Towards Non-invasive Run-time Verification of Real-Time Systems

Ricardo C. Pinto
LaSIGE
Faculty of Sciences - University of Lisbon
ricardo.pinto@fc.ul.pt

José Rufino
LaSIGE
Faculty of Sciences - University of Lisbon
ruf@di.fc.ul.pt

***Abstract*—Support for Run-time Verification (RV) has mostly been provided by software mechanisms, via the instrumentation of code for observing (monitor) and handling deviations from specification. Although this approach is fitting for some domains, it can have a nefarious influence in embedded real-time systems, impacting the system from the analysis to the operation stages.**

A novel alternative to code instrumentation is the embedding of such mechanisms directly in hardware, thus negating the impact in system properties, namely timeliness. The availability of soft-processors and companion System-on-a-Chip (SoC) Intellectual Property cores enable the hardware-based approach to RV.

This paper addresses the foundations for RV support via hardware mechanisms. A flexible observer entity is defined, to be merged into a SoC architecture. Monitoring is performed at the SoC bus that interconnects processor and peripherals, enabling the gathering of information regarding events of interest occurring during system execution and relaying it to external entities for handling.

I. Introduction

Run-time Verification (RV) is a well-know technique in the software world to perform the verification of a system. It is applied to a software design, to be used throughout its life-cycle stages - from early verification to operational deployment. The cornerstone of RV is the monitoring of values and events, and then comparing them to a given specification.

The classical approach to RV has been through *code instrumentation*. The system software is instrumented with specific functions which are not part of the functional specification of the system. These functions are executed during run-time, monitoring and assessing the state of the system, i.e. its adherence to the functional requirements. Instrumented code is therefore closely intertwined with the code dictating the progression of the system itself (*see* Figure 1).

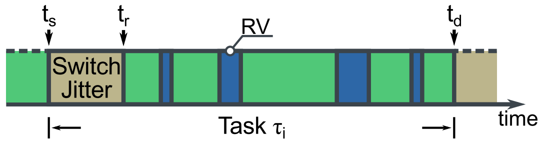

Fig. 1: Task Switching and Execution with RV

The usage of current RV techniques in real-time systems is a double-edged sword: whilst the systems can benefit greatly from RV, the overhead imposed by code instrumentation raises issues stemming in system design up to analysis and operation. At system design, adding code to a real-time task implies a higher Worst-Case Execution Time (WCET) which must be computed for schedulability purposes. At operation, the observation of the system interferes with the system itself due to the *observer effect*[1].

An illustration of such disturbances can be seen in Figure 1, where a piece of code has been added to measure the switching time of a real-time task. The task τ_i has a computed switching time at t_s. The effective time of switching is $t_r = t_p + t_j$, where t_j is the time added by execution jitter, including RV code. There are additional disturbances caused by other RV statements, which result in a higher WCET than a task without RV.

A solution for the RV issues raised by code instrumentation comes from the hardware domain. The growth in the usage of reconfigurable logic supporting System-on-a-Chip (SoC) designs (soft-processors and peripherals) enables the design of innovative solutions to address issues raised by software, and RV is no exception. The inclusion of non-invasive, hardware-based RV mechanisms negates the penalties in timeliness and performance, thus providing results with higher accuracy and without any impact on the system itself.

This paper presents the current work on designing and implementing a hardware-based observer entity aiming at non-intrusive event monitoring. Such entity will provide the support for effective, non-intrusive RV in embedded real-time systems. The remainder of this paper is organized in the following manner: Section II define the System Model to be used in the definition of an observer entity for RV; Section III details the design, specification and implementation in a SoC platform of such observer entity; Section IV presents related work in hardware support for RV and Section V concludes this paper discussing future work directions for achieving robust RV.

This work was partially supported by the EC, through project IST-FP7-STREP-288195 (KARYON) and by FCT, through project PTDC/EEI-SCR/3200/2012 (READAPT), by LaSIGE Strategic Project PEst-OE/EEI/UI0408/2014, and Doctoral Grant SFRH/BD/72005/2010.

[1] The *observer effect* designation stems from physics, where the act of observing a phenomenon interferes with its characteristics.

II. System Model

The observer entity is to be implemented in an embedded real-time system, consisting of an *execution platform* comprised both of hardware and software. Therefore, the definition of such platform is in order. Furthermore, a definition of an *event* is also necessary, formalizing what should be captured by the observer entity.

A. Real-time System Execution

The execution of a real-time system relies on two supporting platforms: *software*, through a Real-Time Operating System (RTOS) providing scheduling and dispatching facilities, together with system primitives for Input/Output (I/O) activities; *hardware*, through an embedded computing platform supporting the execution of the software entity.

