RV: where are we?

Daniel Bristot de Oliveira
Senior Principal Software Engineer
The initial patch set was under submission when I submitted this topic, but it was merged on 6.0!

So we will have a look at the current structure and talk about what is next!
Runtime Verification

- Runtime Verification (RV) is a lightweight (yet rigorous) formal method that with a more practical approach for complex systems.
- Instead of relying on a fine-grained model of a system (e.g., a re-implementation at an instruction level), RV works by analyzing the trace of the system's actual execution, comparing it against a formal specification of the system behavior.

This topic was explored during Daniel's PhD, where he used it as the basis for the creation of a preemption model for the PREEMPT_RT, then used to proof the scheduling latency bound for the kernel-rt.
Runtime Verification: IOW

- As the system runs, it generates events to be analyzed
- These events are analyzed against a well-defined description of the system
  - Online Synchronous if it blocks the system
  - Online Asynchronous if it does not block
  - Offline if it verifies the system based on a trace file
- The analysis produces a verdict
- The system can react to an unexpected event
  - Only makes sense for online RV

The current supported method is online synchronous.
Runtime Verification on Linux

System Trace → Monitor → Specification

Goto fail-safe mode

WARN() Fix the doc

The RV subsystem

RV Introduction
Why formal?

- Because it is precise and unambiguous
  - It is about reasoning, not about code
  - Math is math
  - See this "The Man Who Revolutionized Computer Science With Math*": https://www.youtube.com/watch?v=rkZzg7Vowao
- It is possible to analyze formal properties of your reasoning
  - Does the logic have contradictions?
  - Is the reasoning deadlock free?
- It is closer to other formal types of demonstration
  - Like the demonstration of the scheduling latency (my talk at LPC 2020).
- It helps to document the code
  - And adds value to the documentation for safety critical systems

* The man is Laslie Lamport, not me :-)*
Where are we from?
How it started

- How can I demonstrate the bound for the scheduling latency?
  - In a way that I could convince theoretical researchers
  - But that could also be meaningful for Linux people
- I solved this problem using an automaton model explaining the PREEMPT_RT synchronization.
  - LPC 2018 talk Mind the gap - between real-time Linux and real-time theory
  - Used the model specifications to derive a bound for the scheduling latency
  - LPC 2020 talk: "A theorem for the RT scheduling latency (and a measuring tool too!)"
Automata for complex models

- Automata is a formal method with a large set of operations for composition and validation
- A large automata can be built form a set of small automaton.
- The PREEMPT_RT thread model has:
  - +9k states
  - +21k transitions
  - Build on a set of automata
    - The vast majority of the automaton modules have 2/3 states
    - The largest has 10 states
- (Scientific) journal paper on the subject ->
Automata for RV

- Automata is a well defined formalism
- Automata is a method to model Discrete Event Systems (DES)
  - Formally, an automaton is defined as:
    - \( G = \{ X, E, f, x_0, X_m \} \), where:
      - \( X \) = finite set of states;
      - \( E \) = finite set of events;
      - \( f = \) transition function = \((X \times E) \rightarrow X\);
      - \( x_0 \) = initial state;
      - \( X_m \) = set of final states.
  - The language - or traces - generated/recognized by \( G \) is the \( L(G) \).

There are multiple different automata definitions, in this case we are talking about Deterministic Automata (DA)

Kernel doc explains it!
Automata for RV

- The good thing about automata is that it also has a graphical representation:

A state-machine is a sort of automata! It is one of the basis of CS.
Using automata to prove things

- It worked! By explaining the logical behavior, we derived a timing bound!
- While developing and testing the model ended up finding kernel bugs
  - So people asked my, why not make it generic?

Demystifying the Real-Time Linux Scheduling Latency

Daniel Bristot de Oliveira
Red Hat, Inc, Italy
daniel.bristol@redhat.com

Daniel Casini
Scuola Superiore Sant’Anna, Italy
daniel.casini@santannapisa.it

Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina, Brazil
romulo.dosilveira@ufsc.br

Tommaso Cucinotta
Scuola Superiore Sant’Anna, Italy
tommaso.cucinotta@santannapisa.it

Lemma 7.
\[ L_{BE} \leq \max(D_{ST}, D_{PQD}) + D_{PAE} + D_{PSD} \]  \hspace{1cm} (6)

Proof. The lemma follows by noting that cases (i-a), (i-b), (i-c), (ii-a), (ii-b) are mutually-exclusive and cover all the possible sequences of events from the occurrence of RBP, and set_need_resched, to the time instant in which \( \tau_{I}^{PID} \) is allowed to execute (as required by Definition 1), and the right-hand side of Equation 6 simultaneously upper bounds the right-hand sides of Equations 2, 3, 4, and 5.

Theorem 8: The scheduling latency experienced by an arbitrary thread \( \tau_{I}^{PID} \) is bounded by the least positive value that fulfills the recursive equation:
\[ L = \max(D_{ST}, D_{PQD}) + D_{PAE} + D_{PSD} + \mu(L) + \mu(L) \]  \hspace{1cm} (7)

Proof. The theorem follows directly from Lemmas 7 and Equation 1.
Efficient automata verification

- The PREEMPT_RT model was a super-high-frequency one
  - Multiple events per microsecond
- Doing it in user-space was not efficient due to trace buffer overhead
- So we developed a way to transform the automata into code and run it in the kernel
  - $O(1)$ operating
  - It was faster than just tracing
    - Because it is a simpler operation
- And we ended up finding more bugs

Efficient Formal Verification for the Linux Kernel

Daniel Bristot de Oliveira$^{1,2}$, Tommaso Cucinotta$^2$, and Rômulo Silva de Oliveira$^3$

1 RHET Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.
2 RETIS Lab, Scuola Superiore Sant’Anna, Pisa, Italy.
3 Department of Systems Automation, UFSC, Florianópolis, Brazil.

Abstract. Formal verification of the Linux kernel has been receiving increasing attention in recent years, with the development of many models, from memory subsystems to the synchronization primitives of the real-time kernel. The effort in developing formal verification methods
End efficient

● Running RV code in kernel is faster than tracing it!
RV meets safety-critical systems
RV and safety-critical system

- RV uses formal-method
  - Which is at the top of the list of approaches to use for certification
- Three birds with a single stone!
  - Documents the system using automata
  - It verifies the system at the development/testing phases
  - It monitors the system at runtime
    - Reacting to unexpected events
- It only uses well-established technologies in the safety-critical fields:
  - Pure and straightforward C code
  - Statically allocated memory
  - No loops and so on
RV and safety-critical system

- I started working with people in the Elisa Project
- Gabriele Paoloni and I are working on the approach presented at LPC 2021
RV in safety-critical systems

RV and safety-critical system

- RV can be used to document the kernel
- But also to document how the system should behave:
  - How the kernel is suppose to behave
  - How the user is supposed to behave
- It can be used to split subsystems into small parts
  - Each small part can be qualified as a black box (ISO 26262 part 12)
  - RV monitors the interface (ISO 26262 part 6)
- Further information:
  - See talk ->
  - Together with Gabriele Paoloni
Where are we?
Merged!
Interface and concepts

- RV subsystem is an interface and a set of tools to make RV accessible
- A RV monitor composed of:
  - A model/specification
  - The instrumentation
- A reactor is an action available to the monitor
  - Can be invoked by the monitor if an unexpected even happens
- Monitors can be enabled at runtime
  - Each monitor can have one enabled reactor
Interface demo

Demo video
https://youtu.be/V42BFNKbh0g
Monitor synthesis

- The RV enables conversion of an automata.dot -> kernel RV monitor
  - This is done with a dot2k utility
- **dot2k** creates a skeleton of a RV monitor module
- The monitor also uses a set of C **macros** to generate the **monitor code** for the specific type of monitor
- The only thing left for the humans to do is the **instrumentation**
  - The connection between a kernel event and the model event

Using code generation automatizes the process, making it less prone to error and easier to qualify.
Monitor synthesis demo

Demo video

https://youtu.be/3yDxz1Sl4k8
RV Structure
RV Interface

- kernel/tracing/rv/
  - rv.c: rv interface
    - Startup
    - Register & control monitors
  - rv_reactor.c:
    - Startup
    - Register & control reactors
RV Monitors

- kernel/tracing/rv/monitors
  - Each monitor has its own directory
  - They are independent from the interface
    - The interface only controls the monitoring session
  - For deterministic automata monitors, there are two files:
    - .h - the automata
    - .c - instrumentation and control
  - Other types of monitors might have different files
    - But it is important to keep the specification separated from the code
RV Headers

- include/linux
  - rv.h: monitor interface

- include/rv
  - Instrumentation.h: helper functions for instrumentation
  - automata.h: helpers for automata operation
  - da_monitor.h: helpers for deterministic automata monitor
  - Other types of monitor (timed automata) will add files here
RV Structure

RV Tools

- tools/verification:
  - dot2/:
    - dot2 tools: tools to create the skeleton of the monitor
    - dot2k_templates/: templates for the monitor
  - models/:
    - A place to store models/specification
    - Sample monitors that can be used as starting point
The RV subsystem

RV Documentation

- Documentation/trace/rv/:
  - runtime-verification.rst
  - da_monitor_instrumentation.rst
  - da_monitor_synthesis.rst
  - deterministic_automata.rst
  - For each monitor:
    - monitor_wip.rst
    - monitor_wwnr.rst
RV: what is next?
Watchdog monitor

- It was part of the first patchset (working with Elisa people)
- The idea is to monitor the watchdog usage until reaching a safe state
  - Open -> start -> set a timeout -> ping at least once
  - Avoid some features to reduce the amount of code to be inspected without changing the code
    - Like the hrtimers usage on watchdog dev
- The monitor automaton is generic
- In the patchset I added some options requested by the safety analysis made in the Elisa group
  - But because I did it all in a single patch, the explanation was not clear
- The patchset also included a user-space tool to exercise the monitor and to serve as starting point for the monitor
Monitor options

- We can enrich monitors with additional options/parameters
- For example:
  - int value to be the max safe watchdog timeout
- Each monitor has its folder also to store these specific options
- I am not sure if I will:
  - Add a file per option
    - Easy to use
    - More memory
  - A single option file
    - It will need a syntax like the event's format file
    - Less memory
Modular monitors

- Now models are built-in only
- But they can be loaded as module - I started them as module
- I removed the export symbols in the initial patch set to avoid problems
- I just need to export symbols in include/linux/rv.h
- It has a drawback:
  - Each monitor will have its own DECLARE_TRACE
  - That is why each monitor has a src dir: to store these monitor specific things
Monitors with ebpf

- It is also possible to have monitors in eBPF
- I have a dot2bpf implementation
  - C & libbpf
  - Process the automata in the kernel
  - Feedback to user-space
- But it works and will share most of the kernel headers
  - To deduplicate the code of the eBPF and in-kernel option
- There is also a new set of monitors that I am doing with ETH Zurich that will use eBPF for user-space processing
Qualifying the monitors code

- We need to qualify the monitors' code according to the safety regulations
- This is part of the work we do with Elisa
- Red hat is committed to that as well
- The monitor was designed with this in mind
  - Pure C
  - No memory allocation
  - Self-generated
- But it will certainly require some changes that might add overhead
  - So we might have a da_safe_monitor.h
Other monitors

- Integrating the preemption model
  - It will require some extra work in the instrumentation, adjusting existing tracepoints
  - It will happen along with rtsl integration on rtla
    - So we will have the logical and the timing verification of the proven scheduling latency.
- Other monitors for RT
  - Function calls that are not guaranteed to be real-time
  - [Potential] priority inversion scenarios
- I am working with ETH Zurich and the University of Copenhagen on other types of formalism
  - They are more complex than automata, and allow timing in the equations
Questions?