Multilayer monitoring for real-time applications

Etienne Hamelin, Mihail Asavoae, Selma Azaiez, Alexandre Berne, Cyril Faure, Kods Trabelsi

Multilayer Monitoring for Real-Time Applications

Etienne Hamelin*, Mihail Asavoae*, Selma Azaiez*, Alexandre Berne*, Cyril Faure*, Kods Trabelsi*
*Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France (email: firstname.lastname@cea.fr)

***Abstract*—Validation of timing requirements of multicore, heterogenous and distributed systems is difficult problem because of a large number of situations that introduce temporal variability and/or interference. A possible solution is to augment the system with monitors and to rely on runtime monitoring techniques. In this paper we propose such a runtime monitoring which spans all the semantics layers of a model-based design, from high-level specification to executable code. We showcase our runtime monitoring on a safety-critical application for driving assistance.**

I. Introduction

Runtime monitoring is a lightweight verification technique for detecting property violations in embedded safety-critical systems. A typical runtime verification workflow consists of the definition, the synthesis and the execution of a collection of monitors which observe concrete executions of the system and check their conformance with respect to a set of stipulated properties. The observation aspect requires either a form of instrumentation (e.g. the code is made available) or external annotations (e.g. only black-box / COTS components are provided). The conformance checking aspect usually refers to property violations, or stated differently, to deviations from expected correct runtime behaviors. The design and implementation of a runtime monitoring environment needs to address these aspects as well as the characteristics of the embedded safety-critical system under consideration. In the following we propose a versatile runtime monitoring, which we denote as *multilayer runtime monitoring* and which combines high-level design details with low-level implementation particularities (of both application and execution platform).

We consider a model-based design (MBD) methodology for embedded software, adapted to address complex designs and to respond to complex and precise requirements. A prime example of such a methodology is based on the synchronous dataflow principle, with well-established workflows from Lustre/SCADE [8] or Simulink [9]. A MBD exposes several programming languages: the (just enumerated) high-level languages, intermediate languages like C and finally, low-level languages (e.g. assembly or binary code). In this case a multilayer runtime monitoring mimics the MBD languages, with monitors being placed at each level and an existing simulation/execution environment, likewise available. We also consider a distributed execution platform, which is suitable to map mixed-critical applications.

This research was partially supported by the ECSEL-JU under the program ECSEL-Innovation Actions-2018 (ECSEL-IA) for research project CPS4EU (ID-826276) in the area Cyber-Physical Systems.

In this paper we exemplify our multilayer runtime monitoring, driven by the non-functional requirements of a safety-critical application from the automotive domain. More specifically, in this paper we address timing properties. The MBD workflow starts with a high-level Polygraph design [11], a specialized dataflow-like language, which is further compiled and deployed on a distributed architecture. In this setting, the design of a multilayer runtime monitoring needs to address a certain number of challenges, presented in Section II. Moreover, details of the underlying MBD are in Section III, a high-level presentation of the multilayer runtime monitoring, in Section IV and the proposed case study, in Section V.

II. Challenges

One of the main challenges in designing and analyzing safety-critical applications running on complex computational systems (e.g. heterogeneous, distributed or even virtualized [15]) lies in the computation and communication timing variability of the application's components. This variability is due to various factors, among which we enumerate:

- task-specific behavior:
  - data-dependent (e.g. a task may chose a different behavior mode)
- inter-task interference due to shared software resources:
  - operating system-managed resources (e.g. scheduling, peripherals etc.),
  - exclusive services (e.g. drivers, communication protocols stacks, synchronization primitives, etc.)
- inter-task interference due to shared hardware resources:
  - shared CPU cores, shared caches etc.,
  - shared communication buses, shared peripherals, etc.

Static timing analyzers like aiT [1], or Otawa [5] compute safe bounds on the worst-case execution time of an application (i.e. behaviour-related timing variability). Another source of timing variability is due to interference between tasks; different techniques (e.g. static, statistical etc.) are developed to compute the necessary bounds on the timing behavior. To address interferences, static timing analyzers are extended with specialized shared cache analyses or cache-related preemption delay analyses.

Static and dynamic analyses are used to accurately capture the timing variability, whereas online monitoring is how the system identifies and addresses the deviations from specified/-validated timing values.

