Static Race Detection for Periodic Programs

Varsha P Suresh
IIITB
Bangalore, India
varsha.suresh@iiitb.ac.in

Deepak D'Souza
IISc
Bangalore, India
deepakd@iisc.ac.in

Rekha Pai
IISc
Bangalore, India
rekhapai@iisc.ac.in

Meenakshi D'Souza
IIITB
Bangalore, India
meenakshi@iiitb.ac.in

Sujit Kumar Chakrabarti
IIITB
Bangalore, India
sujitkc@iiitb.ac.in

***Abstract*—We consider the problem of statically detecting data races in periodic real-time programs that use locks, and run on a single processor platform. We propose a technique based on a small set of rules that exploits the priority, periodicity, locking, and timing information of tasks in the program. One of the key requirements is a response time analysis for such programs, and we propose an algorithm to compute this for the case of non-nested locks. We have implemented our analysis for real-time programs written in C in a tool called PEPRACER and evaluated its performance on a small set of benchmarks from the literature.**

***Index Terms*—Real-Time systems, periodic programs, static analysis, data races, WCRT Analysis**

I. INTRODUCTION

Periodic real-time applications (or simply periodic programs) are a class of real-time systems that comprise a set of tasks, each of which comes with an associated priority and periodicity, and are executed according to a scheduling policy like priority-based preemptive scheduling, on a real-time operating system. Many of these systems are safety-critical in nature, being widely employed in avionics, robotics, and autonomous systems.

These systems are also essentially *concurrent* in nature (even if we consider single processor platforms), since a running task may be preempted by a higher priority task, causing them to interleave in time. With concurrency come the attendant problems of data-races: it is not difficult to imagine a scenario where a low priority task is updating a shared data-structure or even a multi-word variable like a `long int`, when it is preempted by a higher priority task that goes on to access the potentially inconsistent shared data. Thus it is common for real-time application developers to use synchronization mechanisms like locks to protect accesses to shared data structures or resources (like an LCD display). Real-Time operating systems typically provide a variety of lock mechanisms from standard locks or semaphores to priority-inheritance based locks [1].

Our focus in this paper is on giving a way to statically (that is by analyzing the source code of the application, rather than running it) detect races in periodic programs that use standard locks. The emphasis in static analysis techniques is on *soundness*: we do not eliminate a pair of conflicting accesses unless we can prove that they do not race. The other side of the coin is *precision*: how close is the set of potential races reported to the actual set of races in the program. The basic technique used in the programming languages community to statically detect races is a *lockset* analysis, which computes the set of locks that are must-held at each statement in a task, and declares two statements to be non-racy if they hold a common lock. More recent techniques [2], [3] exploit priority information to declare accesses to be non-racy: for instance a high-priority task does not need to protect its accesses from lower priority tasks.

However, none of these techniques seek to exploit the inherent periodic nature or execution times of the tasks in these programs. For example, a simple observation is that if two tasks have the same period and don't take any locks, they can never overlap in time. Exploiting timing information is also key to improving the precision of a race analysis technique for these programs. The notion of worst-case response time (WCRT) of a task measures the maximum time an instance of the task may take to complete its execution starting from the beginning of its period. As an example of how we can use conservative WCRT estimates, if we can conclude from the WCRT information that a low-priority task always finishes execution *before* the next arrival of a high-priority task, we can declare them to be non-racy.

While computing the WCRT of tasks in periodic programs is well-studied in the real-time systems community, starting from [4], [5] for periodic programs without locks, and for periodic programs with priority-inheritance-based locks [1], as far as we are aware there are no techniques available for periodic programs with *standard* locks. One of the contributions of this paper is to extend the classical technique of [5] to compute WCRT estimates for programs with *non-nested* locks, given worst-case execution time (WCET) estimates of tasks and `lock-unlock` blocks (or critical sections).

We then go on to give a set of six rules (in the spirit of the ideas described above) to soundly eliminate pairs of conflicting accesses, leading to a sound, efficient, and fairly precise race-detection technique for such programs.

We have implemented our analysis in a tool called PEPRACER for detecting races in such programs written in C. One of the inputs to the tool is a WCET analysis for different blocks in the program tasks, which we obtain using the WCET analysis tool Heptane [6]. We have run our tool on several benchmarks, including robot controllers from the nxtOSEK project [7]. Our tool runs in a fraction of a second on these benchmarks, and on the average eliminates 97% of conflicting access pairs as non-racy.

An overview of our technique is presented in the next

Section on a benchmark example. Periodic programs and their execution semantics are given in Section III. Section IV formally defines the notions of conflicting accesses and data races. Algorithms for computing safe bounds on response times of periodic programs with locks are presented in Section V-B. Section VI gives the rules for disjointedness of tasks and the race detection algorithm for periodic programs. Our experiments on benchmark examples are detailed in Section VII, followed by conclusion.

II. Overview

We provide an overview of our technique with an illustrative example adapted from the “lego_osek” robot controller, based on the OSEK operating system, from [7]. Fig. 1 shows some excerpts from this example. The controller’s job is to control the motion of the two-wheeled robot to follow a line (that it detects using light sensors) but also to detect obstacles along the way (using a sonar sensor) and avoid them by braking and moving to the left. The controller has two tasks `TaskControl` and `TaskObstAvoid` that do the line-following control and obstacle detection and avoidance respectively. `TaskControl` has high priority (higher value indicates higher priority) and runs every 10ms, while `TaskObstAvoid` has low priority and runs every 30ms. The two tasks access some shared locations, including structures for actuating the left and right wheel motors, an LCD display, and a Boolean “obstacle-detected” flag. `TaskControl` reads two light sensor values, does some computation with them, and writes them to the LCD display. The access to the LCD display is protected by acquiring and releasing the `lcd_lock` lock. Finally it computes the new speed and brake values that are then written to the wheel motor structures, after checking that the `obstacle` flag is not set. The `TaskObstAvoid` task reads the sonar and left light sensors, does some computation on them, sets the `obstacle` flag based on these values, and displays them on the LCD (making sure to take a lock on it first). If the `obstacle` flag was set, it goes on to write to the left wheel structure to brake and turn the robot to the left.

We note that there are several conflicting accesses to the shared variables, including lines 13 and 33 to `lcd`, lines 16 and 29 and 16 and 31 on `obstacle`, and lines 19–20 and 36–37 on `left_wheel`. Apart from the accesses to `lcd` which are protected by a lock, the other accesses appear to be racy at first glance. For instance, while `TaskObstAvoid` is updating the left wheel structure, it could be preempted by the higher priority `TaskControl` which goes on to write to the same structure, potentially leading to a harmful race.

Our key idea is to exploit the priority, periodicity, and worst case response times of the tasks, to show that these accesses cannot race. Fig. 2 shows the periodic execution of the two tasks. Notice that if the low priority task is guaranteed to finish its execution before the next instance of the higher priority task is scheduled, there can be no interleaving of the two tasks, and we can declare all the conflicting accesses as non-racy. However, concluding this in the presence of locks is not easy, and our first contribution is a way of computing an estimate

```
 1. // Shared structures and variables
 2. struct motor right_wheel;
 3. struct motor left_wheel;
 4. struct display lcd;
 5. bool obstacle;

 6. void TaskControl() { // Period 10, Priority 2 (high)
 7.   int sensor_right, sensor_left;
 8.   // Read and calibrate sensor values
 9.   sensor_right = get_light_sensor(right);
10.   sensor_left = get_light_sensor(left);
11.   // display sensor values in LCD
12.   lock(lcd_lock);
13.   show_var(sensor_right, sensor_left); // writes to lcd;
14.   unlock(lcd_lock);
15.   // PWM-based motor control, uses sensor values
16.   if (!obstacle) {
17.     right_wheel.speed = ...;
18.     right_wheel.brake = 0;
19.     left_wheel.speed = ...;
20.     left_wheel.brake = 0;
21.   }
22. }

23. void TaskObstAvoid() { // Period 30, Priority 1 (low)
24.   int sonar_value, sensor_left;
25.   // Read and calibrate sensor values
26.   sonar_value = get_sonar_sensor(sonar_sensor);
27.   sensor_left = get_light_sensor(left);
28.   if (...)
29.     obstacle = 1;
30.   else
31.     obstacle = 0;
32.   lock(lcd_lock);
33.   show_var(sonar_value, sensor_left); // writes to lcd
34.   unlock(lcd_lock);
35.   if (obstacle) { // avoid by moving left
36.     left_wheel.speed = ...;
37.     left_wheel.brake = 1;
38.   }
39. }
```

Fig. 1: An example periodic program adapted from Lego-OSEK

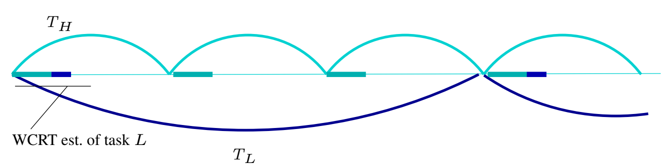


Fig. 2: Task timelines for Lego-OSEK example

of the worst case response times for tasks that take non-nested locks (like in the example program). Using raw WCET times of the tasks and its lock blocks (like lines 12–14) for the platform the robot controller is to be run on, we use Algo. 2 (described in Sec. V) to compute an estimate of the response time of `TaskObstAvoid`. Rule 3 (described in Sec. VI) then allows us to eliminate all the pairs of conflicting accesses as non-racy.

