Exploitation of Non-intrusive Monitoring in Real-Time Embedded System Systems*

Ricardo C. Pinto and José Rufino

LaSIGE - Faculty of Sciences - University of Lisboa
ricardo.pinto@ciencias.ulisboa.pt, ruf@di.fc.ul.pt

Abstract. Software execution monitoring in embedded systems can be performed with different purposes, ranging from system characterization to run-time verification (RV). Traditional RV techniques require the intrumentation of the code for monitoring, which brings an overhead to the execution of the system - both in performance and timeliness. In real-time systems this overhead is exarcebated by the need of new worst-case execution time estimation and schedulability analysis.
In this paper we show how non-intrusive monitoring mechanisms can be exploited to support Run-time Verification (RV) in real-time embedded systems, thus allowing run-time verification without the need for code instrumentation and therefore negating the penalties incurred by instrumentation.

Keywords: Run-time Verification; Real-time Embedded Systems; Non-intrusive Monitoring

1 Introduction

Ever increasing deployments of applications based on cyber-physical systems have pushed the topic of system monitoring and Run-time Verification (RV) for embedded systems into the agendas of both academia and industry. Such systems have a strong (feedback) connection to their surroundings - e.g. autonomous vehicles - and failure of such a system may translate into catastrophic consequences. Therefore, guaranteeing correctness of the behaviour at all times is essential.

The basis of RV consists in monitoring the state of a system. Such state can be described through: values of variables; execution of functions and procedures; input/output activities, materialized by the reading/writing of specific memory addresses, or ports. The result of the monitoring activities is then verified against a specification, in order to assess the adherence of the system's behaviour to the specification - in short, system correctness.

* 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), through LaSIGE Strategic Project PEst-OE/EEI/UI0408/2014, and Individual Doctoral Grant SFRH/BD/72005/2010.

The collection of data needed by monitoring usually requires the modification of the source-code (instrumentation). While such approach can be reasonable for large scale systems, it faces many obstacles upon application to the realm of real-time embedded systems. Some obstacles are the scarcity of resources which characterizes such systems, e.g. computational resources, data storage and energy. Other obstacles are the need to re-evaluate the Worst-Case Execution Time (WCET) and performing a new schedulability analysis - which in the extreme case may deem the system as unschedulable with RV enabled.

Therefore, novel and innovative techniques are in need for enabling RV in real-time embedded systems. An answer comes from the realm of reconfigurable computing platforms, where the usage of System-on-a-Chip (SoC) architectures enables the monitoring to be performed in hardware, at the lowest possible level and in a non-intrusive way, negating the overhead incurred by code instrumentation.

In this paper we present how the exploitation of a non-intrusive monitoring infrastructure can be achieved. Such exploitation can have two facets: *monitoring*, for system characterization purposes, or more importantly, *run-time verification*, paving the way for safer embedded systems.

2 Observer Entity

Contemporary embedded computing platforms are often implemented in a single integrated circuit, being comprised of at least one processing element and varied peripherals for Input/Output (I/O) activities. Memory storing and supporting the executable software is external, being accessed through a memory controller. Such an architecture is commonly called System-on-a-Chip (SoC), due to having a complete embedded system in a single chip. A diagram depicting a generic SoC architecture for embedded systems is shown in Figure 1.

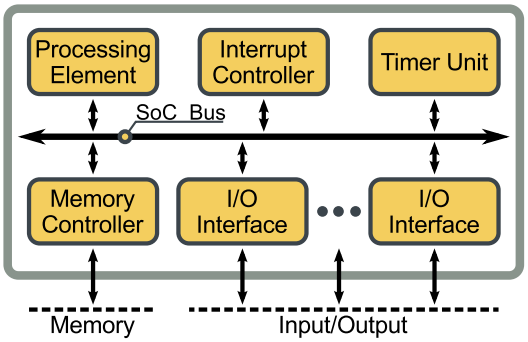


Fig. 1: Generic System-on-a-Chip Architecture

The SoC components are connected via an internal SoC bus. The most common SoC bus architectures [4, 2] share many architectural similarities, namely

memory-mapped access to components. A hardware mechanism attached to the SoC bus will have access to information pertaining to data exchange among the components of the SoC, including the execution of the software application. Therefore, a component providing non-intrusive monitoring mechanisms must be attached to the SoC bus, gathering information concerning the execution of the applications on the computing platform.