- static analysis is performed on the control-flow graph of the source or binary code,

- dynamic analysis is performed on traces extracted from concrete executions on the actual architecture, on an emulated target etc.,
- online monitoring is performed, at runtime, on the embedded target

The list of challenges, while non-exhaustive, is sufficient to expose the potential complexity arising when the timing variability is analyzed. One of the objectives of our multilayer runtime monitoring is to provide the necessary support, in the form of monitors, at various levels in this system representation (e.g. from high-level design to hardware).

III. Description of the Approach

An overview of our approach is presented in Figure 1, adhering to a MBD workflow for dataflow designs. A use case defines the application and a dataflow model, in Polygraph, defines the correct behavior of the system. From the Polygraph model, through a dataflow compilation, we generate elements of the source code, in C (i.e. a scheduler) which are then integrated with other elements of the source code (e.g. user-written code, libraries, RTOS APIs etc.), and finally compiled into a binary. When deploying and running such binaries on target boards, we obtain the execution traces which are then analysed by a processing engine. The goal is to check their conformance with respect to the correct timing, as defined in the high-level specification [17].

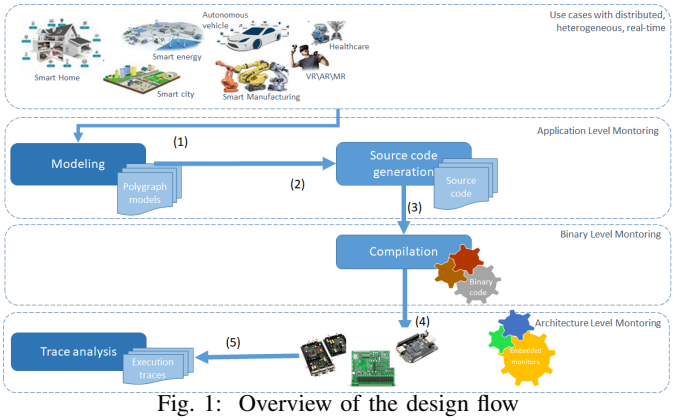


Fig. 1: Overview of the design flow

The first required step for analyzing the timing behaviour of our distributed application is to define accurately what is a "correct" behaviour for this application. To this end, we use the Polygraph model of computation and communication, presented in [11]. Polygraph is a real-time dataflow paradigm that extends the multi-rate synchronous dataflow approach.

The Polygraph model defines a complex set of causal relationships between actor/task firings, message tokens, and time instants. From a Polygraph model verified as both consistent and live, we generate elements of source code, which are linked with other user-provided code files and the instrumentation, and finally compiled into a set of binary firmware. When running these firmware programs on a heterogeneous platform made of single- or multicore targets, we monitor and analyze the execution traces generated from the instrumented programs.

In a MBD workflow, as in Figure 1, the semantics of the initial use case is successively transformed. First, the use case is formalized in the dataflow design (i.e. a high-level application semantics), then the C code is generated (i.e. an intermediate-level application semantics) and finally, it is compiled into the binary (i.e. a low-level application semantics). In order to preserve accurate traceability between the C code and the generated binary, the compilation is usually performed without optimizations. However, a coarser traceability could be preserved in the presence of compilation optimizations, so as to relate the high-level events to low-level functionalities (e.g. at function-level). In this work we consider compiler optimizations, however, regardless of the optimized/un-optimized compilation, it remains the fact that the use case semantics is obfuscated by this chain of transformations and, at the binary level, the use case characteristics (structural and functional) are difficult to identify and reason about. This motivates a runtime monitoring which should preserve some traceability features of the MBD compilation chain. Obviously, such a multilayered runtime monitoring implies the existence of execution environments at each level in the MBD workflow.

IV. Monitoring System

In the following, we briefly present some design decisions behind our multilayer runtime monitoring. We distinguish three levels of interest, starting with a dataflow Polygraph design and ending at the hardware hardware level and we organize this section as such.

A. Application level monitoring

The timing properties derived from the Polygraph design are used by the code generator to instantiate a set of software monitors. Here, we rely on the traceability of dataflow compilation chains. These monitors are inserted into the generated C code; each monitor is dedicated to observe a specific set of events, and related causal or temporal properties (as specified in the dataflow design). Then, these monitors output a trace of timestamped events for further conformance checking with respect to an expected behavior. As previously mentioned, the conformance checking could be of static or statistical nature.