We note that techniques such as [2], [3] that consider task priorities and locks (but *not* periodicities and response times) would not be able to eliminate any of the conflicting acccess pairs, except the accesses to `lcd` which are protected by a lock.

TABLE I: Periodic Program Commands Cmd_V

Statement	Description
`start`	Make all the tasks ready for execution.
`begin`	Start executing the task.
`end`	Finish executing the task.
`skip`	Do nothing.
$x := e$	Assign the value of expression e to x.
`assume`(b)	Enabled only if expression b evaluates to *true*; does nothing.
`lock`(l)	Current task takes lock l if available; otherwise blocks till l becomes available.
`unlock`(l)	Current task releases lock l.

III. PERIODIC PROGRAMS

A *periodic program* is a collection of *tasks*. Each task has an associated *function*, *period*, and *priority*. There is a designated *init* task which is the only task that is ready to run initially. An execution of the program begins with running the $init_f$ function, associated with the *init* task, which initializes shared variables and then makes other tasks ready to run using the `start` command. The *init* task runs only once.

The execution of the tasks is orchestrated by a priority-based preemptive scheduler. It is important to point out here that we are assuming a *single processor* platform. The scheduler selects one of the enabled tasks for execution on a highest-priority-first basis. A task with period T is enabled every T time units. If there are more than one tasks of the highest priority ready to run, the longest waiting task is picked for execution. This is also known as First-Come-First-Served or FCFS scheduling.

The task functions operate on a set of shared variables V using assignment statements and accesses to the shared variables can be synchronized using the `lock`-`unlock` commands. The commands Cmd_V used in a periodic program are shown in Table I.

Formally, a *periodic program* $\mathcal{P}$ is a tuple $(V, L, \mathcal{T})$ where V is a finite set of shared variables, L is a finite set of locks, and $\mathcal{T} = \{\tau_1, \ldots, \tau_k\}$ is a finite set of tasks. A *task* $\tau \in \mathcal{T}$ is a tuple (G_τ, T_τ, p_τ), where G_τ is the task function, T_τ is the period between two invocations of the task, and p_τ is its priority. The task function G_τ is represented as a Control Flow Graph (CFG) $G_\tau = (Loc_\tau, I_\tau, ent_\tau, ext_\tau)$, where Loc_τ is the finite set of locations of G_τ, $I_\tau \subseteq Loc_\tau \times Cmd_V \times Loc_\tau$ is the set of instructions in G_τ, and $ent_\tau, ext_\tau \in Loc_\tau$ are the entry and exit locations respectively in G_τ . Sets $Loc_\mathcal{P}$ and $I_\mathcal{P}$ are the locations and instructions, respectively, in program $\mathcal{P}$. We will drop the subscripts in the notations whenever the context is clear.

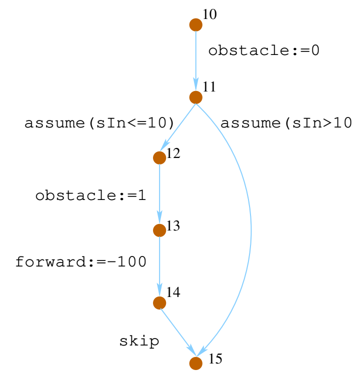
An example periodic program and the CFG representation of the `ObsDect` function are shown in Fig. 3. The periodic program has two tasks that implements a simple robotic controller, apart from the default *init* task. The `ObsDect` task function detects an obstacle based on the sensor input in the ***sIn*** variable and makes a corrective action. The `MoveForward` task function directs the robot to move forward if there is no obstacle. The `ObsDect` task has high priority and runs frequently at every 100 time units while the `MoveForward` task has low priority and runs every 200 time units. Both the tasks access the shared variables *obstacle* and *forward*. We use the convention that a higher value indicates higher priority.

```
init:
1.  obstacle := 0;
2.  forward := 0;
3.  sIn := 0;
4.  start;

// Period = 100, Prio = 2
ObsDect:
10.  obstacle := 0;
11.  if (sIn <= 10) {
12.    obstacle := 1;
13.    forward := -100;
14.  }
15.

// Period = 200, Prio = 1
MoveForward:
20.  if (!obstacle)
21.    forward := 100;
22.
```

(a) An example program



(b) CFG of the `ObsDect` function

Fig. 3: Example program and the CFG representation

We now define the semantics of a periodic program $\mathcal{P} = (V, L, \mathcal{T})$ as a labeled transition system $\mathcal{S}_\mathcal{P} = \langle S, s_{in}, \Rightarrow \rangle$ where S is the set of states, $s_{in} \in S$ is the initial state, and $\Rightarrow$ is the transition relation, as defined below. In the following, $\mathcal{T}_q$ denotes a set of task priority queues and ϵ denotes an empty queue. We also assume that the tasks can have priorities in $P = \{1, \ldots, k\}$. For an integer expression e, Boolean expression b, and an environment ϕ for V, we denote by $[\![e]\!]_\phi$ the integer value that e evaluates to in ϕ, and $[\![b]\!]_\phi$ denotes the Boolean value that b evaluates to in ϕ. For a function $f : X \rightarrow Y$, and elements $x \in X$ and $y \in Y$, we use the notation $f[x \mapsto y]$ to denote the function $f' : X \rightarrow Y$ given by $f'(x) = y$ and for all z different from x, $f'(z) = f(z)$.

A state $s \in S$ is a tuple $(\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc, \phi, tick, r)$ where

1) $\mathcal{R}$ is a priority queue of tasks that are ready to run,
2) $\mathcal{W} \subseteq \mathcal{T}$ is the set of tasks that are waiting to be scheduled,
3) $\mathcal{A} \in L \rightharpoonup \mathcal{T}$ is a partial map that gives, for each lock, the task that has acquired the lock,
4) $\mathcal{B} \in L \rightarrow \mathcal{T}_q$ is a map that gives, for each lock, the priority queue of tasks that are blocked on the lock,
5) $pc \in \mathcal{T} \rightarrow Loc_\mathcal{P}$ gives the current location of the task,
6) $\phi \in V \rightarrow \mathbb{Z}$ is a variable to value map,
7) $tick \in \mathbb{Z}^+$ is the time units elapsed since the program started, and
8) $r \in \mathcal{T}$ is the currently running task.

The initial state $s_{in} = (\epsilon, \{\tau_2, \ldots, \tau_k\}, \emptyset, \emptyset, \lambda\tau.ent_{f_\tau}, \lambda x.0, 0, init)$ denotes that initially the *init* task (which is τ_1) is the running task while no other tasks are ready to run and instead are waiting to be scheduled, none of the tasks have acquired locks and hence they are not blocked, all the tasks are at their entry locations, all the variables are initialized to zero, and so is the

$tick$ counter.

We now define the transition relation $\Rightarrow \subseteq S \times I_{\mathcal{P}} \times S$ as follows. For a state $s = (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc, \phi, tick, r)$, a task τ, and an instruction $\iota = (l, c, l')$ in G_τ, we have $s \Rightarrow_\iota s'$ iff one of the rules in Fig. 4 is satisfied. If for a command c, the conditions on state s specified in the antecedent (the ones mentioned above the line) holds then $s \Rightarrow_\iota s'$ in the consequent (the one below the line) also holds.