2.1 Architecture and Implementation

A non-intrusive Observer Entity (OE) [10] is the crux of the proposed RV system, exploiting the central nature of the SoC bus. The OE hardware mechanism synchronously monitors the SoC bus, comparing the values being exchanged in the bus with a set of configured observation points. Upon detection of an event, i.e. match between bus value and observation point, it time-stamps it and relays the pertaining information to an external system, e.g. an independent Personal Computer (PC) where the event-related information can be stored and processed. A block diagram depicting the OE architecture is shown in Figure 2.

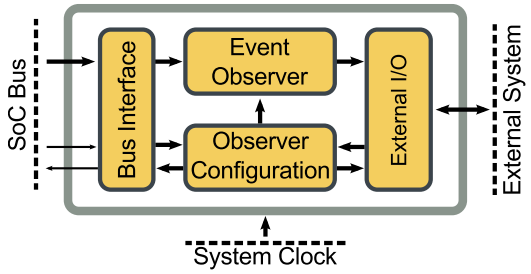


Fig. 2: Observer Entity Architecture

The OE architecture of Figure 2 is composed by several functional blocks: **Bus Interface**, allowing the OE to peek the bus activity and access the configuration via the bus (fallback method); **Event Observer**, which compares the current bus operations against a set of configured observation points stored in the **Observer Configuration**; an **External I/O Interface**, to send the detected event data to an external system and receive the observation points to be configured. For RV purposes, and in a standalone embedded system, the OE can alternatively be configured via software executing in the computing platform.

The OE is specified in VHDL[1], and embedded in a SoC system [3] with a LEON3 processor, a SPARCv8 [14] Central Processing Unit (CPU), embodying a state-of-the-art computing architecture. The LEON3 is the the reference computer architecture for european Space applications, e.g. satellites, being also

[1] **V**ery **H**igh-Speed Integrated Circuit **D**escription **L**anguage

used in other real-time control applications. The SoC bus is the Advanced Microcontroller Bus Architecture (AMBA) [4]. A block diagram with the global system is shown in Figure 3.

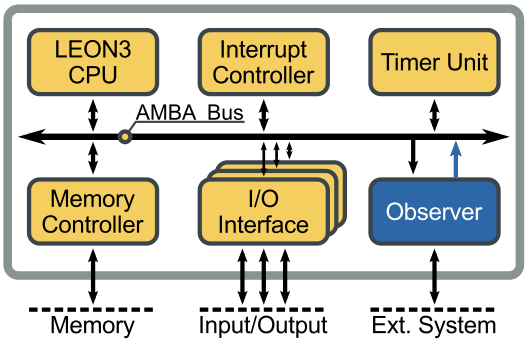


Fig. 3: LEON3 System-on-a-Chip with the Observer Entity

2.2 Operation

The operation of the OE for monitoring application execution is shown in Algorithm 1. The OE monitors the (AMBA) *Bus*, compares (*line 3*) the transfer operations $Bus.trf$ with a configured set of observation points, $Config$. Upon match, it sends a piece of information to the external system (*line 7*). This piece of information is an event *evt*, being comprised of: the time-stamp of the occurrence; the *id* of the event, specified in the configuration (*lines 5-6*).

Algorithm 1: Application Execution Event Monitoring

Input: System clock *hardware_clock_tick*
Output: Event *evt*

```
1 foreach hardware_clock_tick do
2     numTicks ← numTicks + 1
3     if newEvent(Bus) then
4         if ∃ id ∈ Config : Config[id] = Bus.trf then
5             evt.time ← numTicks
6             evt.id ← id
7             outputEvent(evt)
```

The *numTick* value (*line 2*) is incremented at every system clock tick, and used as the event time-stamp. This clock is drawn from the same source as the entire system under monitoring (`System Clock`), usually an external crystal oscillator, capturing the exact instant of occurrence of an event.

3 Observation Points

The exploitation of the Observer Entity (OE) requires its configuration with **observation points**. The extraction of observation points is motivated by the definition of *objects of interest*, such as function calls or variables, which will be monitored. The extraction method uses as working basis the application (program) file, which is a self-contained binary file, i.e. without any dependency on external files.

3.1 Anatomy of an Embedded Application

Embedded applications usually follow a similar development flow: **coding**, mapping a specification into source-code; **compiling**, translating the source-code into one or more machine-code object files; **linking**, merging the object files and external libraries into a self-contained binary file, without external dependencies and ready for execution. The remainder of the text assumes that:

- the compiler is the GNU C Compiler [7];
- the format of the binary file is Executable and Linkable Format (ELF) [15];
- the tools used are part of the GNU Binutils [6] package;
- the debug information format is DWARF [5];
- the target Instruction Set Architecture (ISA) is SPARCv8 [14];
- the target executes the instructions from RAM[2], i.e. at boot time everything is copied into RAM memory;
- there is no virtual memory.

Although these assumptions may seem too tight, they represent the majority of embedded applications, which are developed with standard-conforming tools, either open-source or commercial, and are used in current embedded architectures, such as the SPARC or ARM [12].

Binary File Structure and Organization The binary file contains the application itself and is divided in *sections*, the logical containers which are mapped into memory upon execution. Applications usually have three sections: `text`, containing the instructions (machine-code) to be executed; `data`, containing global and static variables; `bss`, a content-less section which defines the memory area for storing dynamic and uninitialized variables. There may also exist an additional section `rodata`, for read-only data, such as strings, whose contents can be merged together with the `text` section. The binary file contains these sections, together with information regarding how they should be handled by the loader. This includes information regarding how each section must be placed in memory, namely its initial memory address and its size (*see* Figure 4).

[2] Random Access Memory

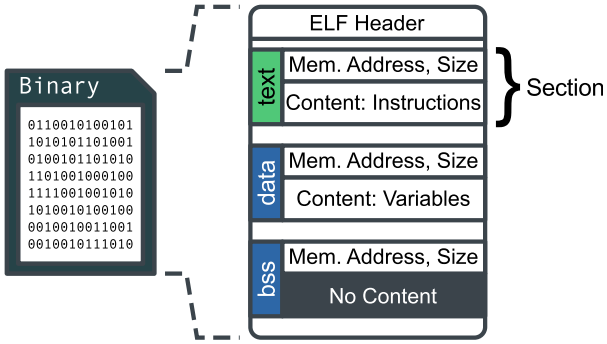


Fig. 4: Binary ELF File Structure

The location of the sections inside the file is defined in the ELF header. The linking process generates a list of *symbols*, which are unique names for referenced entities. Examples of symbols are: function names; global variables; labels, including the ones used as section delimiters. The mapping between symbol names and memory addresses is kept in a `symbol table`, another section contained in the binary file (not shown in Figure 4). Furthermore, the usage of debug switches upon compilation also creates sections with information usable by debuggers, e.g. `debug_info`, detailing the location of the local variables of each function in the stack and the data types of such variables.

Memory Organization and Execution The loading of the binary file into memory places the contents of the sections into *segments*[3]: `text` is placed in the *code* segment, `data` and `bss` are placed in the *data* segment. A *stack* area supports the dynamics of application execution, namely function invocation (*see* Figure 5).

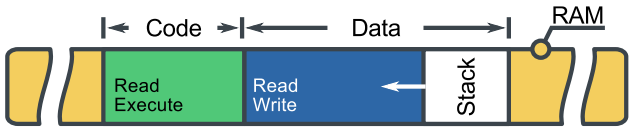


Fig. 5: Application Memory Structure

The stack starts at the end of the data segment, and grows towards the beginning of the data segment. Growth and shrinkage of the stack occur on function invocation/completion. This behavior, together with the function call conventions is defined in the Application Binary Interface (ABI) of the target architecture, e.g. the SPARC ABI [11].