B. Binary level monitoring

In case of safety-critical applications (i.e. requiring safe and tight timing bounds), timing analysis should be addressed in worst-case scenarios. There are two types of worst-case timing analysis: based on static analysis and on measurements. In the measurement-based approach, the binary is actually executed on the architecture or in a simulation environment and the results are interpreted with respect to a set of requirements. The instrumentation is necessary in this case to ensure that the collected executions are representative for the input application. In the context of runtime execution, and thus, in the presence of a monitoring system, the measurement-based

timing analysis seems to be the obvious choice. The code is instrumented at meaningful program points (actor job start, job end, message send/receive) so that runtime observations are possible whenever the code execution passes these points.

The binary instrumentation could and should be coupled with the non-intrusive hardware performance monitoring, which is detailed next.

C. Architecture level monitoring

The software-level runtime monitoring is complemented by a dedicated hardware infrastructure in the form of a Performance Monitoring Unit (PMU), available in most hardware architectures. The PMU allows recording of architectural and micro-architecture events for profiling purposes. Most modern CPU architectures contain a PMU embedded in the silicon, which provides performance counters (PMCs), a set of performance events to be counted Events such as preemptions, cache hits/misses could be tracked via the PMU to analyze the execution time of the system under test [6], [25]. Several commercial tools like ARM Streamline gator kernel module and daemon [2] are available to leverage PMUs in order to provide real-time feedback to engineers and help them to diagnose bugs or identify bottlenecks in software.

V. Case-Study and Experimentation

We showcase our approach by setting up a practical prototype platform, which embeds the monitoring layers described above into a use-case implemented as a Polygraph design.

In Figure 2, we present our use-case. This application is representative of the kind of real-time computing loads associated with advanced driver assistance systems (ADAS): the `Camera` actor generates image frames at a fixed rate e.g. 15 frames per second. These frames are analyzed by two mostly independent sub-chains. On one hand, lanes are detected on every frame using classic computer vision algorithms, and illustrates a time-critical chain. On the other hand `Objects detection` is performed using a deep neural network, and processes only a subsampled set of the input frames, in a soft real-time way. The `Display` sink actor serves as visual output, and additionally checks for deadline misses.

A. Case study implementation

The case study was implemented on a set of two Raspberry Pi multicore embedded computers, equipped with the Linux Operating system patched with PREEMPT_RT to support real-time applications, connected over Ethernet.

The RPi model 3B+ has a Broadcom BCM2835, 4-core ARMv7 processor, with 16KiB private L1 instruction, 16KiB L1 data cache, a 512KiB shared L2 cache.

Actors communicate through ZeroMQ messages using the publish/subscribe communication pattern. ZeroMQ [16], available as both C and Python-compatible, was chosen for its performance and versatility, supporting equally well inter-thread, inter-process process and Ethernet-based communications, independently from actual scheduling-related configuration. In comparison the popular ROS framework, available in C++ and Python, uses by default a single event queue for scheduling callbacks in a run-to-completion manner. This makes configuration and verification of real-time properties complex (see e.g. [34]).

Below is a simplified view of the implementation for an actor thread, where the business logic is inserted into a task and communication structure generated from the Polygraph model with inline tracing statements. The actor detailed, `Perspective Warp`, has a simple Polygraph specification: one input port, one output port, and no clock constraint, i.e. firings are triggered by incoming messages.

Listing 1: Example actor code

```
// actor thread
void *actor_perspective_warp(void *arg) {
  // Input channel: create subscriber socket
  void *z_in = zmq_socket(zctx, ZMQ_SUB);
  // connect to TCP endpoint
  zmq_connect(z_in, "tcp://...");
  // subscribe to topic
  zmq_setsockopt(z_in, ZMQ_SUBSCRIBE, "frame");

  // Output channel: Create publisher socket
  z_out = zmq_socket(zctx, ZMQ_PUB);
  // bind to inter-process com endpoint
  zmq_bind(z_out, "inproc://persp_warp");

  // Allocate image memory space
  img_t *im_in = img_new(...);
  img_t *im_out = img_new(...);

  // Main loop
  while (1) {
    trace_job_start(...);
    // receive (blocking) input message
    zmq_recv_image(z_in, im_in, ...);
    trace_token_received(...);

    // transform front view to bird-eye view
    img_perspective_warp(im_in, im_out, ...);

    // send output message
    trace_token_send(...);
    zmq_send_image(z_out, im_out, ...);
    trace_job_end(...);
  }
}
```

1) Test stubs: The ADAS camera is simulated by a `Image source` actor, implemented with Python and OpenCV. It periodically sends images (300x200 pixels, RGB format) extracted from a video file, shown in Figure 3.