The $START$ rule, for the `start` command executed by the $init$ task, enqueues all the tasks in $\mathcal{W}$ that are waiting to be scheduled onto the ready queue $\mathcal{R}$. This might require a rescheduling of tasks. Hence the current running task r is also enqueued onto $\mathcal{R}$ but ahead of those tasks with the same priority as that of r. It then picks the highest priority task, which is at the head of the updated ready queue, to be the next running task.

The rule uses the $\texttt{ENQ}(Q, S)$ function which when given a priority queue Q of tasks and a set S of tasks, enqueues each task in S onto the queue Q. The function $\texttt{enqb}(Q, s)$ adds the task s onto to the priority queue Q ahead of the elements with the same priority as that of s. The function $\texttt{deq}(Q)$ returns the queue with the head element removed. The function $\texttt{head}(Q)$ when given a priority queue Q of tasks returns the task with the highest priority, which is at the head of Q.

The END rule is defined for the `end` command to signal completion of the running task. Hence the task is inserted into the wait list $\mathcal{W}$. Moreover, the highest priority task in the ready queue $\mathcal{R}$, which is at its head, is removed from $\mathcal{R}$ and made the running task. The rule requires that the ready queue $\mathcal{R}$ be non-empty.

The ALOCK rule is defined for the $\texttt{lock}(m)$ command. If the running task r requests for a lock m which is not acquired by any task (as given by $\mathcal{A}(m) = undef$) then the running task proceeds with acquiring the lock. The BLOCK rule is defined for the $\texttt{lock}(m)$ command when the running task cannot acquire the lock. If the running task r requests for a lock m which is acquired by a task τ' (as given by $\mathcal{A}(m) = \tau'$) then the running task r is blocked by enqueuing it onto the blocked queue $\mathcal{B}(m)$. Further, the highest priority task from the non-empty ready queue $\mathcal{R}$ is made the running task. The $\texttt{enq}(Q, e)$ function, used in the rule, when given a priority queue Q of tasks enqueues e onto Q, as is standard in the literature.

The UNLOCK rule is defined for the $\texttt{unlock}(m)$ command. If the running task requests for the release of the lock m which it was holding or it was the case that no task was holding the lock (as given by $\mathcal{A}(m) = \tau \vee \mathcal{A}(m) = undef$) then the running task can proceed with releasing the lock. Further, if there are no tasks blocked on this lock m (as given by $\mathcal{B}(m) = \epsilon$) then the current task continues to be the running task. The UNLOCK-WK rule is defined for the $\texttt{unlock}(m)$ command when a low priority task is blocked on the lock. If the running task requests for the release of the lock m which it was holding and a task τ', at the head of the blocked priority queue $\mathcal{B}(m)$, is blocked on the lock, of priority lower than the running task, then τ' is unblocked by dequeing it from its blocked priority queue $\mathcal{B}(m)$ and enqueuing it onto the ready queue $\mathcal{R}$. Here, the task τ' is inserted onto the priority based ready queue $\mathcal{R}$ ahead of the other tasks with the same priority as that of τ', using the function `enqb`. Task r continues to be the running task. The UNLOCK-CS rule is defined for the $\texttt{unlock}(m)$ command when a high priority task is blocked on lock m. If the running task requests for the release of the lock m which it was holding and a high priority task τ' is blocked on the lock then τ' is unblocked by dequeing it from its blocked queue $\mathcal{B}(m)$. The task τ', being of higher priority, is selected as the next running task while the current running task r is enqueued onto the ready queue $\mathcal{R}$.

The SCHED rule is selected non-deterministically. If upon incrementing the $tick$ counter, by one, in the current state s some of the tasks, in $\mathcal{W}$, that are waiting to be scheduled may become ready to be scheduled, due to their periods. In that case such tasks are moved to the ready queue $\mathcal{R}$, while removing them from the wait list $\mathcal{W}$. This might trigger a rescheduling of tasks which is essentially to pick a new high priority task as the next running task. Hence the current running task r is also enqueued onto the ready queue, ahead of those tasks with the priority same as that of r. The highest priority task in the updated ready queue, which is at its head, is selected as the next running task.

The SKIP, BEGIN, ASSIGN, and ASSUME rules for the `skip`, `begin`, assignment statement, and `assume` command, respectively are easy to understand.

An *execution* of a periodic program $\mathcal{P}$ is a finite sequence of transitions $\rho = \delta_1, \ldots, \delta_n$ $(n \geq 1)$, such that there exists a sequence of states $s_0, \ldots, s_n$ of S, with each $\delta_i \in \Rightarrow$ of the form (s_{i-1}, ι_i, s_i) for some ι_i, and $s_0 = s_{in}$.

The semantics we have defined so far abstracts away the "real-time" aspect of a periodic program. We can obtain the real-time semantics of a periodic program by considering a concrete execution environment which fixes the execution time of each instruction (say in a bounded interval of time), and restricting ourselves to executions where the tick interrupt is driven by a real-time clock and is consistent with the time taken to execute instructions between two ticks. Henceforth we fix such an environment and focus on the induced subset of executions of a periodic program.

IV. Data Races

Let $\mathcal{P} = (V, L, \mathcal{T})$ be a periodic program. In an execution of $\mathcal{P}$, tasks are executed periodically and hence during the course of execution of $\mathcal{P}$ several instances of a task gets executed.

Consider two tasks τ_1 and τ_2 in $\mathcal{T}$, and two non-empty paths π and π' in G_{τ_1} and G_{τ_2}, respectively. We say π and π' *may happen in parallel* in $\mathcal{P}$ if there is an execution ρ of $\mathcal{P}$, and instances of τ_1 and τ_2 in ρ, in which the paths π and π' interleave (that is, either π' begins after π has begun but not yet ended; or vice-versa).

We now define when two statements s_1 and s_2 (corresponding, to instructions $\iota_1 = (l_1, c_1, l_1')$ and $\iota_2 = (l_2, c_2, l_2')$) in tasks τ_1 and τ_2, respectively, may happen in parallel. Consider the program $\mathcal{P}'$ obtained from $\mathcal{P}$ by enclosing the statements

$$\frac{c = \texttt{skip} \quad pc(\tau) = l \quad \tau = r}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, r)} \text{ SKIP} \qquad \frac{c = \texttt{begin} \quad pc(\tau) = l \quad \tau = r}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, r)} \text{ BEGIN}$$

$$\frac{c = x := e \quad pc(\tau) = l \quad \tau = r}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi[x \mapsto [\![e]\!]_\phi], tick, r)} \text{ ASSIGN} \qquad \frac{c = \texttt{assume}(b) \quad pc(\tau) = l \quad \tau = r \quad [\![b]\!]_\phi = \textit{true}}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, r)} \text{ ASSUME}$$

$$\frac{c = \texttt{start} \quad pc(\tau) = l \quad \tau = r = init}{s \Rightarrow_\iota (\texttt{deq}(\texttt{ENQ}(\texttt{enqb}(\mathcal{R}, r), \mathcal{W})), \emptyset, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, \texttt{head}(\texttt{ENQ}(\texttt{enqb}(\mathcal{R}, r), \mathcal{W})))} \text{ START}$$

$$\frac{c = \texttt{end} \quad pc(\tau) = l \quad \tau = r \quad \mathcal{R} \neq \epsilon}{s \Rightarrow_\iota (\texttt{deq}(\mathcal{R}), \mathcal{W} \cup \{r\}, \mathcal{A}, \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, \texttt{head}(\mathcal{R}))} \text{ END}$$

$$\frac{c = \texttt{lock}(m) \quad pc(\tau) = l \quad \tau = r \quad \mathcal{A}(m) = undef}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}[m \mapsto \tau], \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, r)} \text{ ALOCK}$$

$$\frac{c = \texttt{lock}(m) \quad pc(\tau) = l \quad \tau = r \quad \mathcal{A}(m) = \tau' \quad \mathcal{R} \neq \epsilon}{s \Rightarrow_\iota (\texttt{deq}(\mathcal{R}), \mathcal{W}, \mathcal{A}, \mathcal{B}[m \mapsto \texttt{enq}(\mathcal{B}(m), r)], pc, \phi, tick, \texttt{head}(\mathcal{R}))} \text{ BLOCK}$$