[3] This designation comes from the ELF specification and is not necessarily related to the memory segmentation technique.

Local (function) variables are stored in the stack, and their location is dynamic w.r.t. absolute memory position. Upon the invocation of a function, space is created in the stack for storing the input parameters and local variables - a stack frame. The stack and its organization upon a function call is shown in Figure 6 for the SPARC architecture, following the ABI specification [11].

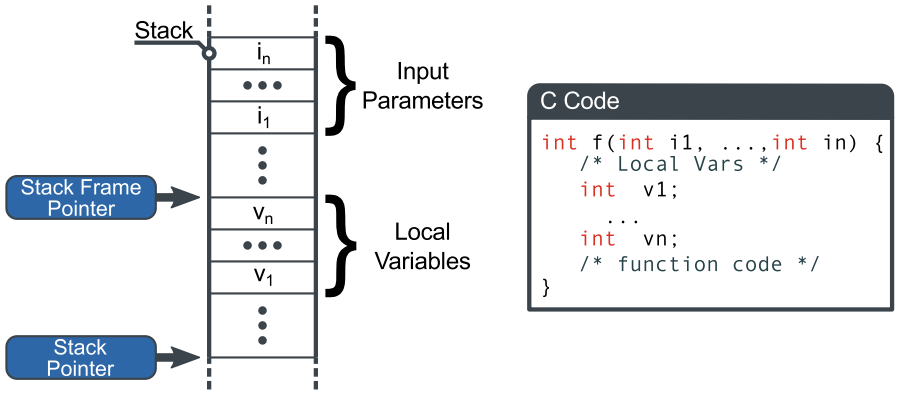


Fig. 6: Stack Organization upon Function Invocation

The `Stack Frame Pointer` is the delimiter of a stack frame, which is the local storage area of a function, used for local variable and parameter reference. Due to the volatility of function execution, (non-static) variables and parameters are always referenced as an offset of the stack frame pointer, thus there is no static reference to them.

3.2 Observation Point Extraction

The objective of extracting observation points from the binary file is to generate a configuration file for the OE machinery. The extraction relies chiefly on the table of symbols, both for monitoring entities which are present there - function calls, global and static variables - but also to derive the (dynamic) address of local variables. An excerpt from the map of symbols of a binary application file is presented in Figure 7.

The interpretation of the symbol map of Figure 7 is the following: *Name* is the name of the symbol; *Value* is the memory address of the symbol; *Class* can be **T**ext, **D**ata, **B**ss; *Type* is function (`FUNC`), variable (`OBJECT`) or a label (`NOTYPE`), e.g. a section delimiter; **Size** is the ammount of space used, applicable to functions and variables; **Section** where the symbol is defined/resides.

In Figure 7 there are five symbols represented: functions `f`, `atoi` and `main`, all residing in the *text* section; global variable `global_var`, in the *data* section; the *bss* section start delimiter, `__start_bss` in the *bss* section. For the purpose of

```
Name                Value   Class        Type          Size      Section
f                  |40001924|   T  |              FUNC|00000058| .text
main               |4000197c|   T  |              FUNC|0000002c| .text
atoi               |40002140|   T  |              FUNC|0000001c| .text
global_var         |4000606c|   D  |            OBJECT|00000004| .data
__bss_start        |40006bc0|   B  |            NOTYPE|        | .bss
```

Fig. 7: Map of Symbols of a C Application

monitoring, the columns of interest are: **Value**, informing of the absolute memory address of the symbol; **Size**, for variable monitoring - both for global/static and local variables.