The `Display` actor, implemented in Python, subscribes to the outputs of both lane and object detection chains, and additionally monitors the end-to-end latency of each processing sub-chain.

2) Lane detection: The image processing pipeline is inspired from the many Python implementations of Advanced Lane Detection published by participants to the popular "Self Driving Car" NanoDegree from Udacity, e.g. [27].

These actors represent a hard real-time processing sub-chain. They are implemented in bare C, as three separate

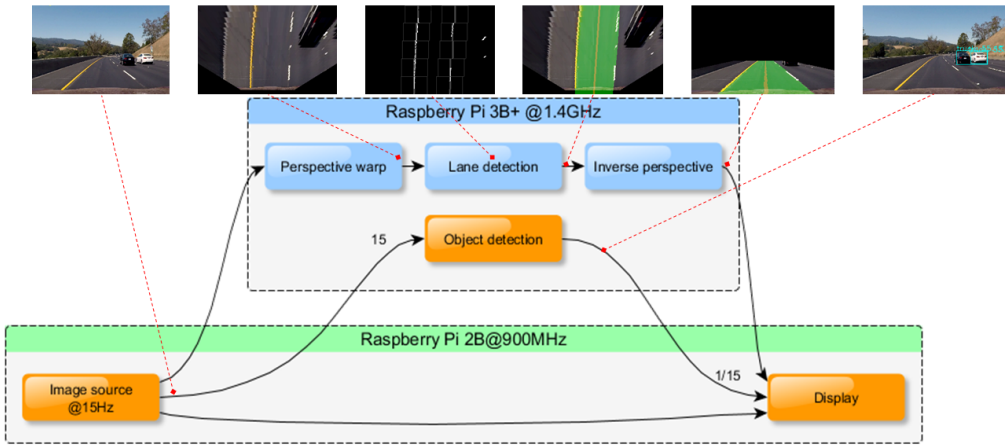


Fig. 2: Use-case dataflow

Fig. 3: Input image

threads. They are run on the isolated cores with high real-time priority.

The `Perspective Warp` actor transforms the front view image into a bird-eye-view image, following a manually-calibrated perspective transform, as shown in Figure 4a.

The `Lane Detection` actor then uses a color filter to extract lane markings, then a boxed histogram search to identify the most plausible markings (local maxima) for each lane, shown in Figure 4b.

A 2nd degree polynomial is fitted through the local maxima. The polynomial is used to extrapolate and draw curved lane besides the detected markings, shown in Figure 4c.

Finally, the `Inverse Perspective` actor projects the lane-marked bird-eye-view into a front view, shown in Figure 4d.

3) Object detection: The `Object detection` actor uses a deep neural network (DNN) to detect objects such as cars, trucks or motorcycles (see 5. The network used is the SSD Mobilenet v3, pre-trained on 300x300 color images using the COCO Small dataset [31]. This network is not trained specifically for ADAS applications, but is however representative of the type of computational load typical of computer vision DNNs.

This actor is implemented in Python, and uses the Tensorflow Lite runtime. Due to its intensive CPU and memory usage, this actor only processes 1 frame per second. This actor is mapped onto the non-isolated cores.

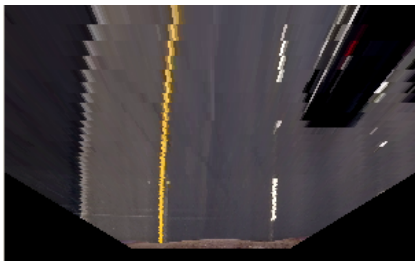
4) Variability sources: In II, we recalled the most prominent sources of temporal variability in real-time applications. The following measures are configured so as to reduce variability to a minimum:

- data-dependent behavior is reduced to a minimum: the memory allocation is authorized only during setup, and the execution flow depends on image size but not image content. Only the Gauss-Jordan matrix inversion used in the polynomial fit step has non-deterministic execution flow, however this represent a small fraction of the execution time (less than $100\mu s$);
- time-critical tasks are scheduled as high-priority real-time threads
- time-critical tasks are run on isolated CPU cores: the `isolcpu` and `taskset` commands prevent the OS from scheduling background services on two CPU cores dedicated to real-time tasks, and consequently ensure interference-free access to private L1 caches;
- inter-task synchronization is limited to blocking on message reception, thus enforcing a correct Polygraph dataflow semantics.