$$\frac{c = \texttt{unlock}(m) \quad pc(\tau) = l \quad \tau = r \quad (\mathcal{A}(m) = \tau \vee \mathcal{A}(m) = undef) \quad \mathcal{B}(m) = \epsilon}{s \Rightarrow_\iota (\mathcal{R}, \mathcal{W}, \mathcal{A}[m \mapsto undef], \mathcal{B}, pc[\tau \mapsto l'], \phi, tick, r)} \text{ UNLOCK}$$

$$\frac{c = \texttt{unlock}(m) \quad pc(\tau) = l \quad \tau = r \quad \mathcal{A}(m) = r \quad Q = \mathcal{B}(m) \neq \epsilon \quad \texttt{head}(Q) = \tau' \quad p_{\tau'} \leq p_r}{s \Rightarrow_\iota (\texttt{enqb}(\mathcal{R}, \tau'), \mathcal{W}, \mathcal{A}[m \mapsto undef], \mathcal{B}[m \mapsto \texttt{deq}(Q)], pc[\tau \mapsto l'], \phi, tick, r)} \text{ UNLOCK-WK}$$

$$\frac{c = \texttt{unlock}(m) \quad pc(\tau) = l \quad \tau = r \quad \mathcal{A}(m) = r \quad Q = \mathcal{B}(m) \neq \epsilon \quad \texttt{head}(Q) = \tau' \quad p_{\tau'} > p_r}{s \Rightarrow_\iota (\texttt{enqb}(\mathcal{R}, r), \mathcal{W}, \mathcal{A}[m \mapsto undef], \mathcal{B}[m \mapsto \texttt{deq}(Q)], pc[\tau \mapsto l'], \phi, tick, \tau')} \text{ UNLOCK-CS}$$

$$\frac{v = \texttt{inc}(tick) \quad S = \{s \in \mathcal{W} \mid n \in \mathbb{N} \ \wedge \ n.T_s = v\}}{s \Rightarrow_* (\texttt{deq}(\texttt{ENQ}(\texttt{enqb}(\mathcal{R}, r), S)), \mathcal{W} \setminus S, \mathcal{A}, \mathcal{B}, pc, \phi, v, \texttt{head}(\texttt{ENQ}(\texttt{enqb}(\mathcal{R}, r), S)))} \text{ SCHED}$$

Fig. 4: Transition relation capturing the execution semantics of a periodic program

s_1 and s_2 in `skip` statements. Formally, we obtain $\mathcal{P}'$ by replacing the instruction ι_1 by the instructions $(l_1, \texttt{skip}, m_1)$, (m_1, c_1, m_1'), and $(m_1', \texttt{skip}, l_1')$, where m_1 and m_1' are new locations in Loc_{τ_1}; and similarly for ι_2. Let π_1 be the path $l_1 \overset{\texttt{skip}}{\to} m_1 \overset{c_1}{\to} m_1' \overset{\texttt{skip}}{\to} l_1'$ in $G_{\tau_1'}$, and similarly π_2 in $G_{\tau_2'}$. We now say s_1 and s_2 *may happen in parallel* in $\mathcal{P}$, if the paths π_1 and π_2 may happen in parallel in the program $\mathcal{P}'$.

Two statements are called *conflicting* if they are read/write accesses to the same variable, and at least one of them is a write. We say two statements s_1 and s_2 in $\mathcal{P}$ are involved in a *data race* (or are simply *racy*) if they are conflicting accesses that may happen in parallel.

Finally, we define what it means for a "block" of code to happen in parallel with another. A *block* of code in $\mathcal{P}$ is specified by a pair (l, X), where for some task τ in $\mathcal{P}$, l is a location in Loc_τ and $X \subseteq Loc_\tau$ is a subset of locations reachable from l, in task τ. An *initial path* in a block $B = (l, X)$ of a task τ in $\mathcal{P}$, is a non-empty path in G_τ that begins at l and stays within the set of locations X, except possibly for the last location in the path. We say a statement $s = (m, c, m')$ in $\mathcal{P}$ *belongs to* block $B = (l, X)$ if m belongs to the set X. We say two blocks B_1 and B_2 of $\mathcal{P}$ *may happen in parallel* if there are two initial paths π_1 in B_1 and π_2 in B_2, which may happen in parallel with each other. Otherwise, B_1 and B_2 are *disjoint*.

V. Response Time and its Computation

Our aim in this section is to give a way of computing a safe bound on the response time of tasks in a periodic program with locks. We begin by recalling some of the basic notions.

Consider a sequential piece of compiled code B executing on a given hardware platform. Assume that the code does not have to compete for the processor time with other processes (in particular there is no preemption, and lock statements succeed without blocking). The execution time of B may still vary depending on reads of input and other shared locations, which are assumed to return non-deterministic values during the execution. If we consider the supremum of these execution times we obtain the *worst-case execution time* (WCET) of B on the given platform. There are many static analysis techniques and tools that help us obtain conservative estimates on the WCET of a program on a given platform. We refer the reader to [8] for a survey of these techniques and tools.

Let us now consider a periodic program $\mathcal{P} = (V, L, \mathcal{T})$ which we want to execute in a given execution environment. Let τ be a task in $\mathcal{T}$. Consider an execution ρ of $\mathcal{P}$ in this environment. There could be many instances of τ executing in ρ. Let us consider one such instance, where at time t, τ

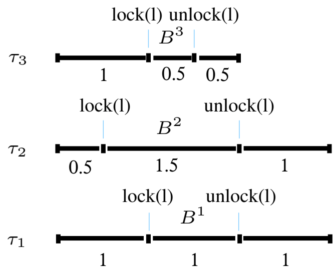


Fig. 5: Block WCETs of tasks of example program

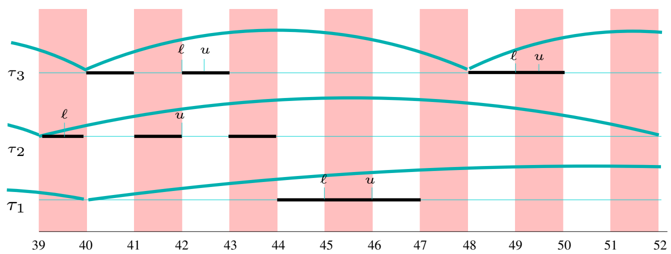


Fig. 6: Illustrating response time

moves into the ready queue with the program counter pointing to its start location. Let t' be the time at which this instance completes (that is τ executes its `end` instruction). Then the *response time* of this instance of τ is $t' - t$. We are interested in the *worst case response time* (WCRT) of τ which is defined to be the supremum of the response times of instances of τ over all instances of τ and all executions of $\mathcal{P}$ in the given environment.

In a similar way we can define the WCRT of a block of code B in τ, where we take the initial time t to be time the instance of τ is in the ready queue with the program counter pointing to the beginning of B, and t' to be the time the last instruction of B completes.

We note that the response time of a task (or a block of code) may exceed its WCET, as the task may lose processor time due to preemption by higher priority tasks, or due to blocking lock attempts. To illustrate this, consider a periodic program with three tasks τ_1 (priority 1, period 20), τ_2 (priority 2, period 13), and τ_3 (priority 3, period 8). Suppose the tasks have a simple structure comprising straight-line code, and each of them takes and releases a common lock l. Let the WCET for each segment of the tasks be as shown in Fig. 5. Consider a portion of a possible execution of $\mathcal{P}$ shown in Fig. 6. We note that τ_2, which has a WCET of 3, is ready to run at time 39 but completes execution only at time 44. Thus its response time in this instance is 5. This was due to the 2 units of processor time taken away by task τ_3 in its interruption during τ_2's execution. Notice also that the top priority task τ_3 is delayed by 1 unit of time waiting for τ_2 to release the lock it had acquired before it was preempted.

We say a periodic program $\mathcal{P}$ is *schedulable* if the WCRT of each task is less than or equal to its period. However, since it is difficult to know the exact WCRT, we will look for a conservative WCRT estimate which is less than or equal to the period of the task, to declare that a program is schedulable.

A. Computing Response time without Locks

In the classical setting of periodic programs without locks a conservative estimate of the WCRT for each task can be computed using Eq (1) below [5], [4]. Let $\mathcal{P} = (V, L, \mathcal{T})$ be a periodic program. We assume for convenience in the rest of this section that $\mathcal{P}$ has tasks $\tau_1, \ldots, \tau_n$ with distinct priorities. WLOG we assume τ_i has priority i. Further, each task τ_i has a WCET estimate C_i. Consider the equation below from [5] which in turn is based on the analysis in [4]:

$$R_i = C_i + \sum_{j>i} (\lceil R_i / T_j \rceil \cdot C_j) \qquad (1)$$

Theorem 1 ([5], [4]). *The least solution to Eq 1, whenever it exists, is an upper bound on the WCRT of task τ_i.*

Proof. Let L be *any* solution to Eq (1). We argue that L must upper bound the response time of *any* instance of task τ_i. Consider an instance of task τ_i that is enabled (enters the ready queue) at time t. Consider the time point $t + L$. If we ask ourselves how much processor time can be taken away in the interval $[t, t + L]$ by a higher priority task τ_j, it is clearly bounded by $\lceil R_i / T_j \rceil \cdot C_j$. Thus, the total time that can be taken away by all higher priority tasks put together is bounded by $\sum_{j>i} (\lceil R_i / T_j \rceil \cdot C_j)$. This leaves at least C_i time for task τ_i to execute, and hence it must complete execution by $t + L$. ☐

Algo. 1 below, which is similar to the recursive procedure proposed in [5], computes the least solutions to Eq (1) to compute conservative estimates of the WCRT of tasks, and thereby tell whether a periodic program is schedulable or not.

Algorithm 1: Check Schedulability (No Locks)

Data: Periodic program $\mathcal{P}$ without locks, WCET estimates C_i for τ_i
Result: $\mathcal{P}$ schedulable or not, and if so WCRT estimate for each task
foreach *task* τ_i **do**
  $L_i := C_i$;
  while *(L_i is not a solution to Eq (1) and $L_i < T_i$)* **do**
    $L_i := L_i + \sum_{j>i} (\lceil L_i / T_j \rceil \cdot C_j)$;
  end
  if *(L_i does not satisfy Eq (1) or $L_i > T_i$)* **then**
    return "Unschedulable";
  end
end
return "Schedulable", $L_1, \ldots, L_n$;

B. Computing Response Time with Locks

Thm. 1 no longer holds (and Algo. 1 is no longer sound) when tasks are allowed to take locks. This can be seen from the example program and sample execution in Figs. 5 and 6, where for instance task τ_3 has a response time of 3, but the least solution to the corresponding Eq (1) is 2. However, as we show below, it is possible to extend the classical approach to handle *non-nested* locks.

Before we consider the general case, it will be instructive to first consider the example program of Fig. 5. Let C_1, C_2, C_3 stand for the WCET estimates for tasks τ_1, τ_2, τ_3 respectively, and C_l^1, C_l^2, C_l^3 for the WCET estimates of the blocks B^1, B^2, B^3 respectively. Let us first begin by asking what is the response-time of the block B^1. Recall that this is the portion of code between the `lock`(l)-`unlock`(l) statements in τ_1. Since B^1 does not contain any lock statements, the response time for this follows Eq (1), and we can write Eq (6) to capture its response time. In a similar way the response time of the block B^2 is given by Eq. (5).

Next, we consider the top priority task τ_3. The only extra time it may spend is in waiting for its `lock`(l) instruction to succeed. This may happen because one of the lower priority tasks has acquired lock l and is yet to release it. Suppose this task is τ_2. Then τ_2 must be somewhere in block B^2. But how long can it be before τ_2 releases l? This is at most the *response time* for B^2. In a similar way, if τ_1 has taken the lock, τ_3 may end up waiting for at most the response time of B^1. Note also that τ_3 may have to wait for at most *one* of τ_2 or τ_1 to complete its lock block, never both. Thus, its response time is given by Eq (2).

Now let us consider task τ_2. It may be delayed either (a) waiting for its `lock`(l) statement to succeed because τ_1 has taken the lock l; or (b) because τ_3 takes away some time by preempting it. The former is bounded by the response-time of B^1, while the latter is bounded by the number of times τ_3 can interrupt it times the WCET of τ_3. Thus the response time of τ_2 is captured by Eq (3).

$$R_3 = C_3 + \max(U_l^2, U_l^1) \tag{2}$$
$$R_2 = C_2 + U_l^1 + \lceil R_2/T_3 \rceil \cdot C_3 \tag{3}$$
$$R_1 = C_1 + \lceil R_1/T_3 \rceil \cdot C_3 + \lceil R_1/T_2 \rceil \cdot C_2 \tag{4}$$
$$U_l^2 = C_l^2 + \lceil U_l^2/T_3 \rceil \cdot C_3 \tag{5}$$
$$U_l^1 = C_l^1 + \lceil U_l^1/T_3 \rceil \cdot C_3 + \lceil U_l^1/T_2 \rceil \cdot C_2 \tag{6}$$

To find the least solution to Eqs (2–6), we can apply the analogue of Algo. 1 to first compute $U_l^2 = 3.5$ and $U_l^1 = 6$ using Eqs (5–6). We can now use these values to compute the values $R_1 = 8$, $R_2 = 13$, and $R_3 = 8$. Since these are within the respective time periods of the tasks, we declare that the program is schedulable.

We can now tackle the general case. Consider a periodic program $\mathcal{P} = (V, L, \mathcal{T})$ satisfying the following assumptions (in addition to distinct priorities):

- $\mathcal{P}$ does not use nested locks. In particular, each task τ_i has a finite number of *lock*(l)-blocks $B_{l,1}^i, \dots, B_{l,n_{l,i}}^i$, with $n_{l,i} \geq 0$, for each lock variable $l \in L$. These blocks are pairwise disjoint.
- There is a bound N_l^i on the number of times τ_i takes lock l in any of its executions.
- The WCET of each task τ_i is C_i, and of each block $B_{l,k}^i$ is $C_{l,k}^i$.

The equations below capture the WCRT of the tasks and lock blocks of $\mathcal{P}$:

$$R_i = C_i + \sum_{l \in L} (N_l^i \cdot \max_{j<i} U_{l,k}^j) + \sum_{j>i} (\lceil R_i/T_j \rceil \cdot C_j) \tag{7}$$

$$U_{l,k}^i = C_{l,k}^i + \sum_{j>i} (\lceil U_{l,k}^i/T_j \rceil \cdot C_j) \tag{8}$$

Theorem 2. *The least solution to the system of Eqs (7,8), whenever it exists, is an upper bound on the corresponding WCRT of tasks τ_i and the blocks $B_{l,k}^i$.*

Proof. Once again we show that any solution to the sytems of equations (7) and (8) is an upper bound on the WCRT of the tasks and lock blocks of $\mathcal{P}$. Let $L_1, \dots, L_n$ and $L_{l,k}^i$ (for $i \in \{1, \dots, n\}$, $l \in L$, and $k \in \{1, \dots, n_{l,i}\}$) be a solution to the equations above. We first argue that the WCRT of a block $B_{l,k}^i$ is bounded by $L_{l,k}^i$. Since the block is free of lock statements, this is like the classical case and a similar argument to Thm. 1 applies to conclude that $L_{l,k}^i$ is an upper bound on the WCRT of $B_{l,k}^i$.

To argue that the WCRT of task τ_i is bounded by L^i, consider an execution of an instance of task τ_i where it is made ready at time t. Consider the time interval t to $t + L^i$. We claim that τ_i must finish its execution before $t + L^i$. Task τ_i may lose time because of two reasons: (a) it is blocked on one of its `lock`(l) instructions because some other task τ has taken the lock l. Now it must be the case that τ is a *lower* priority task than τ_i. Suppose τ had a higher priority than i. Then either it must have got blocked after acquiring l and before releasing it, or it was preempted by a still higher priority task τ'. The former case is ruled out since we don't allow nested locks. We can now apply similar reasoning to τ', and so on; but the buck must stop at the highest priority task. Since it cannot be preempted, it must be blocked waiting to acquire another lock; this is a contradiction to our no nested lock assumption. Thus, the total time that can be taken away due to τ_i waiting for a lock is bounded by the second term in Eq. (7). The second reason τ_i may lose time is (b) because of preemption by higher priority tasks. Like before, this is bounded by the third term in Eq. (7). Thus, there must remain at least C_i amount of time in the interval t to $t + L^i$ for τ_i to execute, and hence it must complete execution before $t + L^i$. □

Algo. 2 is an algorithm to compute the least solution to the system of Eqs. (7,8), and check schedulability of a periodic program with non-nested locks.

VI. Rules for Disjointness

In this section we describe a set of rules which tell us when two tasks of a periodic program are disjoint (that is, can never happen in parallel). We will then use these rules to propose a race-detection algorithm for periodic programs.

Algorithm 2: Check Schedulability With Locks

Data: Periodic program $\mathcal{P}$ with locks, WCET estimates C_i for τ_i and $C^i_{l,k}$ for lock block $B^i_{l,k}$
Result: $\mathcal{P}$ schedulable or not; if schedulable, WCRT estimates for each task

foreach *block* $B^i_{l,k}$ **do**
  $L^i_{l,k} := C^i_{l,k}$;
  while *(*$L^i_{l,k}$ *does not satisfy Eq (8) and* $L^i_{l,k} < T_i$*)* **do**
    $L^i_{l,k} := L^i_{l,k} + \sum_{j>i}(\lceil L^i_{l,k}/T_j \rceil \cdot C_j)$;
  end
  if *(*$L^i_{l,k}$ *does not satisfy Eq (8) or* $L^i_{l,k} > T_i$*)* **then**
    return "Unschedulable";
  end
end
foreach *task* τ_i **do**
  $L_i := C_i + \sum_{l \in L}(N^i_l \cdot \max_{j<i} L^j_{l,k})$;
  while *(*L_i *does not satisfy Eq (7) and* $L_i < T_i$*)* **do**
    $L_i := L_i + \sum_{j>i}(\lceil R_i/T_j \rceil \cdot C_j)$;
  end
  if *(*L_i *does not satisfy Eq (7) or* $L_i > T_i$*)* **then**
    return "Unschedulable";
  end
end
return "Schedulable", $L_1, \ldots, L_n$;

A. Disjoint Block Rules

Let $\mathcal{P} = (V, L, \mathcal{T})$ be a periodic program that (a) satisfies the no-nested-lock condition of Sec. V-B, and (b) has WCRT estimates R_τ for each task τ satisfying $R_\tau \leq T_\tau$ (that is, $\mathcal{P}$ is schedulable). The rules below tell us when two whole task bodies, or two blocks within them, are disjoint.

- Rule 1 (Same-Priority): *Let τ and τ' be two distinct tasks in $\mathcal{T}$ such that:*
  - *τ and τ' have the same priority (i.e. $p_\tau = p_{\tau'}$); and*
  - *Neither τ nor τ' shares a lock with a lower priority task.*

  Then τ and τ' are disjoint.
- Rule 2 (Same-Period): *Let τ and τ' be two distinct tasks in $\mathcal{T}$ such that:*
  - *τ and τ' have the same period (i.e. $T_\tau = T_{\tau'}$); and*
  - *Neither τ nor τ' shares a lock with a lower priority task.*

  Then τ and τ' are disjoint.
- Rule 3 (Low-Multiple-of-High): *Let τ_l and τ_h be two tasks in $\mathcal{T}$ such that:*
  - *τ_l has a lower priority than τ_h; (i.e. $p_{\tau_l} < p_{\tau_h}$);*
  - *The period of τ_l is a multiple of the period of τ_h (i.e. $T_{\tau_l} = k \cdot T_{\tau_h}$ for some $k \in \mathbb{N}$);*
  - *τ_h does not share a lock with a task of lower priority than τ_l; and*
  - *The WCRT estimate R_{τ_l} of τ_l is at most the period of τ_h (i.e. $R_{\tau_l} \leq T_{\tau_h}$).*

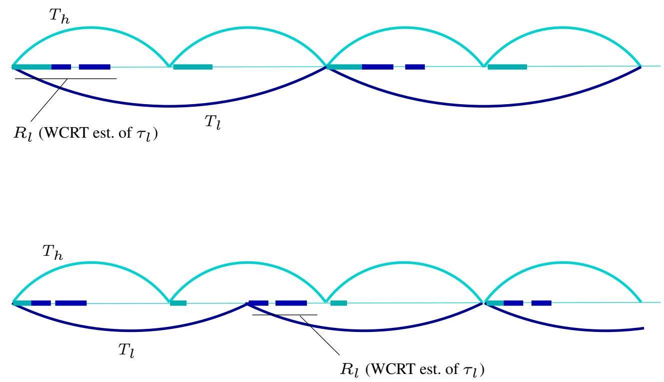


Fig. 7: Illustrating Rules 3 (above) and 5 (below)

  Then τ_l and τ_h are disjoint.
- Rule 4 (High-Multiple-of-Low): *Let τ_l and τ_h be two tasks in $\mathcal{T}$ such that:*
  - *τ_l has a lower priority than τ_h;*
  - *The period of τ_h is a multiple of the period of τ_l; and*
  - *τ_h does not share a lock with a task of lower priority than τ_l.*

  Then τ_l and τ_h are disjoint.
- Rule 5 (Low-WCRT): *Let τ_l and τ_h be two tasks in $\mathcal{T}$ such that:*
  - *τ_l has a lower priority than τ_h;*
  - *τ_l and τ_h have periods such that neither is a multiple of the other.*
  - *τ_h does not share a lock with a task of lower priority than τ_l.*
  - *Let m be the minimum* strictly positive *value in the set $\{(k \cdot T_{\tau_h}) \mod T_{\tau_l} \mid k \in \mathbb{N}\}$ (note that such an m must exist by the second condition above). The WCRT estimate R_{τ_l} of τ_l is at most m (i.e. $R_{\tau_l} \leq m$).*

  Then τ_l and τ_h are disjoint.
- Rule 6 (Lock): Let B_l and B'_l be two `lock`(l)-`unlock`(l) blocks in distinct tasks τ and τ' respectively. Then B_l and B'_l are disjoint.

Fig. 7 illustrates Rules 3 and 5.
We now claim that Rules 1–6 are sound in that:

Theorem 3. *Consider a periodic program $\mathcal{P}$, with no nested locks, and WCRT estimates which make it schedulable. Consider two blocks which satisfy the premise of one of the rules; then the identified blocks are indeed disjoint in $\mathcal{P}$.*

Proof. Let us fix a periodic program $\mathcal{P}$ without nested locks, and with WCRT estimates R_τ for each task τ in $\mathcal{P}$, which witness the schedulability of $\mathcal{P}$. Now suppose τ and τ' are two tasks in $\mathcal{P}$ satisfying the premise of Rule 1, namely that they have the same priority and neither of them shares a lock with a lower priority task. Now if there were no higher priority tasks and τ and τ' took no locks at all, then clearly τ and τ' can never overlap in their execution instances, since neither can preempt the other. However, even if there was a higher priority task say τ'', note that by our scheduling semantics, if τ'' were to interrupt τ during its execution, τ would resume execution ahead of any other tasks of the same priority that may be ready. So τ and τ' cannot interleave due to the preemption by a higher priority task. The other possible cause for interleaving could be because say τ gets blocked while trying to take a lock l that is already held by some other task of higher or lower priority. However, as argued earlier, a higher priority task holding l is ruled out. The case of a lower priority task holding l is ruled out by the premise of Rule 1. Thus it follows that τ and τ' cannot overlap in any execution. The soundness of Rule 2 follows a similar argument.

For Rule 3, suppose the period of τ_l is a multiple of τ_h. Let us say τ_l is made ready at some time t (which must be a multiple of its period T_{τ_l}). Now either t is also a multiple of T_{τ_h}, in which case τ_h will begin execution before τ_l, or τ_h is next scheduled at some time $t' > t$. In the former case, the only reason τ_h may not complete before τ_l gets to execute, is that τ_h is blocked on aquiring a lock. As in earlier arguments, this lock can only have been acquired by a task of priority *lower* than τ_l. But this is ruled out by the premise of the rule. In the latter case, by the premise of the rule, $t + R_{\tau_l} \leq t'$. Hence τ_l will complete its execution before τ_h can preempt it at t'.