The extraction of observation points for function execution and global/static variable is straightforward, from the symbol map. However, the observation points for local variables are dynamic (*see* Figure 6), and are not present in the symbol map. The extraction of these observation points is based on the offset (displacement) of the variable w.r.t. the stack frame pointer. Such displacement is constant across function invocations, but the stack frame pointer value may not be. Nevertheless, the local variable offset can be easily acquired by resorting to the debug symbols present in the `debug_info` of the ELF binary application file. The `debug_info` section provides for each function auxiliary data, in the DWARF format, comprising the variable location offset w.r.t. the stack frame pointer, and the data type (size) of the variable(s). This is the preferred way, but it assumes the source code has been compiled with the debug switches enabled.

Alternatively, given the existence of a binary file with a `symbol table` section, it is also possible (to some extent) to extract the the observation points, using the disassembly operations. This method however, it is much harder given the lack of structured information, as opposed to the debug symbol method.

3.3 Observation Point Configuration

The extraction of observation points is used to generate a configuration file for the OE. The configuration of an observation point uses the following parameters:

- Memory Address
- Read/Write (relevant for variables)
- Unique ID

The configuration file is read by an application which communicates with the OE via the external interface. An example of such file is shown in Figure 8.

In the example of Figure 8, where there are four observation points. The syntax of the configuration file is the following: the `Address` field contains the address to be monitored; the `ID` field is the unique ID to be given to events originating on that address; the `R/W` is the read/write direction of the event, i.e. if

```
# This is a comment
#Address    ID    R/W  Type  Size  Displ   FSize
40002140     0      0     0
4000606c     1      1     1     4
40001924     0      0     2     4      4      58
```

Fig. 8: Observer Entity Configuration File

it should be generated on a read, write or both; the `Type` identifies the object type to be monitored: instruction; global/static variables; local variables of a function. These four fields are required for instruction monitoring. For variable monitoring, additional fields are needed: `Size`, in bytes, for capturing the variable's value; and in the case of local variables the displacement `Displ` of the variable relative to the stack frame pointer and the function size `FSize`, for the OE to know when the execution is being performed inside the function.

For example, in the third entry of Figure 8 is exemplified the monitoring of a variable local to a function with entry point located at address `0x40001924`. The variable to be monitored has an offset of 4, which must be subtracted from the actual value of the stack frame pointer (stored in the corresponding CPU register), in order to obtain the memory address to be monitored.

The process of extracting the observation points and creating a configuration file for the OE is integrated in a work flow, and takes as input: *binary application file*, with the application; *object of interest list*, with the functions and variables to be monitored. The work flow is depicted in Figure 9.

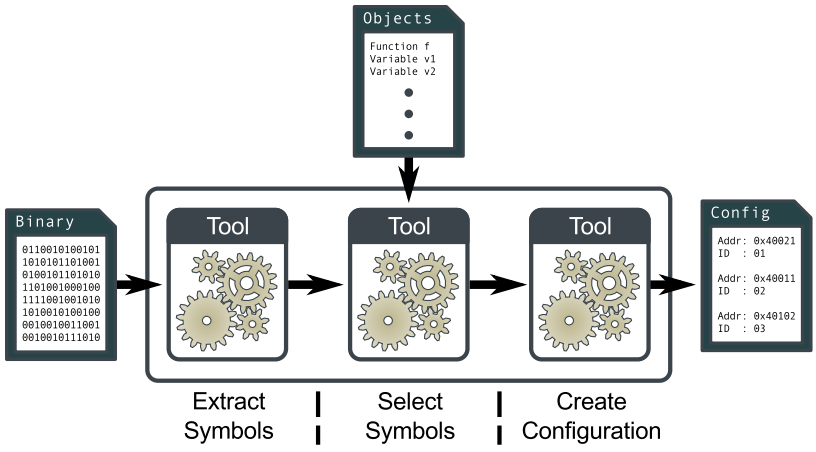


Fig. 9: Observation Point Extraction Flow for Symbol-based Monitoring

The tools referred in Figure 9 are: GNU Binutils tools, such as `nm`, for symbol extraction; standard Unix tools, such as `grep`, for symbol selection and `awk` for formatting the selected symbols into a configuration file to be loaded into the OE. All these are wrapped in a shell script. However, these can be invoked from an especially crafted tool.

3.4 Deployment

The final step towards having observation data is the deployment of both the binary and configuration file with the observation points into the SoC, enhanced with the OE machinery. The SoC is implemented in a reconfigurable hardware device, part of a prototyping board, thus representing a reference embedded system architecture (*see* Figure 10). The operation of the system will generate monitoring data regarding the configured observation points. An external system can receive the monitoring data, for *offline* exploitation purposes.

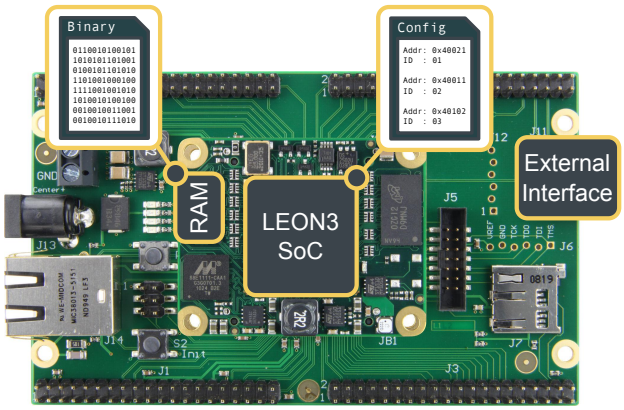


Fig. 10: Reference Reconfigurable-based Embedded System used in Prototyping

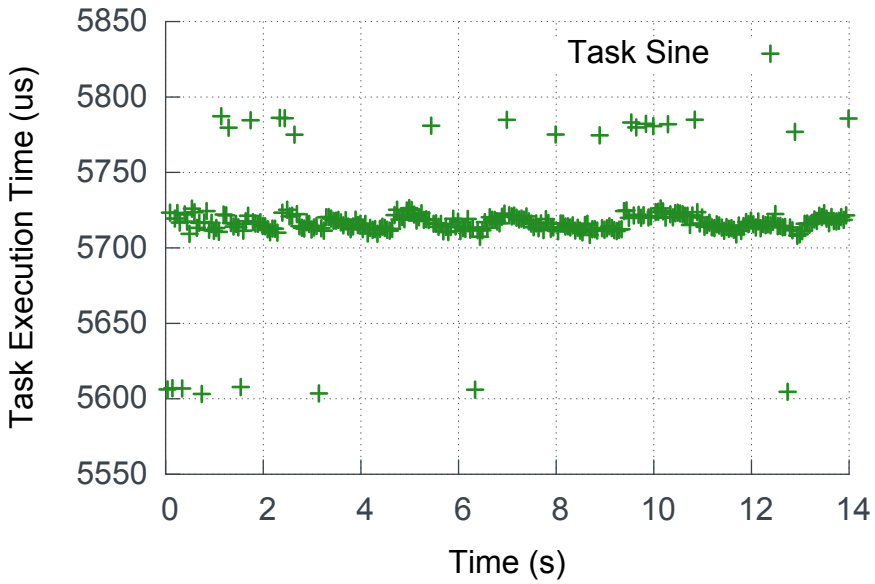
4 Exploitation of Non-intrusive Monitoring Data

The exploitation of the monitoring data created by the OE can be split in four categories: **Performance Evaluation and Timeliness Profiling**, to assess the performance and timeliness properties of the embedded system via another system external to the SoC platform, e.g. a Personal Computer; **Scheduling Analysis** enabling the analysis of a given task set schedule as provided by analytic tools; **WCET Measurement**, assessing the correctness of the WCET estimate obtained from external tools; **Run-time Verification**, to assess the correctness of the execution at run-time, by the real-time embedded application.

4.1 Performance Evaluation and Timeliness Profiling

The transfer of the monitoring data to an external system enables the posterior *offline* analysis of such data. One of the envisaged uses for the observation infrastructure is the evaluation of performance and timeliness properties of an embedded system, e.g. system under different operating conditions.