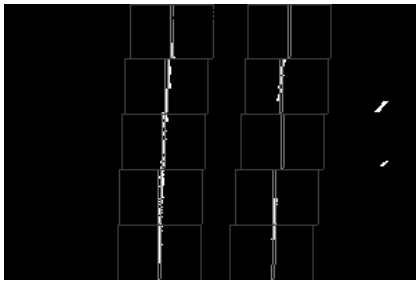
The execution times of the non-critical Object detection sub-chain on the other hand show much more variability, due to e.g.:

- data-dependent behavior: due to its black box nature, we could not inspect to what extent the TFLite runtime behavior does depend on actual image content. To the least, we observe a variable execution time, and assume that some operations depend on e.g. the number of objects detected.

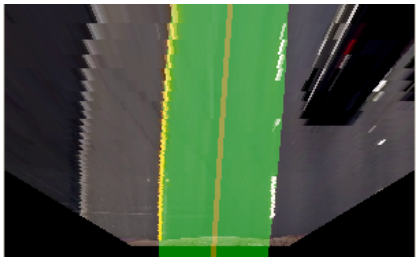
This task configuration is expected to reduce the risk of interference between non-critical and critical tasks. However, the shared L2 cache remains a possible source of interference,



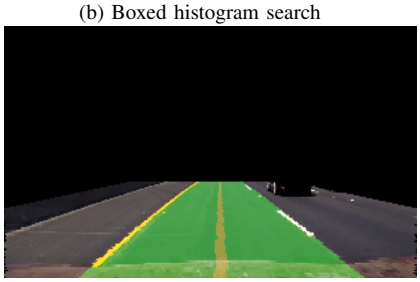
(a) Perspective warp



(b) Boxed histogram search



(c) Polynomial-fitter lane markings



(d) Inverse-perspective warped lane markings

Fig. 4: Lane detection chain: intermediate stages

Fig. 5: Detected objects

since both critical and non-critical task chains process memory chunks larger than the last-level L2 cache.

5) Monitoring infrastructure: The monitoring infrastructure is integrated within each Python and C-based actor thread, using the low-overhead LTTNG-UST infrastructure (Linux Tracing Toolkig - New Generation, User-Space Tracing). In our configuration, LTTNG-UST tracing was tested to cost approximately $5\mu s$ per trace point, well below the time granularity of the monitored events. We monitor all Polygraph dataflow relevant events, that is: actor job start, job end, message sent, message received. Using these events, we can compute the execution times of each actor job.

In addition, the `Image Source` actor generates a timestamp, which is transported together with the image payload during all processing steps. This enables other actors computing the input and output latency of each actor (time of arrival of input data, and time of departure of output data, relative to original input timestamp).

The real-time, NTP synchronized, clock `CLOCK_REALTIME` is used to compare timestamps from distributed platforms with millisecond-level precision, whereas the monotonic clock `CLOCK_MONOTONIC` is used to compare timestamps inside a given platform, up to microsecond precision for e.g. actor execution times.

The event trace generated by this monitoring infrastructure, showed in Figure 6 allows to analyze the evolution in time of execution times of critical actor jobs (in μs; top section). In particular, the execution time of critical actors is relatively stable (standard deviation is below 2% of the average). The input/output latency is computed at each actor job (in ms; middle section).

In the PMU trace (lower section), we see hardware-level metrics for the `Perspective warp` actor. In particular, from PMU monitored metrics we compute the instruction per cycle ratio (IPC), and miss rate at private (L1) as well as shared (last level) cache. In our application, a significant variability of the miss rate at shared cache LL ($stddev \approx 12\% \cdot avg$) is probably caused by interference from the non-critical, memory-intensive, `Object detection` actor. This variable LL miss rate would cause a variable access time for data outside L1 cache, and variable instruction per cycle (IPC). However since most data reads only hit the L1 cache (miss rate $< 1\%$), the effect of LL cache interference is not much visible on actor execution time or the IPC metric in this run.

B. Static WCET Analysis

In order to better establish correlations between the monitoring infrastructure and the architecture timing, we have considered a cycle-accurate, WCET analysis for the `Perspective Warp` actor. More precisely, we consider a reference implementation of this actor which is then evaluated with Otawa static timing analyzer [5] for ARM processors. Whereas specialized analyses are necessary to address shared caches or the possibility to accommodate the preemption, Otawa proposes single-core, non-preemptive analysis, hence we only directly analyze L1 cache effects. Let us elaborate next on these two aspects of the experimentation.