For Rule 4, suppose T_{τ_h} is a multiple of T_{τ_l}. Consider a time t when τ_l is made ready. If τ_h is not also enabled at t, then by schedulability, τ_l must complete before $t + T_{\tau_l}$, which is before the time τ_h is enabled next. Hence they cannot overlap in this case. If τ_h is also enabled along with τ_l at t, then it must begin execution before τ_l does. The only reason it may not complete before τ_l is allowed to begin execution, is that it is blocked on a acquiring a lock l held by a task of lower priority than τ_l. But this is ruled out by the premise of the rule.

For Rule 5, again consider τ_l and τ_h satisfying the premise of the rule. Let t be a time point where τ_l is made ready. Either t is a multiple of T_{τ_h}, in which case τ_h is also made ready at the same time; or it is not, and arrives at some time t' later than t. The former case is similar to the situation considered in earlier cases, and the instances of τ_l and τ_h cannot overlap. In the latter case, by the premise of the rule, we must have $t + R_{\tau_l} \leq t + m \leq t'$, and hence τ_l would finish its execution by t', and the two tasks cannot overlap. The soundness of Rule 6 is standard. □

B. Computing the value m in Rule 5

Rule 5 requires us to compute the value m which is the smallest positive remainder that we can get by dividing an integral multiple of T_{τ_h} by T_{τ_l}. It is not difficult to see that all possible remainders must occur in the interval $[0, T]$ where T is the LCM of T_{τ_l} and T_{τ_h}. Thus it is sufficient to look at the multiples of T_{τ_h} upto T, and set m to be the minimum positive remainder we get by dividing these by T_{τ_l}.

C. Race Detection Algorithm

We now present the algorithm to detect races in periodic programs. Algo. 3 first identifies the set of shared variables accessed in the program and then lists all the conflicting access pairs, which are all assumed to be potentially racy initially. The algorithm, using the rules in Sect. VI and the *lockset analysis*, described next, then prunes out the pairs of accesses found to be non-racy.

An iterative lockset analysis computes the set of locks held at each statement in a program $\mathcal{P}$. At the program entry, it is assumed that no locks are held. For the `lock`(l) command, locks held are the set of locks held before this command along with the lock l. For the `unlock`(l) command, locks held are the set of locks held before this command with the lock l removed. For any other command, the lockset remains the same as held in the previous command. The *join* operation, in this analysis, is the intersection of locksets.

The algorithm uses the notion of *covers* which needs further explanation. Let τ_1 and τ_2 be two tasks in a periodic program $\mathcal{P}$ and s_1 and s_2 be two statements in $\mathcal{P}$. We say the pair of tasks (τ_1, τ_2) *covers* the pair of statements (s_1, s_2) if either s_1 is a statement in task G_{τ_1} and s_2 is a statement in task G_{τ_2} or vice versa (*i.e.* s_1 in G_{τ_2} and s_2 in G_{τ_1}).

Algorithm 3: Race Detection

Data: Periodic program $\mathcal{P}$
Result: List of potential races PR
Identify the set of shared variables V;
Find the list CA of conflicting accesses on V;
$PR := CA$;
Find list DT of disjoint tasks using rules in Sec. VI;
foreach *pair (s_1, s_2) of conflicting accesses in PR* **do**
  if *there is a pair (τ_1, τ_2) of tasks in DT, such that (τ_1, τ_2) covers (s_1, s_2)* **then**
    // (s_1, s_2) are non-racy
    $PR := PR - \{(s_1, s_2)\}$;
  end
end
Perform lockset analysis on each task in $\mathcal{P}$;
foreach *pair (s_1, s_2) of conflicting accesses in PR* **do**
  let L_1 be the lockset at s_1 and L_2 be that at s_2;
  if $L_1 \cap L_2 \neq \emptyset$ **then**
    // (s_1, s_2) are non-racy
    $PR := PR - \{(s_1, s_2)\}$;
  end
end
return PR;   // Set of potential races

VII. Experimental Evaluation

In this section, we first describe the implementation of Algo. 3 to detect races in periodic programs. We then explain the benchmarks used to evaluate the implementation followed by a discussion of the results.

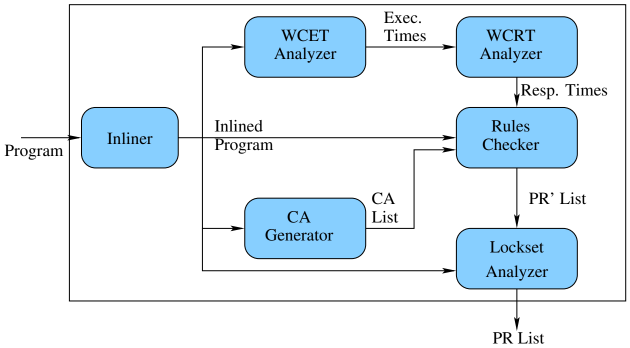


Fig. 8: Schematic of PEPRACER

A. Implementation

We implemented Algo. 3 in the tool PEPRACER as shown in Fig. 8. The tool has a preprocessor, which inlines the functions in the input program, a time analyzer which computes WCET of tasks using Heptane [6], and then their WCRT using Algo. 2. The CA generator identifies the shared variables, which are essentially global variables, in the program and then lists the conflicting access pairs. The Rules Checker identifies disjoint task pairs using the response times and eliminates conflicting accesses that are non-racy. The rules are successively applied, in the order listed in Sect. VI, on to the conflicting accesses to eliminate non-racy pairs. The Lockset Analyzer computes the locks held at each statement in the program and further eliminates the remaining conflicting accesses that are non-racy. The tool finally displays the potentially racy pairs.

We implemented PEPRACER in the OCaml based C Intermediate Language (CIL) static analysis framework [9]. The Inliner step in PEPRACER uses the built-in *inline* pass in CIL while the lockset algorithm and Rules Checker are implemented as new passes in CIL. The implementation of the WCET Analyzer is explained next.

a) WCET Analysis: WCET analysis was carried out on the benchmarks using the Heptane [6] tool. Heptane accepts inputs in the form of C programs. To prepare the benchmark programs the following modifications were made to them: All non-C constructs were translated to suitable C constructs, e.g. TASK in OSEK programs were converted to correspondingly named functions. All code was merged into a single C file. Some benchmark programs did not have the source for some of their parts. Heptane needs the source code for the entire program being analysed. Hence, all code for which source code was not available was replaced with minimal stubs. Loop bounds were provided using `ANNOT_MAXITER` as required by Heptane. These loop bounds were computed by manual inspection.

For each benchmark, WCET was separately computed for each of its task entry functions. Heptane supports WCET analysis for ARM and MIPS architectures. Where possible, WCET was run using default settings for both architectures. The difference between the WCET results for both architectures were found to average around 4%, never exceeding 20%.

Some aspects which may lead to our WCET estimates not being conservative are as follows:

1) Stub functions were used for those parts of the code whose source was not available. This accounts for $< 1\%$ of the total code analysed.
2) Loop bounds were defined using manual inspection.
3) A small number of lines of code had to be masked to prevent Heptane from crashing.

For more accurate WCET analysis, data corresponding to the specific target architecture being considered should be used. Several WCET analysis tools are available [8] both in the commercial and academic domain. The choice of the analysis tool would influence the accuracy of the WCET analysis.

B. Benchmarks

We tested the implementation on few benchmark periodic programs shown in Table II. The programs are taken from nxtOSEK benchmark set, lego-osek-master project, ev3OSEK benchmark set, nxt-osek-sumo-master project, and examples in [10] and [11]. These programs are designed to run on OSEK real time operating system and we have abstracted the program to make it suitable for the analysis. For example, we have abstracted shared data structures as simple variables and hence accesses to any field in the structure is considered as an access to its corresponding simple variable. The non-periodic tasks in some of the programs are taken to be tasks with arbitrarily high period. We have inlined the helper functions called in the tasks along with the calls to library functions. This will bring out the accesses to shared structures in the library. For example, the *ecrobot* library function `ecrobot_set_motor_speed`, which is called in *lego_osek.c*, accesses the shared write NXT_PORT_A port. The `GetResource`, `ReleaseResource` functions used to take and release locks, respectively, are taken to be the `lock`, `unlock` command in our analysis. We have annotated the program with task attributes like periodicity, priority, and WCET time, along with details of locks held.