An example is presented next, based on an application running on the Real-Time Executive for Multiprocessor Systems (RTEMS) Real-Time Operating System (RTOS) [1]. The system under evaluation is composed by a task, named `Task Sine`, which produces the sample values of a sine wave with a given frequency. The task is executed periodically, with a 50 *ms* period. The monitoring aims at measuring the *execution time of the task*, under several load conditions. The monitoring data is collected over a period of 14 seconds. In the first experiment the system is lightly loaded. This data is represented in a graphical manner through Figure 11, together with a table containing its statistical analysis.

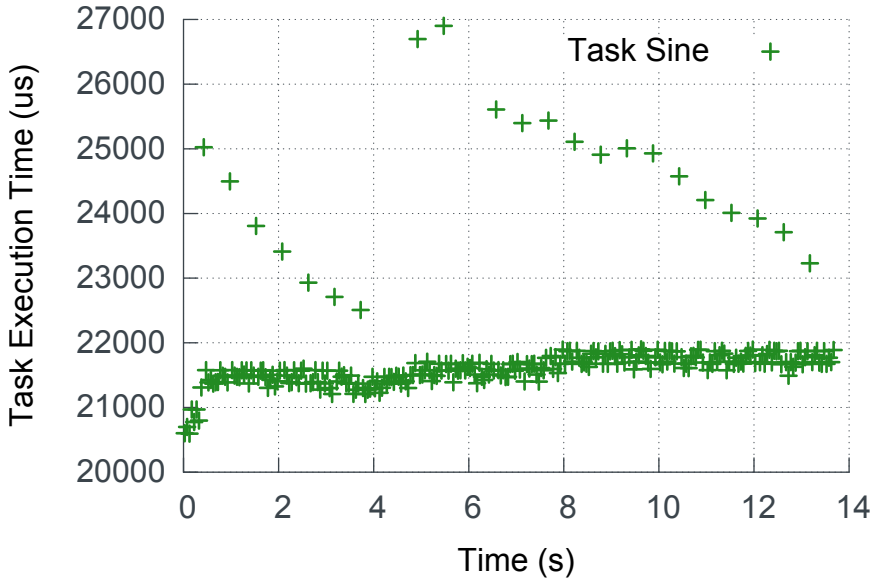


Minimum (μs)	Maximum (μs)	Average (μs)	Std. Deviation
5603.180	5787.240	5719.536	22.686

Fig. 11: Real-time Task Execution Time Measurement in a Lightly-loaded System

The null competition for the processing resources allows `Task Sine` to exhibit a somewhat stable execution time, i.e. with low variance.

Another experiment was performed on the same system, but with a heavily competing higher-priority aperiodic task, with the intent of monitoring a heavily-loaded system. The results are shown in Figure 12.



Minimum (μs)	Maximum (μs)	Average (μs)	Std. Deviation
20596.660	26901.980	21819.251	878.484

Fig. 12: Real-time Task Execution Time Measurement in a Heavily-loaded System

In this case, it is clear that there are disturbances, given the highly irregular shape of the graphical display of the monitored data. These stem from the interference caused by the execution of the aperiodic task to the execution of the Task Sine, causing several interruptions (preemptions) of Task Sine. Furthermore, it can be noticed that the execution times are larger by one order of magnitude; however, the maximum execution time recorded (*26.9 ms*) is still lower than the deadline of the `Task Sine` (*50 ms*).

4.2 Scheduling Analysis

Another facet of timeliness profiling is given by task scheduling analysis in a real-time system. Given a set of real-time tasks, with certain timeliness requirements, it is necessary to assess if a system can be built satisfying every task timeliness requirements, i.e. if the task set is schedulable.

Cheddar [13] is a tool providing schedulability analysis for a set of real-time tasks, which is also able to output the resulting schedule. The OE can be exploited to perform experimental evaluation of the analytic results provided by Cheddar. Moreover, the experimental results can be filtered and fed to Grasp [9], a tool enabling visualization of real-time systems' execution, namely task execution and switching. Such a chain would enable a work flow for designing and validating task schedule, combining the analytic results of a tool with the visualization capabilities of another.