1) Hardware: Current embedded systems are implemented resorting to computing platforms which are integrated in a single integrated circuit. An instance of such computing platform is a microcontroller, which has a processing element (CPU) and several I/O peripherals to exchange data with the environment or other systems. Such systems are known as System-on-a-Chip (SoC), due to its level of integration. A diagram showing such a system in presented in Figure 2.

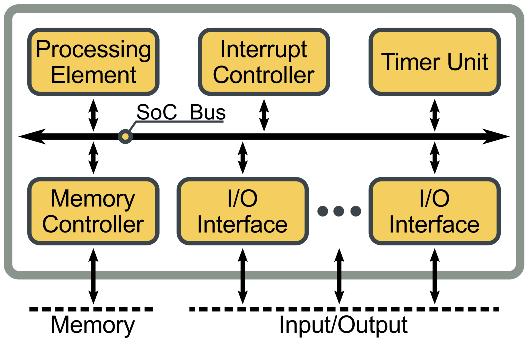

Fig. 2: Generic SoC Computing Platform Architecture

The hardware platform provides the resources necessary for the software entity: *Processing Element*, offering the processor resources; *I/O Interfaces*, exchanging data with external systems and/or the environment; *Memory*, to hold the software executable and state (variables); *Timer Unit*, providing the system with the ability to count time; *Interrupt Controller*, managing the interrupt requests coming from peripherals and feeding them to the processing element.

These components are interconnected through a SoC bus, in a (multi-)master/slave. Components are memory-mapped, with each component on the SoC being accessed through a range of addresses. The operations to be performed are either read or write, in a similar fashion to a memory device. These operations are initiated through master components, which are the only ones allowed to initiate a new transfer. The bus also embeds the interrupt request lines, allowing the slave devices to signal the masters of their need to communicate, e.g. signal they have data available to be transferred to the master.

All (software) instructions to be executed by the processing element have to pass through the bus, coming from the memory component. The transfer of data through the bus can be modelled as $Bus_{trx} \overset{def}{=} (operation, data, address)$, where: *address* is the value of the addressed component; *data* is the value of the data being exchanged; *operation* is the direction of the data, e.g. read or write. There may be additional control signals like transfer length, which are not of interest for monitoring purposes, and therefore not considered.

2) Software: The software platform comprises a set of tasks τ_i, mapping the intended functional specification into software. The execution of the tasks is supported by a RTOS. The usage of a RTOS provides the scheduling and dispatching of the tasks together with primitives to perform I/O activities, inter-task synchronization and communications.

Task execution cannot be decoupled from RTOS execution. Every time a task invokes a system primitive, RTOS facilities are used to fulfill its function, thus deviating the task execution from its designed flow. Furthermore, a scheduling function is performed periodically, deciding which task should be given processor resources. A companion dispatch function is also performed, to perform the actual task switching. An illustration with an example of these is shown in Figure 3.

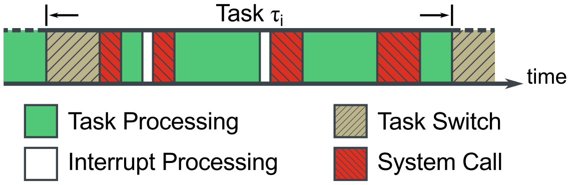

Fig. 3: Generic Task Execution Model

During task execution in an RTOS, there are at least three components: *Task Switch*, when the RTOS dispatches the task to be run; *Task Processing*, with code of the task itself; *System Call* for I/O activities or inter-task synchronization and communication. An additional component is the *Interrupt Processing*, which is executed upon interrupt signalling.

B. Events

An *event* is the information gathered through monitoring of a parameter of interest, e.g. memory address, interrupt line. An event ϵ is defined as a tuple $\epsilon \overset{def}{=} (t, s, i)$ where: t is the time of occurrence; s is the source of the event; i is the specific information pertaining to the event. The time t increases monotonically, establishing event causality. Two or more events may occur at the same time, i.e. $t_{k+1} \geq t_k$.

Broadly speaking, events can have two origins: *hardware*, such as interrupts, memory accesses; *software*, such as values of variables, flow of execution (instructions) or interrupt handling routines.

The identification of events, both in time and source is the basic functionality required for monitoring. Additional information regarding the event enrich the quality of the information provided by the monitoring, and thus lead to the support of RV mechanisms.

III. Observer Entity

The observer does not interfere with the behaviour of the observed system, thus negating the observer effect. The properties of the observer are: **non-intrusive**, not requiring code instrumentation nor affecting system operation; **configurable**, being able to accommodate different event triggers.

A. Design

Using the previously defined system model as working basis, an Observer Entity (OE) is defined, to be integrated in an embedded computing platform equivalent to the one presented in Figure 2. The ability to connect to an internal bus architecture is crucial, enabling the observation of data transfers and signalling taking place inside the computing platform, namely instructions and interrupts.