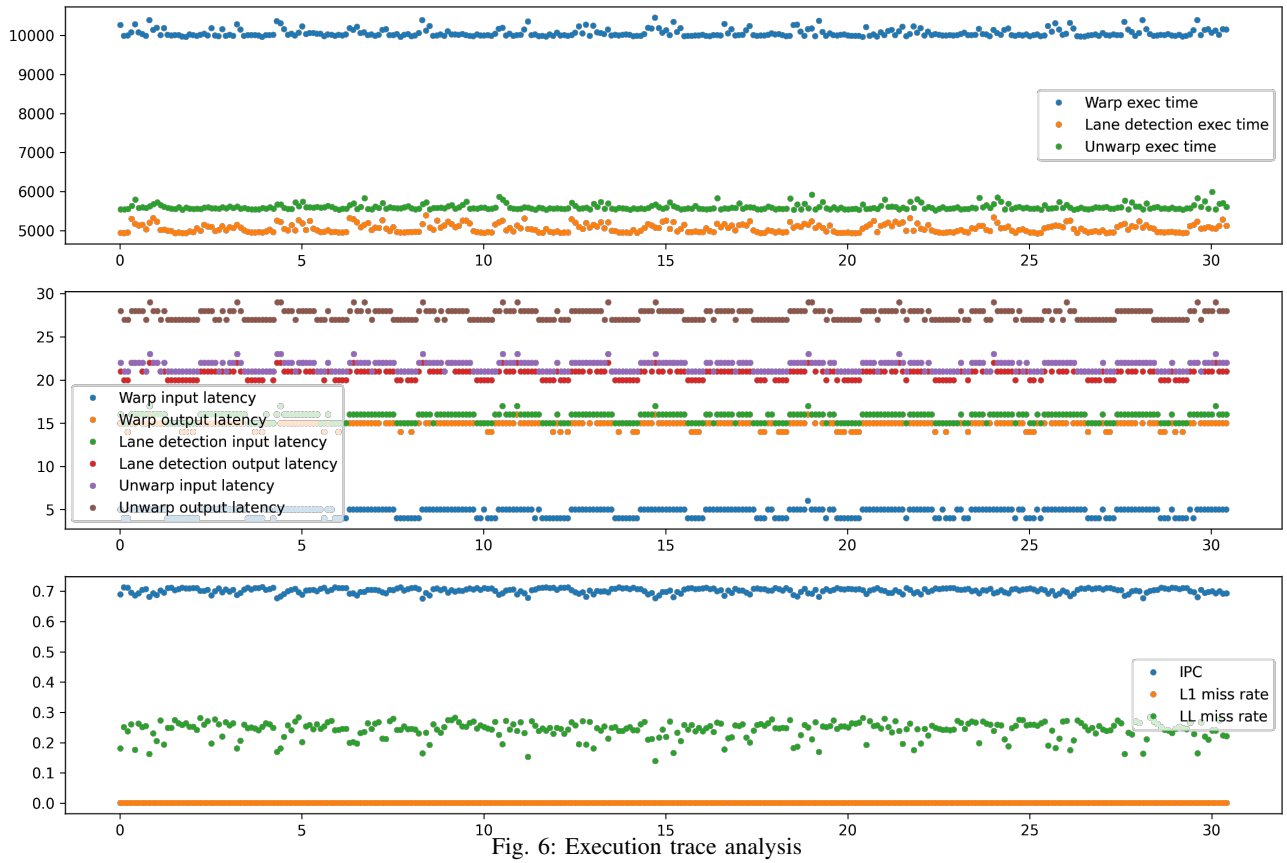


Fig. 6: Execution trace analysis

The reference implementation of the `Perspective Warp` actor is characterized by the following elements. First, the input image, as shown in Figure 3 is of fixed size (i.e. 300x200 pixels) and the output image, as shown in Figure 4a is also of fixed side (i.e. 200x133 pixels). We note that both image sizes are modifiable, but we fix them as a referenced input-output, as we have previously stated. Second, the code is organized in three stages: memory allocation of all the manipulated images, followed by a perspective transformation algorithm (through a resolution of a system of linear equations) and finally, the generation of the resulted imaged. We note that our contribution (wrt. the monitoring infrastructure) is to observe the timing contribution of the memory accesses. Third, the code is stripped of auxiliary functionalities, namely calls to the ZMQ library and other tracing operations. Finally, we also addressed a highly-optimized version (`-O3`) of this code (as used in all the other tests). From the architecture point of view, we analyze with Otawa single-core timing, using cache analyses for L1 caches (and without considering the L2 shared cache). As such, the resulting code presents its core functionality which is then evaluated with the static timing analyzer in order to obtain a timing bound in the absence of other interferences (i.e. from `ObjectDetection` and Linux background services).

