Hardware Support to Non-intrusive Runtime Verification on Processor Technologies*

José Rufino

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal; email: jmrufino@ciencias.ulisboa.pt

Abstract

Software-based instrumentation probes always disturbs the functional and non-functional properties of a system, even if in a minimal way. To avoid the disturbance of system operation, by instrumentation probes, nonintrusive runtime verification must rely on hardware-based technology. This paper reviews classical processor technology to understand which kind of support is provided on each processor family, its intrusiveness, functionality and offered system support.

Keywords: Hardware-based runtime verification

1 Introduction

The traditional approach to runtime verification is to instrument the software of a functional system with small pieces of code that, acting as observers, assess the software state in runtime. Software-based instrumentation inherently disturbs the functional or non-functional properties of a system, namely with respect to timing properties, which are crucial to embedded and real-time system design [1, 2, 3]. They always exhibit some degree of intrusiveness, even if minimal.

Software-based observing components affect the normal behaviour of the observed system, throughout what is called “the observer effect” or “the probe effect” [4]. The delays implicitly associated with the insertion of software-based probes may affect the timing characteristics of concurrent programs. The removal of such probes from the software, which will lead to shorter program/task execution times, may render a given task set unschedulable, due to changes in the corresponding cache-miss profile [5, 6, 7].

Hardware-based approaches are inherently non-intrusive, i.e. they do not affect system operation. Though hardware-based observation is in essence non-intrusive, monitoring functions, i.e. runtime verification (RV) may have some degree of intrusiveness. Non-intrusiveness, may then be referred to as a RV constraint. RV constraints are not only relevant, but in fact fundamental, for highly critical systems [2].

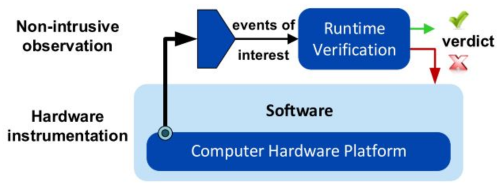


Figure 1: Non-intrusive runtime verification

This paper reviews classical processor technology to understand which kind of support is provided on each processor family, its intrusiveness, functionality and, in general, offered system support.

A comprehensive overview of various hardware (including on-chip), software and hybrid (i.e., a combination of hardware and software) methodologies for system observation and verification of software execution in runtime is provided in [8]. System observing solutions can be designed to be directly connected to some form of system bus, enabling information gathering regarding events of interest, such as data transfers and signalling taking place inside the computing platform, namely instruction fetch, memory read/write cycles and interrupt requests, with no required changes on the target system's architecture, as shown in the diagram of Figure 1. Examples of such kind of hardware-based observation approaches are proposed in [9, 10, 11, 12, 13].

The paper is organized as follows. Section 2 presents a description of the previous related work. Section 3 reviews the classical processor technology looking for non-intrusive runtime verification support. Section 4 describes the evaluation experiment for a particular processor technology (SPARC LEON) and, finally, Section 5 presents some concluding remarks and future research directions.

2 Previous Work

The application of non-intrusive runtime monitoring to embedded systems has been discussed in [8,14] and, more specifically, in safety critical environments [13].

Configurable minimally intrusive event-based frameworks for dynamically runtime monitoring was developed in [15], which was later complemented with a combination of hardware and software observability [3].

Additionally, the RV concept has been applied to autonomous systems [16] and to a AUTOSAR-like real-time operating

*This work was partially supported by FCT, through funding of LASIGE Research Unit, ref. UID/CEC/00408/2013, and by FCT/CAMPUS FRANCE (PHC PESSOA programme), through the transnational cooperation project 3732 (Portugal) / 37932TF (France), Non-intrusive Observation and Run-Time verification of cyber-pHysical systems (NORTH). This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI), supported by COST (European Cooperation in Science and Technology).

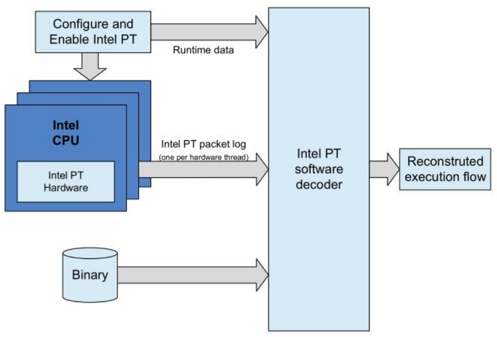

Configure and Enable Intel PT
Runtime data
Intel CPU
Intel PT Hardware
Intel PT packet log (one per hardware thread)
Intel PT software decoder
Reconstruted execution flow
Binary

Figure 2: Intel processor trace (Intel PT)

system aiming the automotive domain [17]. A runtime monitoring approach for autonomous vehicle systems requiring no code instrumentation by observing the network state is described in [18].