C. Results

We ran our tool on the benchmark programs on an Intel Quad Core i7-3770 3.40GHz machine running Ubuntu 18.04.4. Table II shows the results of running our tool. The "Tasks" column gives the number of tasks in the program, "Sched." gives whether the program is schedulable or not (by Algo. 2), the number of conflicting accesses in a program is listed under the "CA" column, and the count of potentially racy pairs are given under the "PR" column. The "%Elim." column gives the percentage of conflicting accesses that are found to be non-racy. The last column gives the time taken by the tool, which was calculated using the Linux `time` command.

Our tool detects that the tasks in `avionicsN.c` program to be non-schedulable which is also detected by [11]. Rules 3, 4, and 5 depend on the response times of the tasks and we bypassed the application of these rules for `avionicsN.c`. The "PR" column in the table for `avionicsN.c` gives the count of potentially racy pairs detected due to the application

TABLE II: Results

Program	LoC	Tasks	Sched.	CA	PR	% Elim.	Time (sec)
fse_obstacle.c	24	2	Y	3	0	100	0.12
avionicsN.c	588	15	N	51	42	82	0.13
biped_robot.c	314	3	Y	1	0	100	0.12
sumo.c	799	4	Y	468	0	100	0.14
nxtgt.c	209	4	Y	15	0	100	0.13
lego_osek.c	807	2	Y	220	0	100	0.12
objectfollower.c	473	3	Y	16	0	100	0.12

of other rules. Our tool is able to filter out a large part of the conflicting access (CA) pairs as non-racy (on an average of 97% of CA pairs are eliminated).

Table III gives the coverage of the rules (Rule 1 - Rule 6). Here each rule is independently applied on to the conflicting accesses to demonstrate the value of each rule. Column "R1" gives the count of CA pairs flagged as non-racy, independent of the application of other rules. The case is similar with other columns. Recall that the non-trivial rules like Rules 3-5 use periodicity and/or response time to declare CA pairs as non-racy. A careful analysis of the count for these in Table III reveals their usefulness in flagging non-racy pairs. Some pairs are detected by these rules while not covered by the other simpler rules. It is even worthwhile observing that the CA pairs detected as non-racy by Rule 6 (the one based on locks) are covered by other rules. The developers can use this information to decide on whether to use expensive constructs like `lock-unlock` to ensure mutual exclusion when the task periodicity and response time can ensure it.

TABLE III: Rule Coverage

Program	CAs	R1	R2	R3	R4	R5	R6
fse_obstacle.c	3	0	0	3	0	0	0
avionicsN.c	51	0	9	-	-	-	0
biped_robot.c	1	0	0	0	0	1	1
sumo.c	468	0	202	202	468	0	464
nxtgt.c	15	0	0	0	15	0	0
lego_osek.c	220	0	0	220	0	0	216
objectfollower.c	16	0	0	11	11	16	2

VIII. Related Work

We begin with work related to computing response times and schedulability analysis. Apart from the work of [4], [5] already mentioned, feasibility analysis for real-time periodic tasks without locks have been studied by Baruah et al [12] and Pellizzoni and Lipari [13]. Baruah [14] studies schedubility under Earliest Deadline First and Stack Resource Policy (EDF+SRP) and gives an efficient algorithm for checking schedulability. Bertogna et al [15] study resource holding times (how long a task may hold on to a lock/resource) and give algorithms for computing and minimizing these times.

In closely-related classical work on real-time systems that use locks, Sha et al [1] consider a very general setting of priority-based preemptive scheduling, with FCFS among waiting tasks of the same priority (similar to our setting), with arbitrarily nested locks, and give sufficient conditions for schedulability of programs under these conditions. However the locks they consider are priority inheritance based locks which elevate the priority of a task if it is in a critical section to a level based on the priorities of the tasks waiting for (or that might acquire) this resource. Programs with such locks have the useful property that the blocking time of a task is bound by the longest WCET of a lock block (critical section) of a lower priority task. This facilitates their analysis and bounds on response time. In our setting of standard locks (though restricted to be non-nested) it is not clear if such properties can be exploited.

Related work on verification of periodic programs can be broadly classified into two categories: Verification of periodic programs using techniques like model checking, symbolic execution etc., and detecting data races in programs for embedded applications similar to periodic programs, using static analysis techniques.

Periodic programs with tasks prioritized in a rate monotonic fashion and communicating using shared variables, have been verified against safety properties using bounded model checking with different kinds of locks in [16], [17] and [18]. In their first paper of the series [16], the authors provide a time-bounded verification of safety properties where the sequentializations of programs are considered with respect to number of jobs of each task within the time bound. Priority and preemption locks are considered in [16] and the work is extended to include Priority Inheritance Protocol (PIP) locks in [18]. [17] proposes a new sequential composition mechanism to reduce the number of sequentializations and make the bounded verification scalable. However, the verification is bounded to a certain depth, and in general cannot be used to soundly detect all data races.

PLC programs are very similar to our periodic programs and are widely used in embedded safety critical software. Symbolic execution of PLC programs is developed in [10] where the authors convert PLC programs into C programs and use their rate-monotonic, priority-based, preemptive scheduling semantics to reduce the number of inter-leavings considered. The only way to use their symbolic execution to detect data races would be for the developer to introduce a counter for each shared variable and increment and decrement this counter, and then check for violations of assertion that encode a racy accesses to these variables. This technique is unlikely to be scalable.

Static analysis based techniques for detecting data races embedded software kernels and applications have been of recent research interest [2], [19], [3]. Schwarz et al [2] provide an algorithm to detect data races in multi-task programs with priority ceiling locks. Additional synchronization mechanisms including dynamic threads, suspend-resume of scheduler and tasks etc. are considered in [3]. Both these works exploit priorities and locks, but do not consider periodicity and WCRT information like we do, and would lead to less precise results on the class of periodic programs considered in this paper.

IX. CONCLUSION

In this work we have proposed a technique for statically detecting data races in periodic real-time programs with locks. Our contribution includes a response time analysis for such programs when the locks are used in a non-nested manner. Going forward, some interesting directions include using the insights in this paper to perform precise and efficient data-flow analysis for such programs; improving the tightness of the response time analysis; and extending the technique for detecting high-level races for the class of such programs and for periodic programs with other scheduling policies.

REFERENCES

[1] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[2] M. D. Schwarz, H. Seidl, V. Vojdani, P. Lammich, and M. Müller-Olm, "Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol," *POPL SIGPLAN Not.*, vol. 46, no. 1, p. 93104, Jan. 2011.

[3] R. Tulsyan, R. Pai, and D. D'Souza, "Static race detection for RTOS applications," in *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS*, ser. LIPIcs, vol. 182. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 57:1–57:20.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, p. 4661, Jan. 1973.

[5] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 01 1986.

[6] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, ser. OASICS, J. Reineke, Ed., vol. 57. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 8:1–8:12.

[7] "nxtOSEK/JSP: RTOS for Lego MindStorms NXT," 2013. [Online]. Available: http://lejos-osek.sourceforge.net/

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[9] G. Necula, "CIL – infrastructure for C Program Analysis and Transformation (v. 1.3.7)," http://people.eecs.berkeley.edu/ necula/cil/, 2002.

[10] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. ACM, 2017, p. 326336.

[11] C. D. Locke, L. Lucas, and G. J. B., "Generic avionics software specification," *Technical Report CMU/SEI-90-TR-8*, 1990.

[12] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real Time Syst.*, vol. 2, no. 4, pp. 301–324, 1990.

[13] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real Time Syst.*, vol. 30, no. 1-2, pp. 105–128, 2005.

[14] S. K. Baruah, "Resource sharing in edf-scheduled systems: A closer look," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*. IEEE Computer Society, 2006, pp. 379–387.

[15] M. Bertogna, N. Fisher, and S. K. Baruah, "Resource holding times: computation and optimization," *Real Time Syst.*, vol. 41, no. 2, pp. 87–117, 2009.

[16] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-bounded analysis of real-time systems," ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 7280.

[17] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, "Compositional sequentialization of periodic programs," in *Verification, Model Checking, and Abstract Interpretation*, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds. Springer Berlin Heidelberg, 2013, pp. 536–554.

[18] S. Chaki, A. Gurfinkel, and O. Strichman, "Verifying periodic programs with priority inheritance locks," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 137–144.

[19] N. Chopra, R. Pai, and D. D'Souza, "Data races and static analysis for interrupt-driven kernels," in *Programming Languages and Systems*, L. Caires, Ed. Cham: Springer International Publishing, 2019, pp. 697–723.