The Otawa timing analyzer is also highly configurable, featuring, on the architecture side, an infrastructure for ARM, RISC-V or PPC architectures (to name a few) and on the analysis side, a wide range of abstractions to compute flow-facts from both the input program and the underlying architecture (e.g. standard cache may- and must- analyses). Also, Otawa proposes an advanced scripting to facilitate the extraction of loop addresses, in order to introduce loop bounds. We note that Otawa is accompanied by oRange [10] tool to compute these loop bounds, however, due to the fact that our code contains only for-loops, it is not necessary to consider oRange for these bounds. In our experimentation with the `Perspective Warp` actor, we consider the ARM architecture of Otawa with a simple pipeline and instruction and data caches, for which the aforementioned analyses are selected. In this setting, Otawa returns slighly more than 12 million cycles for each image processed, for a code which is dominated by memory accesses (55% instructions). This estimate is roughly three times the cycle count measured by runtime monitoring.

The Otawa analyzer only considers a 1 level cache, whereas our target features 2 cache levels; for this reason we cannot expect Otawa to provide an accurate figure. However since the L1 miss rate is low (cf. Fig. 6, consistent with Otawa analysis), the effect of 2nd level cache cannot explain the large difference. This discrepancy needs to be further investigated.

While these results are used as a baseline WCET behavior for `Perspective Warp`, similar investigations could be performed for the other time-critical actors of our case study.

VI. Related Work

Model-based design (MBD) approaches for real-time systems consider as, for the high-level application, a design developed using synchronous languages like Lustre [13]. For this particular language (as for others in the same family), a notion of observer could be defined [14] to address design properties. More precisely, an observer, according to [14] is another program which observers the behavior of the original application (in Lustre) and determines eventual deviations from stipulated correct behaviors. In other words, an observer is a monitor for the high-level application and a high-level Polygraph design, as in our work, could also support such an approach.

The Polygraph formal model is also used by Alkalee, a spin-off of CEA LIST, as the base of their modeling tool Euphilia [32]. Alkalee moreover provides the Receef runtime environment [33], which can supervise actors' communication events according to the input system model. The Receef monitors can additionally trigger various containment strategies for communication faults, e.g. when an expected data sample is not received on time. To the best of our knowledge, the Receef monitoring infrastructure does not cover, nor interface to, low-level architectural observation points such as hardware performance counters.

In the context of the low-level application and the underlying execution platform (wrt. the same MBD workflow), a typical monitoring systems relies on hardware performance counters (HPCs) to observe the behavior of the system under analysis. As such, HPCs are used to address security properties, e.g. in [7], safety properties, e.g. in [20] and non-functional requirements, e.g. energy [18] and timing [22]. The key aspects in using HPCs to cover a wide range of properties are that modern processors support them and also that their overhead wrt. the runtime analysis is minimal. We also consider HPCs in our current work for the same reasons, however, we also aim to correlate the results of their measurements with different observation methods (i.e. at different semantics levels in the system design). Such correlations define our framework of multilayer monitoring for mixed-criticality systems.

The worst-case timing analysis is necessary to ensure that critical tasks in mixed-criticality systems are able to meet their timing deadlines. There are two types of timing analysis - using static-based [30] and measurement-based [29] methods. Our multilayer monitoring framework draws inspiration from both, as follows. The static timing analysis, when applied to MBD workflows (i.e. a survey of methods and techniques is presented in [4]) considers the application to be represented and analyzed at each level [19] and aims to establish traceability properties between these levels. In the same way, our approach aims to exploit the traceability towards increasing the confidence between the observed timing behavior at the various levels. The measurement-based timing analysis uses the HPCs to estimate timing bounds of critical tasks, having problems of compositionality [21]. Another way of performing measurement-based timing analysis is by code instrumentation followed by the application of statistical and/or probabilistic methods on the collected results [3]. In the same way, our approach considers the HPCs, but with a broader goal, that of establishing (composable) connections with different levels in a MBD workflow.

VII. