A Real-Time System Monitoring driven by Scheduling Analysis

Stéphane Rubini, Valérie-Anne Nicolas, Frank Singhoff
Lab-STICC UMR 6285, UBO, UBL, Av. Le Gorgeu, 29200 Brest, France; email: {surname.name}@univ-brest.fr
José Rufino
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal; email: jmrufino@ciencias.ulisboa.pt

Abstract

Real-Time system engineers may introduce task scheduling analysis at the early stage of the design process. System temporal behavior and task schedules are strongly related. The noncompliance to an expected schedule is a symptom of an erroneous state that may result in a serious risk for the system integrity. Spreading the design task model, as a timing reference to guide run-time verification, is a kind of extension to the model driven design paradigm.

This paper presents the overall architecture of a non-intrusive hybrid monitor. Configured by the result of a scheduling simulation, the monitor is intended to observe the system execution and raised an alarm in case of divergence with the predicted schedule. To advance this goal, a first experiment shows the scheduling of 2 tasks rebuilt from the events collected by the monitor while the RTEMS OS' *scheduler was executing.*

Keywords: health monitoring, real-time system, scheduling analysis.

1 Introduction

In real-time systems, the scheduling is a set of rules that govern the order of the processing on the system's hardware resources. Beyond the need to provide program codes functionally corrects, the designers must also ensure timeliness of theirs results. To fulfill timing requirements of real-time systems, the scheduling of the tasks must be taken into account at the early stages of the design. Scheduling analysis works from an abstract view of tasks, the task model, which defines their timing behavior independently of the nature of computation they have to do.

We propose to extend the scheduling simulation field of use onto run-time verification of hard real-time systems. Hard real-time systems are characterized by deterministic execution and strict time constraints. From a given task model that specifies the timing parameters to enforce, the analysis tools verify their respect during the design step. They also define a deterministic awaited execution trace of the system tasks at run-time.

A *health monitor* can observe at run-time the sequence of events that describes the evolution of the task states from the schedule point of view, and compare them to those predicted by the simulation. Our goal is to configure the monitor from the task models, and then to use a unified specification from the design of the system to its run-time supervision. This paper focuses on the overall hardware architecture of the monitor. We evaluate its ability to collect and report a trace of scheduling events observed on a target system. Only an initial and restricted version of the scheduling comparison module is presented here as a more complete implementation remains to be developed.

The outline of the paper is the following. The first part characterizes the task models we want to monitor. Next, the architecture of the health monitor is described. Some technical problems about the rebuilding of long term time stamps is emphasized. Section 3 presents the status of the project, and the results of the initial experiments. The last part of the article describes some related works and concludes.

2 Task model and run-time verification

The systems we are targeting for run-time monitoring are time-triggered hard real-time systems on uni-processor execution platform, like the systems that control the critical functions in transport vehicles. The number of tasks and their parameters (e.g. deadlines, release times) are fixed and specified at design time. Tasks are periodic and the complete system itself has a repetitive temporal behavior, eventually achieved after a stabilization time that can be determined by scheduling analysis. We observe such a stabilization phase when the initial release time is not the same for all tasks (i.e. the offset parameter of some tasks is different from 0). The scheduling of the tasks, computed off-line or by static schedulers, must be deterministic.

From the above assumptions, the scheduling trace produced by a scheduling analysis tool from a given task model may serve as a "golden reference" for the verification of a system also implementing this task model. Fig. 1 sums up this approach.

The main interest to do monitoring at the schedule level is the restricted number of event types to observe and their common semantics on multiple systems. As opposed to the operations

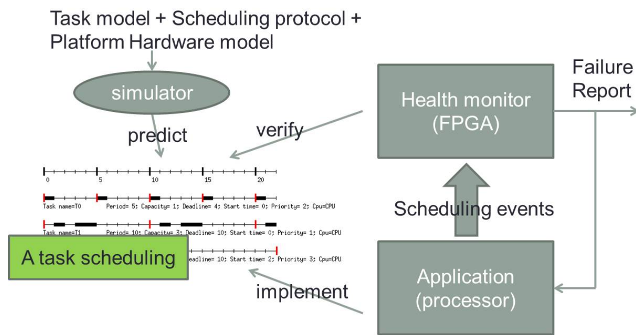

Figure 1: Scheduling analysis as "golden" reference.

performed by the tasks which are generally different for each application, scheduling concepts remain similar.

A challenge of the implementation of this approach is to define how to manage discrepancies between the task model specification, the simulation results and the real execution on the target platform. For instance, the release of a set of tasks can be stated as simultaneous for the scheduling while related events are emitted and detected in a sequential order. The matching of the physical time as approximated in the observed system, in the monitor and in the result of simulation constitutes another example of practical problems that need to be solved.

The first step of the approach is based on a scheduling simulation tool. Our team has already developed such a tool, that is called Cheddar [1]. The second step is to have a monitor which allows us to observe the real-time system by inducing a weak perturbation. The next section presents the architecture of this health monitor, and gives details about the implementation of some of its functions.

3 Monitor Architecture

The monitor verifies that the trace of observed events is conform to the scheduling simulation predictions. Hence, there is no need for reporting the events if the system works as expected.

However, more meaningful information is how the system goes into an erroneous state, that is, from the monitor observation, what is the preceding sequence of events before the failure. This working mode will be named in the sequel *back trace mode*. But, the analysis of the consequence of an error can also be another outcome of the monitor report. In this second case, the monitor switches in a *forward trace mode*, which transfers all captured events to the supervision station.

So, the hardware monitor we are developing is structured following the previous objectives: run-time verification of the scheduling, and reporting of the cause or the consequences of a discrepancy. Fig. 2 shows the 5 main components of the its architecture and their interactions.

The "event capture" component is in charge of events collect, either by observing the behavior of the monitored system from an external point of view, or by receiving the event explicitly transmitted to it. A time stamp is adjunct to each

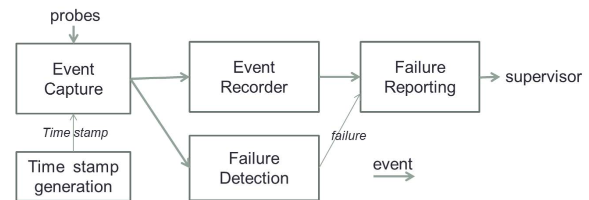

Figure 2: Hardware monitor architecture.

traced event, generated by the "time stamp generator". Time stamped events are stored in the "event recorder", while the "Failure detection" component verifies that the sequence of events respects an expected order and some timing constraints. At last, the failure reporting component aims to extract the event trace from the monitor, to carry it on a supervision station for post-processing and analysis.

The next paragraphs give details about the design and functions of these 5 components.

3.1 Event capture and recording

Inside the recorder, a FIFO buffer, implemented by a circular array, stores the collected events, associated to their time stamps. When the buffer is full, the oldest events are forgotten.

If the monitor enters into reporting mode, the event recorder behavior depends on the chosen trace mode. In back trace mode, event recording is stopped as soon an erroneous event sequence has been detected, and only the events already present in the buffer are transmitted towards the supervision station. In forward trace mode, the monitor resets the array and the event recording will work as a temporary buffer between the event collector and the supervision station. If the buffer becomes full, the event recording stops, and the monitor only flushes the events available in the array. This behavior ensures that the event trace is not corrupted by intermediate missing events.

Event capture The basic interest of hybrid monitoring solutions is in reducing the interference on the observed system.

Hardware event sensors could use a technique like bus snooping [2], which limits the system disturbance. However, its implementation is technically difficult on processing systems that include complex memory hierarchy. Moreover, the point where sensors should listen could be unreachable from the monitor side [3]. Software sensors are easiest to implement but require source or OS code instrumentation. However, software sensors could impact the application temporal behavior.

Currently, we use software probes that write in monitor's memory-mapped registers. An event is coded on a 32 bits word, and is composed of an event type and a source identifier (i.e. a task identifier). The monitor manages the time-stamping of an event by a dedicated hardware component (see the next paragraph for details), and so the interference on the target system is expected to be limited (few memory word transfers by task job).

3.2 Failure detection

The failure detection module is in charge of verifying the system is working as expected. A micro-coded sequencer implements this function; the micro-code is included in the hardware configuration (see section 4).

Currently, the sequencer can only detect a periodic, – after the stabilization time –, and totally ordered suite of events. The first constraint is in accordance with the assumption on the target system, whereas the second one should be partially weakened in future designs. Fig. 3 shows the architecture of the detector. The values in the microcode memory defines the sequence of expected events. The events, represented by their source and their type identifiers, must arrive before or after a given time expressed in a micro-instruction.

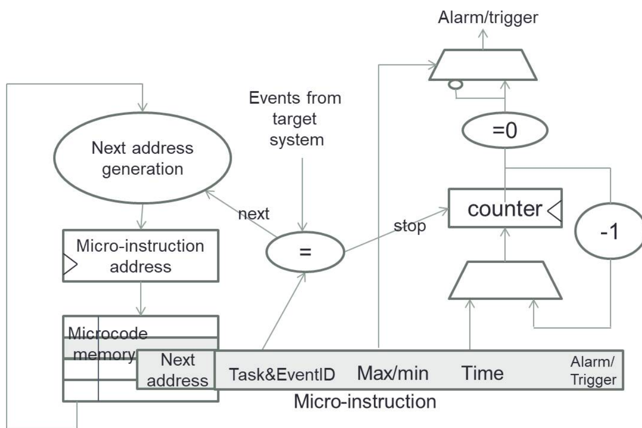

Figure 3: Failure detector (simplified hardware schematic).

We do not give anymore details about this failure detection module, because its architecture remains to be enrich to verify a more extended set of properties on the scheduling event trace.

3.3 Time stamping

Constrains of real-time system executions do not only concern the occurrence of events, but also the instant at which these events have been produced. So, time stamps go with the collected events. The following observations justify the way the time stamps is generated:

- non-intrusive: the access to the current time could imply calls to run-time (OS) services, then disrupting the execution of the observed application. The amount of information transfers to the health monitor must also be restricted to the bare minimum.
- independent; erroneous time management on the observed system could be difficult to analyze if event time stamps are issued from the same time reference.
- adapted: resolution, cycling and representation of time must be adapted to the need and the potential of the hardware monitor implementation. These requirements should be different than those of the target system.

The preceding remarks lead us to generate the time stamps within the hardware monitor, at the moment the events are received. We assume the duration of events collect trough Memory-Mapped register is constant, and therefore the time interval between 2 events is the same in the observed system and in the monitor.

The size, in number of bits, of the time stamp are constant, and must be small enough to limit (1) the storage needs to keep the trace in the monitor, and (2) the communication bandwidth to transfer the trace on the supervision station. A clock produces the time stamp, whose resolution depends on a periodic signal generated by frequency division from the basic system clock.

Fig. 4 shows the synopsis of the time stamp generation circuit: The frequency divisor creates periodic ticks at a frequency defined at the start up of the monitored system. This signal controls the increasing of a counter which gives time stamps when needed for a new event.

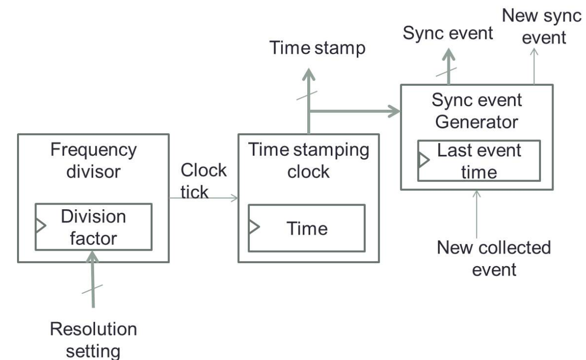

Figure 4: Time stamp management (simplified hardware schematic).