High quality trace data in a multi-core environment uses an approach based on non-intrusive full observation, meaning not only the program counter, but also other data read/write cycles, cache and bus operations are included in the trace [9].

A set of first contributions and discussion of technical issues such as metadata management, format and storage on practical examples are addressed in [19]. A description of the fundamentals of a trace are presented in [20].

3 Processor Technology

This section reviews different processor families to determine the support they provide, its intrusiveness and functionality.

3.1 Intel: Processor Trace

The Intel processor trace (PT) [21] is an extension of the Intel Architecture that captures information concerned with software execution, on each hardware thread, using dedicated hardware facilities. So, when an execution completes some special-purpose software can do processing of the captured trace data and reconstruct the exact program flow (Figure 2). Intel PT has an execution overhead cost: though a target less than 5% overhead is desirable, there are some applications with 35% overhead, being 20% an average value.

The captured information is collected in data packets, as described in [22] and summarized next. A set of packets (Packet Stream Boundary, PSB and Paging Information Packet, PIP), act as heartbeats generated at regular intervals (every 4 KiB) and record changes in attributing a linear address to an application. The MODE packet provides the decoder relevant execution information for binary interpretation and trace log and the Overflow (OVF) packet is issued when a processor experiences an internal buffer overflow. Three different packets, ranging different precisions, are used to get time information: Timestamp Counter (TSC) which provides some portion of a software-visible timestamp counter; Mini Timestamp Counter (MTC) which is more frequent and used with TSC to get accurate timestamps for less cost; Cycle Counter (CYC) packets provide even finer grain timestamp information. The Core Bus Ratio (CBR) contains the core bus clock ratio.

In a control flow tracing context, the following packets are used: Taken Not-Taken (TNT) tracks the direction of conditional branches (taken or not taken); Target IP (TIP) record the target value of the IP (Instruction Pointer) register in indirect branches; Flow Update Packet (FUP) provide the value of the IP for asynchronous events (interrupt and exception).

Each packet of the trace output is written to memory in a collection of variable-sized regions of physical memory. Therefore, with the knowledge of binary information, one can reconstruct the entire control flow of the original software, together with the precise timing of each branch.

Since the decoding of the traces is "several orders of magnitude slower than tracing", one may think a proprietary design where the Intel PT decoder memory area is set as a dual-port memory device, thus providing independence and allows non-intrusive runtime verification. However, these schemes are very specialised.

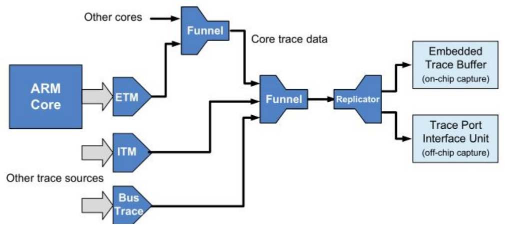


Figure 3: ARM CoreSight

3.2 ARM: CoreSight Technology

The next system we analyse is based on the ARM technology and its non-intrusive observation scheme, generically known as ARM CoreSight [23, 24, 25].

The architecture of ARM CoreSight is represented in Figure 3. The simplest form of trace is that generated by the software executing on the cores. Optimizations on this approach allow writing to the ARM Instrumentation Trace Macrocell (ITM), which streams the trace data direct to a trace buffer, as shown in Figure 3. This provides a high bandwidth channel that allows the delivery of more instrumentation points. However, the drawback of this approach is its natural intrusiveness.

To avoid instrumentation, hardware trace is an option, materialized by the ARM Embedded Trace Macrocell (ETM), is extremely popular. As shown in Figure 3, there is one ARM ETM for each core. In hardware trace, special-purpose logic watches the address, data and control signals within the System-on-Chip (SoC) compresses that information and emits to a trace buffer, which itself can be subdivided in to three main categories: program/instruction trace; data trace; and bus (or interconnect fabric) trace. The ARM ETM is thus a non-intrusive observer.

