Verifying Periodic Programs with Priority Inheritance Locks

Sagar Chaki\textsuperscript{1}, Arie Gurfinkel\textsuperscript{1}, Ofer Strichman\textsuperscript{2}

FMCAD, October 22, 2013

\textsuperscript{1}Software Engineering Institute, CMU
\textsuperscript{2}Technion, Israel Institute of Technology
Periodic Embedded Real-Time Software

Avionics Mission System

Rate Monotonic Scheduling (RMS)

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>weapon release</td>
<td>10ms</td>
</tr>
<tr>
<td>radar tracking</td>
<td>40ms</td>
</tr>
<tr>
<td>target tracking</td>
<td>40ms</td>
</tr>
<tr>
<td>aircraft flight data</td>
<td>50ms</td>
</tr>
<tr>
<td>display</td>
<td>50ms</td>
</tr>
<tr>
<td>steering</td>
<td>80ms</td>
</tr>
</tbody>
</table>

Domains: Avionics, Automotive
OS: OSEK, VxWorks, RTEMS
We call them periodic programs

Context: Time-Bounded Verification [FMCAD’11, VMCAI’13]

Periodic Program
- Collection of periodic tasks
  - Execute concurrently with preemptive priority-based scheduling
  - Priorities respect RMS
  - Communicate through shared memory

Time-Bounded Verification
- Assertion A violated within X ms of a system’s execution from initial state I?
  - A, X, I are user specified
  - Time bounds map naturally to program’s functionality (e.g., air bags)

Locks
- CPU-locks, priority ceiling protocol locks [FMCAD’11, VMCAI’13]
- priority inheritance protocol locks

Main focus of this paper
Periodic Program (PP)

An N-task periodic program PP is a set of tasks \{\tau_1, \ldots, \tau_N\}
A task \tau is a tuple \langle I, T, P, C, A \rangle, where

- \( I \) is a task identifier = its priority
- \( T \) is a task body (i.e., code)
- \( P \) is a period
- \( C \) is the worst-case execution time
- \( A \) is the release time: the time at which task becomes first enabled

Semantics of PP bounded by time \( X \) is given by an asynchronous concurrent program:

\[
\begin{align*}
\text{k}_i &= 0; \\
\text{while (k}_i < J_i \&\& \text{Wait(\tau}_i, \text{k}_i)) \\
\text{T}_i() ; \\
\text{k}_i &= \text{k}_i + 1;
\end{align*}
\]

blocks \( \tau_{\downarrow i} \) until time
\[
A_{\downarrow i} + k_{\downarrow i} \times P_{\downarrow i}
\]

\[
J_{\downarrow i} = X/P_{\downarrow i}
\]
Priority Inheritance Protocol (PIP)

Ensure mutual exclusion when accessing shared resources

Works by dynamically raising and lowering thread priorities

- **Lock:**
  - If lock, is available, grab it.
  - Otherwise, block; the thread holding the lock “inherits” my priority

- **Unlock:** Release lock. Return to normal priority.

Provably avoids the priority inversion problem

- High-priority task is blocked on a lock held by low-priority task

However, incorrect usage leads to deadlocks

- In contrast to priority ceiling locks and CPU locks [FMCAD’11, VMCAI’13]
Our Contributions

Time-bounded verification of reachability properties of PP with PIP locks
  • Based on **sequentialization** [LR08], but supports PIP locks
  • **Challenge:** # sequentialization rounds needed for completeness cannot be statically determined
  • **Insight:** whether more rounds needed can be statically determined
  • **Solution:** Iterative-deepening search with fixed point check

Deadlock detection in PPs with PIP locks
  • Builds dynamically the **Task-Resource Graph**
  • Aborts if a cycle in that graph is detected