Considering a divisor factor register on 24 bits, a time register implemented on 16 bits, and a basic system clock at $100MHz$, the timer resolution goes from $10ns$ to $160ms$, and the timer counter overflow (cycling) occurs after about $0.6ms$ at the worst case.

The instant t_i at which an event i occurs is $t_i = k_i.t_{cycle} + ts_i.t_{res}$, with $k_i \in \mathbb{N}^+$, t_{res} and t_{cycle} being the timer resolution and the timer cycling period respectively. ts_i is the time stamp bound to the event i; k_i represents the number of counter overflows since the starting of the system.

The supervision station can get ts_i, t_{res} and t_{cycle}, but does not have access to k_i, since the event trace has been collected in the past, and only time stamps ts_i are associated to the events. So, to be able to rebuild the event instant at the level of the supervision station, the monitor must be sure to receive the next event within the counter overflow period subsequent to a given event. With this condition, $k_i = k_{i-1}$ if $ts_i \geq ts_{i-1}$, and $k_i = k_{i-1} + 1$ otherwise.

The instant t_i can be computed from the instant of the previous event by the following equation:

$$t_i = t_{i-1} + \begin{cases} (ts_i - ts_{i-1}).t_{res} & if \quad ts_i \geq ts_{i-1} \\ t_{cycle} + (ts_i - ts_{i-1}).t_{res} & if \quad ts_i < ts_{i-1} \end{cases}$$

To build the series of the event instants, the condition previously stated, i.e. the time between two collected events is less than the timer overflow period, must be ensured by the monitor. The component "Sync event generator" in Fig. 4 produces pseudo "sync" events to respect a minimum rate of event occurrence.

4 Implementation and first Experiments

Hardware platform The development board "ZedBoard" built by *AVNET* has been chosen to evaluate the ability of the hardware monitor to verify the health of a real software at run-time. This board is built around a *Xilinx* System-On-Chip Zynq7000. The Soc Zynq contains a Dual ARM Cortex A9 core processing system, and a programmable logic area of the family *Virtex7*. The connection with the supervision station (a Linux PC) is based on a USB2 serial link (115200 bauds UART emulation).

A VHDL model of the hardware monitor has been synthesized and implemented into the Zynq's FPGA. The current capacity of the event recorder is 1024 events. The failure detection can recognize sequences of 32 events. The micro-instructions that encode the sequence are stored in an internal memory (BRAM), whose the content is currently defined in the VHDL model. However, it is also possible to populate the BRAM by a direct updating of the FPGA configuration. With these parameters, the circuit occupies less than 5 % of the available FPGA resources whatever their type (LUT, BRAM ...).

First experiment: A two-task system This first experiment verifies the monitor ability to collect, time stamp and transfer events to a supervisor station in the forward trace mode. We consider a simple task model composed of 2 tasks, whose the periods are 20 and $5ms$, and the capacities 6 and $2ms$ respectively. The second task has a greater priority than the first one. The RTEMS OS[1] controls the target system. Its *Deterministic Priority Schedule*r [4] has been instrumented to signal scheduling events.

The Cheddar tool[2] is a scheduling analysis tool able to select and apply a set of analysis methods from a given task model and execution platform. Scheduling simulation results can be exported or imported as an XML file which contains the instants of significant scheduling events. Fig. 5 is a visualization of the events collected by the monitor after importation into *Cheddar*. The first period of each task appears as too short, due to time rounding in Cheddar and RTEMS tick resolution ($1ms$ in this experiment).

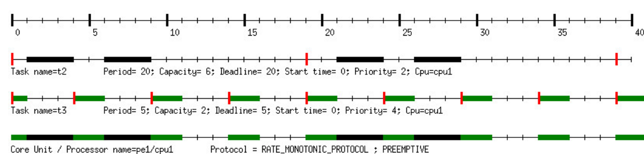

Figure 5: Collected events shown in the Cheddar's tool time line. Axis time unit represents 1 ms.

[1] http://www.rtems.org

[2] http://beru.univ-brest.fr/~singhoff/cheddar/

The transformation of the simulation trace into an expected and timed sequence of events must meet several challenges: How to relate logical simulation time and real execution ones, how to deal with task model abstraction (0-cost task switching for instance), how to order simultaneous simulation events, object matching between the simulation and the execution platform ...

5 Related works

An overview and a classification of monitors focused on timing constraints is established in [5]. Criteria like the adaptability, data collection methods, type of targeted systems or the monitor implementation organize the classification. Following this classification, our monitor is a "hybrid" monitor, based on a "tracing" data collection method and dedicated to the observation of "general" "real-time" and "embedded" system target.

In [6], Bandur et al. show how to implement a timed automaton on a micro-controller. The execution time of instructions in this micro-controller must be deterministic. The supported timed automaton assumes only one clock and that the time interval of concurrent outgoing transitions must be the same. The approach of this article could be a basis for an improvement of the "failure detection" module in our monitor.

Finally, Peters and Parnas argue in [7] that monitors should be based on the design requirements of the observed systems. They identify some necessary condition allowing a monitor to be feasible. The approach we propose follows this idea, although the parameters of a task model can be seen as a derivative of the system specification.

6 Conclusion

This article describes the overall architecture of a hardware monitor. First experiments have shown the ability of the monitor to collect the scheduling events of a sample task model controlled by the RTEMS operating system. The execution time of a software event sensor in the scheduler is less than $200ns$ and its intrusivity is limited.

Our goal is to derive automatically the monitor configuration from the real-time system task model and its scheduling. To achieve this objective, various assumptions or choices must be expressed and then specified in the design models. An architecture description language like AADL [8] can both specify the task model for the scheduling simulator and supply matching information to configure the monitor. We expect that expressing such information should contribute to increase the quality and conformity of the systems implementation.

Acknowledgments This work and Cheddar are supported by Brest Métropole, Ellidiss Technologies, CR de Bretagne, CG du Finistère and Campus France PESSOA programs number 27380SA and 37932TF.

References

[1] F. Singhoff, J. Legrand, L. Nana, and L. Marcé (2004), *Cheddar: a flexible real-time scheduling framework*, ACM SIGAda Ada Letters, vol. 24, pp. 1–8, ACM Press, New York, USA.

[2] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi (1990), *A noninterference monitoring and replay mechanism for real-time software testing and debugging*, IEEE Transactions on Software Engineering, vol. 16, no. 8, pp. 897–916.

[3] M. M. Gorlick (1991), *The flight recorder: an architectural aid for system monitoring*, in ACM SIGPLAN Notices, vol. 26, pp. 175–181, ACM.

[4] G. Bloom and J. Sherrill (2014), *Scheduling and thread management with RTEMS*, ACM SIGBED Review, vol. 11, no. 1, pp. 20–25.

[5] N. Asadi, M. Saadatmand, and M. Sjödin (2013), *Run-time monitoring of timing constraints: A survey of methods and tools*, in Proceedings of the the 8th International Conference on Software Engineering Advances (ICSEA), Venice, Italy, pp. 391–401.

[6] V. Bandur, W. Kahl, and A. Wassyng (2012), *Micro-controller assembly synthesis from timed automaton task specifications*, in International Workshop on Formal Methods for Industrial Critical Systems, pp. 63–77, Springer.

[7] D. K. Peters and D. L. Parnas (2002), *Requirements-based monitors for real-time systems*, IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 146–158.

[8] P. H. Feiler and D. P. Gluch (2012), *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*, Addison-Wesley, 2012.