In terms of cost, for program/instruction trace macrocells can be quite small: only one byte/instruction/processor is required. Unfortunately, the cost of implementing data trace

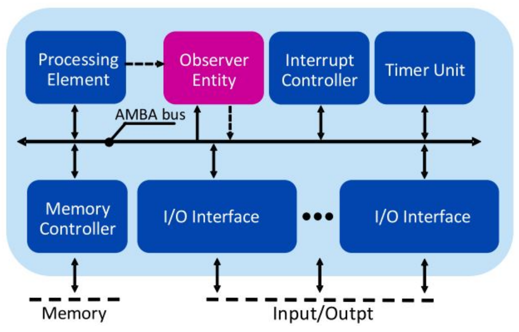


Figure 4: SPARC LEON processor and observer entity

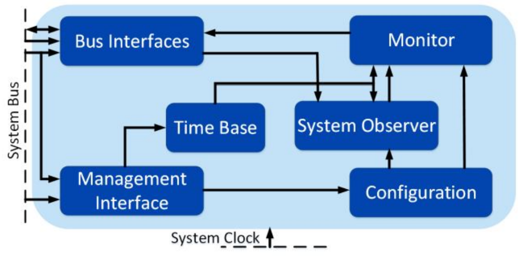


Figure 5: Observer Entity Architecture

is highest: trace macrocells need to be larger, data is more difficult to compress (data trace from an ARM ETM typically requires 1-2 bytes/instruction/processor). Each captured trace data have attached a timestamp.

The collected data is replicated and presented in two different resources: an internal (on-chip) embedded trace buffer; a trace port allowing the captured data to be externally processed.

3.3 SPARC LEON: Dedicated Observer

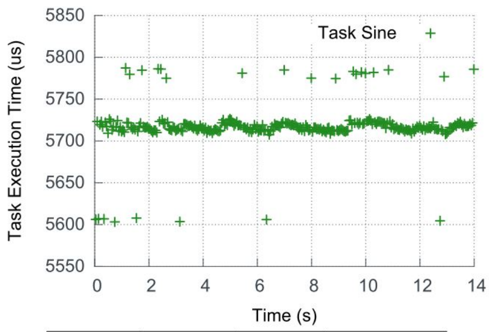
The next system we analyse is embedded in a SoC system with a LEON processor [26], a SPARC CPU [27], embodying a state-of-the-art computing architecture. The LEON is the reference architecture for European Space applications, e.g. satellites, being also used in other real-time control applications. The SoC bus is the AMBA bus [28]. A block diagram with the global system is presented in Figure 4.

Since SPARC LEON does not have specific tools for code observation and tracing, one have designed one (also shown in Figure 4). The Observer Entity (OE) infrastructure can observe the AMBA bus and capture a set of relevant events: instruction fetch; memory read/write cycles; interrupt requests. Alternatively, the OE can be plugged in a cache internal bus, for a more precise observation.

The OE is specified in VHDL[2] and the event capture is independent and made in parallel with the operation of the functional system. Therefore, the OE integrates all the mechanisms required for a non-intrusive observation. The monitor option supports non-intrusive runtime verification.

The OE comprises the modules of Figure 5: Bus Interfaces, capturing all physical bus activity, such as bus transfers or

[2]Very High-Speed Integrated Description Language.



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

Figure 6: Task Execution Time Measurement

interrupt vectors; Management Interface, enabling observer entity configuration; Configuration, storing a dynamically defined set of events; the System Observer itself, detecting events of interest; Monitor, which detects possible violations to the specified system behaviour; Time Base, which allows to time stamp the events of interest.

4 Evaluation

An example of a runtime monitoring function is presented next, assuming the use of a SPARC LEON processor; as software counterpart an application running on the RTEMS real-time operating system is used [29]. The software system under evaluation is composed by a task, named Task Sine, which produces 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 as well as its amplitude. Both the execution time and the amplitude are monitored. This data is represented in a graphical manner through Figure 6, together with a table containing its statistical analysis. The null competition for the processing resources allows Task Sine to exhibit a somewhat stable execution time, i.e. with low variance. In this experiment, given the monitoring bounds, no error is detected. This will not be the case if the monitoring values have a lower bound.

5 Conclusion

This paper reviews classical processor technology to understand which kind of support is provided on each processor family (Intel, ARM and SPARC LEON), its intrusiveness, functionality and offered system support.

Each processor family was reviewed and we characterize the offered support to observation. Together with this, we address the non-intrusiveness and functionality.

For the SPARC LEON, which received a freshly designed non-intrusive runtime verification scheme, we have conducted a very simple experiment that evaluate the proposal.

References

[1] M. E. Shobaki and L. Lindh (2001), *A hardware and software monitor for high-level system-on-chip verification*, in Proceedings of the 2nd IEEE International Symposium on Quality Electronic Design (ISQED 2001), (San Jose, CA, USA), pp. 56–61.

[2] L. Pike, S. Niller, and N. Wegmann (2011), *Runtime verification for ultra-critical systems*, in 2nd International Conference on Runtime Verification (RV 2011), (San Francisco, USA), pp. 310–324, Springer.

[3] J. C. Lee and R. Lysecky (2015), *System-level observation framework for non-intrusive runtime monitoring of embedded systems*, ACM Transactions on Design Automation of Electronic Systems, vol. 20, no. 42.

[4] J. Gait (1986), *A probe effect in concurrent programs*, Software - Practise and Experience, vol. 16.