Implementation and Empirical Evaluation
Example: A Periodic Program

<table>
<thead>
<tr>
<th>Task</th>
<th>Prio ($l_i$)</th>
<th>WCET ($C_i$)</th>
<th>Period ($P_i$)</th>
<th>Arrival Time ($A_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_2$</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>1</td>
<td>4</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_0$</td>
<td>0</td>
<td>3</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Two PIP locks: 1 and 2
$l\downarrow_i =\text{acquiring lock } i$
$u\downarrow_i =\text{releasing lock } i$
Example: One Schedule

- τ↓1 unblocks, grabs \( \downarrow \uparrow 1 \), and resumes execution
- τ↓1 Inherits priority of \( \downarrow \uparrow 2 \)

Note: A scheduling point is either a preemption (\( \uparrow \)), a block (\( \ast \)), or a job end (\( \downarrow \))
Example: Viewing as a Round-Based Schedule

Note: A scheduling point is either a preemption (↑), a block (*), or a job end (↓)
Define: A round ends if the scheduling point is either a block, or a job end
Define: A round continues if the scheduling point is a preemption
Example: Viewing as a Round-Based Schedule

- Note: A scheduling point is either a preemption (↑), a block (*), or a job end (↓)
- Define: A round ends if the scheduling point is either a block, or a job end
- Define: A round continues if the scheduling point is a preemption
Sequentialization With PIP locks and fixed #Rounds

1. Create fresh variables for each round
2. Distribute jobs across rounds
3. Execute jobs using variables for the round it is in
4. Equate ending value at round $i$ to beginning value at round $i+1$
5. Building on prior work [VMCAI13] – adding PIP locks non-trivial
Complete Algorithm: Iteratively Increase #Rounds

**Challenge:** Different schedules have different number of rounds
- #Rounds = #Jobs + #Blocks
- #Blocks depends on the execution and preemption

**Solution:** Start with a small number of rounds (equal to #Jobs)
- Add more rounds iteratively till counterexample found, or fixed-point reached
Overall Algorithm

1: function PIPVERIF(C)
2: \( R := |J| \)
3: loop
4: \( x := \text{VERIFROUNDS}(C, R) \)
5: if \( x = \text{INCROUNDS} \) then \( R := R + 1 \)
6: else return \( x \)

Aborts if a job blocks but all \( R \) rounds already allocated

\[
[S_a(C, R)] = \emptyset \iff b^{R-|J|} \bullet a \not\in [C]
\]

\[
[S_b(C, R)] = \emptyset \iff b^{R+1-|J|} \not\in [C^\circ]
\]
Implementing $VerifRounds(C,R)$

Supports C programs w/ tasks, priorities, priority ceiling protocol, shared variables

Works in two stages:
1. **Sequentialization** – reduction to sequential program w/ prophecy variables
2. **Bounded program analysis**: bounded C model checker (CBMC, HAVOC, …)

Periodic Program in C

Sequentialization

Construct $S\downarrow a (C,R)$ and $S\downarrow b (C,R)$

CBMC

Check $[S\downarrow a (C,R)] = \emptyset$ and $[S\downarrow b (C,R)] = \emptyset$

Uses non-determinism (prophecy variables) to allow all possible interleavings between jobs and $R - |J|$ job block events

SAFE

UNSAFE + CEX

INCROUNDS

Periods, WCETs, Initial Condition, Time bound
Deadlock Detection: Encoding TRG

- **TRG**: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
Deadlock Detection: Encoding TRG

- **TRG**: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
Deadlock Detection: Encoding TRG

• **TRG**: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
• Transitive closure of TRG maintained and updated dynamically
• Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
Deadlock Detection: Encoding TRG

- **TRG**: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
TRG: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
Transitive closure of TRG maintained and updated dynamically
Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
Deadlock Detection: Encoding TRG

- **TRG**: Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)
NXTway-GS: a 2 wheeled self-balancing robot

Original: nxt (2 tasks)
• balancer (4ms)
  – Keeps the robot upright and responds to BT commands
• obstacle (50ms)
  – monitors sonar sensor for obstacle and communicates with balancer to back up the robot

Ours: aso (3 tasks)
• balancer as above but no BT
• obstacle as above
• bluetooth (100ms)
  – responds to BT commands and communicates with the balancer

Verified consistency of communication between tasks
## Experimental Results

<table>
<thead>
<tr>
<th>File</th>
<th>T</th>
<th>J</th>
<th>Rn</th>
<th>Vars</th>
<th>Cls</th>
<th>SAT</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nxt.bug1a.c</td>
<td>29</td>
<td>15</td>
<td>15</td>
<td>1.4M</td>
<td>4.3M</td>
<td>26</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.bug1b.c</td>
<td>58</td>
<td>15</td>
<td>15</td>
<td>2.5M</td>
<td>7.5M</td>
<td>54</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.bug1c.c</td>
<td>61</td>
<td>15</td>
<td>15</td>
<td>2.6M</td>
<td>8.1M</td>
<td>57</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.ok1.c</td>
<td>746</td>
<td>15</td>
<td>17</td>
<td>2.9M</td>
<td>9.0M</td>
<td>714</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug1a.c</td>
<td>73</td>
<td>15</td>
<td>15</td>
<td>2.7M</td>
<td>8.3M</td>
<td>68</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug1b.c</td>
<td>64</td>
<td>15</td>
<td>15</td>
<td>2.6M</td>
<td>8.0M</td>
<td>59</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug1c.c</td>
<td>33</td>
<td>15</td>
<td>15</td>
<td>1.7M</td>
<td>5.1M</td>
<td>29</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.ok1.c</td>
<td>4148</td>
<td>15</td>
<td>19</td>
<td>3.5M</td>
<td>10.9M</td>
<td>4,088</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug2a.c</td>
<td>43</td>
<td>15</td>
<td>15</td>
<td>1.6M</td>
<td>4.9M</td>
<td>39</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug3a.c</td>
<td>48</td>
<td>15</td>
<td>15</td>
<td>1.7M</td>
<td>5.1M</td>
<td>45</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug3b.c</td>
<td>35</td>
<td>15</td>
<td>15</td>
<td>1.5M</td>
<td>4.6M</td>
<td>32</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug3c.c</td>
<td>55</td>
<td>15</td>
<td>15</td>
<td>1.6M</td>
<td>4.9M</td>
<td>52</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.ok3.c</td>
<td>879</td>
<td>15</td>
<td>16</td>
<td>1.8M</td>
<td>5.5M</td>
<td>866</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug4a.c</td>
<td>63</td>
<td>15</td>
<td>15</td>
<td>2.0M</td>
<td>6.1M</td>
<td>58</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug4b.c</td>
<td>908</td>
<td>15</td>
<td>16</td>
<td>2.1M</td>
<td>6.4M</td>
<td>898</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.ok4.c</td>
<td>3,047</td>
<td>15</td>
<td>17</td>
<td>2.2M</td>
<td>6.7M</td>
<td>3,027</td>
<td>SAFE</td>
</tr>
</tbody>
</table>
Related, Ongoing and Future Work

Related Work

• Sequentialization of Periodic Programs with CPU locks and priority ceiling protocol locks (FMCAD’11, VMCAI’13)
• Sequentialization of Concurrent Programs (Lal & Reps ‘08, and others)
• Sequentialization of Periodic Programs (Kidd, Jagannathan, Vitek ’10)
• Verification of periodic programs using SPIN (Florian, Gamble, & Holzmann ‘12)
• Verification of Time Properties of (Models of) Real Time Embedded Systems
• Model Checking Real-Time Java using JPF (Lindstrom, Mehlitz, and Visser ‘05)

Ongoing and Future Work

• Verification without the time bound
• Memory Consistency based Sequentialization
• Abstraction / Refinement
• Modeling physical aspects (i.e., environment) more faithfully
• More Examples
QUESTIONS?