The ability to monitor interrupts and memory addresses allows to measure the latency of a task switch, from the instant the Timer Unit signals the passing of a tick to the scheduler to the time the task switch is completed. The design of an entity with the previous requirements results in the architecture shown in Figure 4.

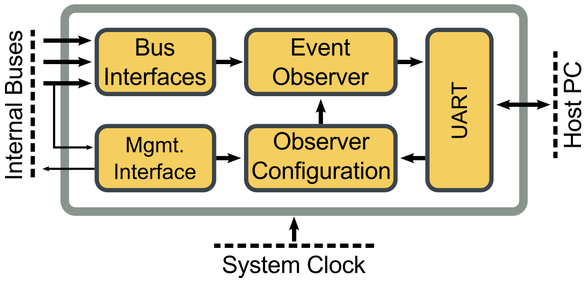

Fig. 4: Observer Entity Architecture

The OE shown in Figure 4 is plugged to the internal buses of a SoC architecture, and is comprised of several modules: *Bus Interfaces*, managing the physical interface to the buses and performing the detection of bus activity, e.g. bus transfer or interrupt; *Management Interface*, handling the support for configuration via the bus itself; *Observer Configuration*, storing the aforementioned configuration, i.e. which events should be detected; *Event Observer*, detecting events of interest based on the configuration and tagging them to be relayed to other systems; *UART*[2], providing an Out-of-band (OOB) interface for relaying the detected events to another system, e.g. a Personal Computer (PC).

The configuration of the OE can be performed through: the system being monitored itself, preferably upon system initialization; OOB, via the UART. The key-point of the OE is that it can be reconfigured after the system is deployed. Such architecture effectively enables non-intrusive hardware monitoring, with the flexibility of being able to accommodate detection of different events.

The operation of the OE is performed at every hardware clock cycle, synchronously with the bus. The OE continuously monitors the bus to detect the start of a new bus transfer operation, or the assertion of an interrupt line. The operation of the OE is described by the algorithm shown in Algorithm 1.

Algorithm 1: Event Monitoring

```
Input: System clock hardware_clock_tick
Output: Event evt
Initialize(Config)
foreach hardware_clock_tick do
    numTicks ← numTicks + 1
    if newTransfer($Bus_{trx}$) then
        if $\exists\, id \in Config : Config[id] = Bus_{trx}.address$
        then
            evt.time ← numTicks
            evt.source ← Config[ID].source
            evt.info ← $Bus_{trx}.data$
            outputEvent(evt)
        foreach
        $id \in Config : Config[id] = Bus_{trx}.interrupt$ do
            evt.time ← numTicks
            evt.source ← Config[ID].source
            evt.info ← null
            outputEvent(evt)
```

The tick count `numTicks` increases monotonically with each hardware clock tick. For a 50 MHz clock, it increases every 20 ns. If bus activity is detected, the configuration table is checked and if there is a match, the event is tagged with: timestamp, source and info, e.g. data in a memory access.

B. Implementation

The implementation of the OE architecture shown in Figure 4 is being performed in VHDL[3], and integrated in a LEON3 SoC [1], which includes the LEON3 soft-processor. The LEON3 processor implements the SPARC V8 Instruction Set Architecture (ISA) [2], and is connected to the peripherals through ARM Advanced Microcontroller Bus Architecture (AMBA) [3]. A diagram showing the LEON3 SoC together with the Observer is shown in Figure 5.

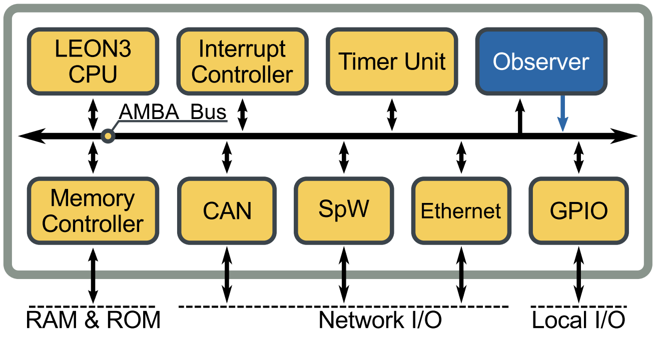

Fig. 5: Observer Entity on a LEON3 System-on-a-Chip

The SoC design provides several IP cores implementing I/O interfaces, together with a Memory Controller for Random-Access Memory (RAM) and Read-Only Memory (ROM).

[2] UART - Universal Asynchronous Receiver/Transmitter

[3] VHDL stands for Very High Speed Integrated Circuit Hardware Description Language

The AMBA bus is a high performance multi-master bus. Data is transferred in parallel, with one transfer per clock-cycle. Addressing is performed through memory-mapping, where each component in the SoC is seen as a range of memory addresses. The memory where software code resides is no exception, exposing its accesses to hardware monitoring.