[5] T. Lundqvist and P. Stenstrom (1999), *Timing anomalies in dynamically scheduled microprocessors*, in Proc. of the 20th Real-Time Systems Symposium, IEEE, Dec. 1999.

[6] R. Wilhelm et al (2008), *The worst-case execution time problem - overview of methods and survey of tools*, ACM Transactions on Embedded Computing Systems (TECS), vol. 7, Apr. 2008.

[7] T. H. Nam (2017), *Cache Memory Aware Priority Assignment and Scheduling Simulation of Real-Time Embedded Systems*, PhD thesis, Université de Bretagne Occidentale, Brest, France.

[8] C. Watterson and D. Heffernan (2007), *Runtime verification and monitoring of embedded systems*, IET software, vol. 1, pp. 172–179.

[9] R. Backasch, C. Hockberger, A. Weiss, M. Leucker, and R. Lasslop (2013), *Runtime verification for multicore SoC with high-quality trace data*, ACM Trans. on Design Automation of Electronic Systems, vol. 18.

[10] R. C. Pinto and J. Rufino (2014), *Towards non-invasive runtime verification of real-time systems*, in Proc. 26th Euromicro Conf. on Real-Time Systems - WIP Session, (Madrid, Spain), pp. 25–28, Euromicro.

[11] T. Reinbacher, M. Fugger, and J. Brauer (2014), *Runtime verification of embedded real-time systems*, Formal Methods in System Design, vol. 24, pp. 203–239.

[12] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu (2008), *Hardware runtime monitoring for dependable cotsbased real-time embedded systems*, in Proceedings of the Real-Time Systems Symposium (RTSS 2008), (Barcelona, Spain), pp. 481–491, IEEE.

[13] A. Kane, O. Chowdhury, A. Datta, and P. Koopman (2015), *A case study on runtime monitoring of an autonomous research vehicle (ARV) system*, in Proc. 15th Int. Conf. on Runtime Verification, vol. 9333 of LNCS, (Vienna, Austria), pp. 102–117, Springer.

[14] T. Reinbacher, K. Y. Rozier, and J. Schumann (2014), *Temporal-logic based runtime observer pairs for system health management of real-time systems*, in Proc. 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 8413 of LNCS, (Grenoble, France), pp. 357–372, Springer.

[15] J. C. Lee, A. S. Gardner, and R. Lysecky (2011), *Hardware observability framework for minimally intrusive online monitoring of embedded systems*, in Proc. 18th Int. Conf. on Engineering of Computer Based Systems, (Las Vegas, USA), pp. 52–60, IEEE.

[16] G. Callow, G. Watson, and R. Kalawsky (2010), *System modelling for run-time verification and validation of autonomous systems*, in Proc. 5th Int. Conf. on System of Systems Engineering, (Loughborough, UK).

[17] S. Cotard, S. Faucou, J.-L. Bechennec, A. Queudet, and Y. Trinquet (2012), *A data flow monitoring service based on runtime verification for AUTOSAR*, in Proceedings of the 14th Int. Conf. on High Performance Computing and Communications, (Liverpol, UK), IEEE.

[18] A. Kane (2015), *Runtime Monitoring for Safety-Critical Embedded Systems*, PhD thesis, Carnegie Mellon University, USA.

[19] S. Jaksic, M. Leucker, D. Li, and V. Stolz (2017), *CO-EMS open traces from the industry*, in Proc. of Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RVCuBES 2017), vol. 3, (Seatle, USA.).

[20] G. Reger and K. Havelund (2016), *“What is a trace? a runtime verification perspective*, in Proc. of 7th Int. ISoLA 2016 - Leveraging Applications of Formal Methods, Verification and Validation (T. Margaria and B. Steffen, eds.), vol. LNCS 9953, (Corfu, Greece).

[21] J. Reinders (2013), *Intel Processor Tracing*, Intel Corporation, Sept. 2013.

[22] Intel (2013), *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-017 ed.

[23] W. Orme (2008), *Debug and trace for multicore SoCs*, ARM White paper.

[24] ARM 2013, *CoreSight technical introduction: a quickstart for designers*, ARM White paper EPM-039795.

[25] ARM (2013),*ARM CoreSight Architecture Specification*, 2.0 ed.

[26] Aeroflex Gaisler A.B. (2014), *GRLIB IP Library User’s Manual.*

[27] SPARC International Inc. (1992), *The SPARC Architecture Manual.*

[28] ARM Limited (1999), *AMBATM Specification.*

[29] The RTEMS Project (2017), *RTEMS: Real-Time Executive for Multiprocessor Systems - User Manual*, release 5.0.0 (master) ed. http://www.rtems.org.