4.3 WCET Measurement

Yet another facet of timeliness profiling is the verification of the WCET of a given task set, being a particularly important part in the design-cycle of a real-time embedded system. The existence of data regarding the timeliness profiling of the deployed system can be exploited to provide insight into WCET characterization, and assist the activities in the design-cycle.

Usually the WCET values provided by the estimation tools are pessimistic, and can be off the real task execution values by an order of magnitude [8]. The monitoring of real task execution values allows to get a more realistic estimation of the WCET value, which can then be fed into the schedulability analysis tools.

4.4 Run-time Verification

An extremely useful exploitation of the non-intrusive monitoring data is feeding verification techniques, in order to achieve RV. Given that embedded systems often need to interact with the physical environment, which is uncertain by nature, the usage of RV is a valuable asset to enable adaptive behaviour of embedded applications.

For example, the control system of an unmanned aerial vehicle (*drone*) may be designed to provide thrust until the aircraft has reached a certain altitude. Both variables - thrust and altitude - can be monitored in a non-intrusive way. Their values can then be fed into a RV mechanism, which would be coupled with the fault detection, isolation and recovery mechanisms of the control system. Such an approach improves on the current state-of-the-art by its non-intrusiveness and flexibility.

5 Conclusion and Future Work

The availability of a non-intrusive monitoring infrastructure can enrich an embedded application at all stages of its life-cycle, by allowing the collection of execution data from the system. Such data can be exploited to assess properties pertaining to system performance and timeliness; scheduling analysis and Worst-Case Execution Time (WCET) measurement; and serve as a supporting basis for Run-time Verification (RV) techniques.

This paper has shown the work flow for exploiting an hardware-based Observer Entity (OE), embedded in a System-on-a-Chip (SoC) architecture. The crux of the OE configuration is the definition and extraction of observation points, by combining the information contained in the binary application file with a list of objects of interest to be monitored. A tool-based approach enables an effective work flow that culminates in the generation of a configuration file for the OE machinery.

A line for future work points towards the refinement of the interaction between tools for observation point extraction. Such refinement aims at streamlining the whole process.

The main focus of future work, however, is researching effective RV for real-time embedded systems. Such techniques can benefit greatly with the formalisms brought by the formal languages, namely Temporal Logics, which can be useful tools to model the behaviour of embedded applications, and therefore used in effective RV mechanisms.

References

1. RTEMS: Real-Time Executive for Multiprocessor Systems. http://www.rtems.org
2. The CoreConnect Bus Architecture. White Paper (Jan 1999)
3. Aeroflex Gaisler A.B.: GRLIB IP Library Users Manual (Apr 2014), http://gaisler.com/products/grlib/grlib.pdf
4. ARM Limited: AMBA Specification (May 1999)
5. DWARF Debugging Information Format Committee: DWARF Debugging Information Format (2010)
6. Free Software Foundation: GNU Binutils, https://www.gnu.org/software/binutils/
7. Free Software Foundation: GNU Compiler Collection, http://gcc.gnu.org/
8. Garrido, J., Zamorano, J., de la Puente, J.A.: Static Analysis of WCET in a Satellite Software Subsystem. In: OASIcs-OpenAccess Series in Informatics. vol. 30. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
9. Holenderski, M., van den Heuvel, M., Bril, R.J., Lukkien, J.J.: 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). pp. 37–42 (2010)
10. Pinto, R.C., Rufino, J.: Towards Non-invasive Run-time Verification of Real-time Systems. In: Proceedings of the 2014 European Conference on Real-time System - Work in Progress Session. ECRTS '14, Euromicro (2014)
11. SCO Inc.: System V Application Binary Interface (1998)
12. Seal, D.: ARM Architecture Reference Manual. Pearson Education (2001)
13. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a Flexible Real Time Scheduling Framework. In: ACM SIGAda Ada Letters. vol. 24, pp. 1–8. ACM (2004)
14. SPARC International Inc.: The SPARC Architecture Manual (1992)
15. TIS Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (1995)