The specification [3] defines two types of bus: AMBA High-performance Bus (AHB), for high-throughput components, e.g. CPU, RAM, Ethernet; Advanced Peripheral Bus (APB), for low throughput components, e.g. UART, General Purpose I/O (GPIO). Both these buses are connected to the OE, enabling monitoring of data exchange between SoC components.

The implementation present in the LEON3 SoC embeds the interrupt request lines of the peripherals in the AMBA bus. Such embedding eases the monitoring of both data transfers and interrupt requests, since both are available through the same bus interface.

The logical structure of the information pertaining to a monitored event is shown in Figure 6, described in VHDL.

```
-- Event
type event_t is record
  Time      : timestamp_t;    -- Time of occurrence
  Source    : event_source_t; -- Source
  Info      : event_info_t;   -- Specific Data
end record event_t;

-- Event Source
type event_source_t is record
  ID     : integer; -- ID: Task t1, Interrupt 15
  Class  : integer; -- Classes, e.g. hw or sw
end record event_source_t;

-- Event Specific Data
type event_info_t is record
  address   : address_t;   -- Address
  data      : data_t;      -- Data
  operation : operation_t; -- Operation, e.g. read
end record event_info_t;
```

Fig. 6: VHDL Description of Event Information

The `timestamp_t` data type should be wide enough to avoid rollover of time information, and is dependent on the system clock frequency. The rollover can be seen as a violation of the monotonicity, where time goes backward. The `Source` field stores data pertaining to the source of the event. Furthermore, it can be configured with the `Class` extra field, to provide context for the event specific data field, `Info`. The event contained in the format shown in Figure 6 is converted into a stream of bits to be transmitted by the UART via a VHDL function.

C. Event Data Output & Exploitation

The usage of an UART as an OOB mechanism enables the processing and exploitation of the generated event data on an external system, such as a PC. Therefore, the OE can be used to: characterize the system during the Verification & Validation (VV) stage, including timeliness; log the behavior for performance and timeliness assessment; enable proactive fault-tolerance mechanism design and execution.

The data output format is configurable via the Observer Configuration module, and can be formatted to be directly used by specific verification and visualization tools. The formatting is performed directly in hardware, by using specific serializing functions to convert the storage representation of Figure 6 into a stream of bits to be output by the UART. Output of raw data together with software filtering on the host system is also possible, enabling flexible monitoring data exploitation.

An envisaged visualization tool to be used with the OE is Grasp [4]. Grasp was designed for the visualization of real-time system execution, namely task execution and switching. Such a tool enables the verification of scheduling information, by visualizing the task execution and preemption points.

IV. Related Work

The design of hardware RV mechanisms has been receiving a growing interest. A first approach to monitoring was introduced in [5], with minimal code instrumentation. The approach in [6] solved the instrumentation issue, using a dedicated CPU.

Some of the the approaches address both the issue of monitoring and verification in a single instance [7]. The verification procedure is mapped into soft-microcontroller units, embedded within the design, and use formal languages such as past-time Linear Temporal Logic (ptLTL) for verification, with clauses checked by a CPU embedded in the design.

V. Conclusion and Future Work

Online monitoring and Run-time Verification (RV) of embedded real-time systems is a topic which is expected to grow in the coming years, thrusted by the design of autonomous control applications. The application of RV to real-time systems, however, brings an overhead which may be too costly, due to the impact in timeliness. The availability of soft-processors and SoC designs opens room for novel monitoring and RV, supporting non-invasive hardware-based RV for autonomous applications.

The provision of a reconfigurable and non-invasive Observer Entity (OE) for monitoring is the first step towards more sophisticated mechanisms and services. The data gathered by the OE can be used to feed verification clauses, thus enabling flexible non-invasive hardware-based RV.

References

[1] *GRLIB IP Library Users Manual*, Aeroflex Gaisler A.B., Apr. 2014. [Online]. Available: http://gaisler.com/products/grlib/grlib.pdf

[2] *The SPARC Architecture Manual*, SPARC International Inc., 1992.

[3] ARM Limited, *AMBA Specification*, ARM Specification 2.0, May 1999.

[4] M. Holenderski, M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 37–42.

[5] M. El Shobaki and L. Lindh, "A Hardware and Software Monitor for High-level System-on-Chip Verification," in *2001 International Symposium on Quality Electronic Design*, 2001, pp. 56–61.

[6] J. C. Lee, A. S. Gardner, and R. Lysecky, "Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems," *Engineering of Computer-Based Systems, IEEE International Conference on the*, vol. 0, pp. 52–60, 2011.

[7] T. Reinbacher, M. Függer, and J. Brauer, "Runtime verification of embedded real-time systems," *Formal Methods in System Design*, pp. 1–37, 2013.