapsto t], \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, id \rangle}\ \textsc{lock-aq-int}$$

$$\frac{c = \texttt{unlock}(m) t = r pc(t) = l (\mathcal{A}(m) = r \vee \mathcal{A}(m) = undef)}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{A}[m \mapsto undef], \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id \rangle}\ \textsc{unlock}$$

$$\frac{c = \texttt{unlock}(m) t \in \mathcal{R} ISR(t) t \neq r pc(t) = l = ent_{\mathcal{F}(t)} \mathcal{P}(t) > \mathcal{P}(r) \mathcal{A}(m) \neq t id = false}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, t, r, ss, id \rangle}\ \textsc{unlock-int}$$

$$\frac{c = \texttt{block} t = r task(t) pc(t) = l (ss \vee id) = true}{s \Rightarrow_{\iota} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, r, i, ss, id \rangle}\ \textsc{blk-ns}$$

$$\frac{c = \texttt{block}\ \ t = r\ \ task(t)\ \ pc(t) = l\ \ (ss \vee id) = false\ \ \exists ts \in \mathcal{R}.task(ts)\ \wedge\ ts \neq r\ \wedge\ \mathcal{P}(ts) = max(\{\mathcal{P}(u) | u \in \mathcal{R} - \{r\}\ \wedge\ task(u)\})}{s \Rightarrow_{\iota} \langle \mathcal{B} \cup \{r\}, \mathcal{S}, \mathcal{R} - \{r\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, ss, id \rangle}\ \textsc{blk-cs}$$

$$\frac{c = \texttt{start} \;\; t = r = 0 \;\; (ss \vee id) = \mathit{false} \;\; \exists ts \in (\mathcal{S} \cup \mathcal{R}).task(ts) \;\wedge\; \mathcal{P}(ts) = max(\{\mathcal{P}(u) | u \in \mathcal{S} \cup \mathcal{R} \;\wedge\; task(u)\})}{s \Rightarrow_{\iota} \langle \mathcal{B}, \emptyset, \mathcal{S} \cup \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc[t \mapsto l'], \phi, ts, i, \mathit{false}, \mathit{false} \rangle} \;\textsc{start}$$

$$\frac{t \in \mathcal{B} task(r)((ss \vee id) = \mathit{true} \;\vee\; \mathcal{P}(t) \leq \mathcal{P}(r))}{s \Rightarrow_{*} \langle \mathcal{B} - \{t\}, \mathcal{S}, \mathcal{R} \cup \{t\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc, \phi, r, i, ss, id \rangle} \;\textsc{unblk-ns}$$

$$\frac{t \in \mathcal{B} task(r)(ss \vee id) = \mathit{false} \mathcal{P}(t) > \mathcal{P}(r)}{s \Rightarrow_{*} \langle \mathcal{B} - \{t\}, \mathcal{S}, \mathcal{R} \cup \{t\}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc, \phi, t, i, ss, id \rangle} \;\textsc{unblk-cs}$$

$$\frac{t \in \mathcal{R} task(t) t \neq r (ss \vee id) = \mathit{false} \mathcal{P}(t) = \mathcal{P}(r)}{s \Rightarrow_{*} \langle \mathcal{B}, \mathcal{S}, \mathcal{R}, \mathcal{P}, \mathcal{A}, \mathcal{F}, pc, \phi, t, i, ss, id \rangle} \;\textsc{tshare}$$

For the commands `skip`, `x:=e`, `assume`, `disableint`, `enableint`, `lock`, and `unlock` permitted in an ISR thread, the following constraints need to hold on s'. If the current ISR thread is executing the last statement then r' is the highest priority ISR which was interrupted, if there exists one, and $i' = i$. If no ISRs were interrupted then $r' = i$, the interrupted task thread and $i' = main$, a default value. Also, $pc'(t) = ent_{\mathcal{F}(t)}$.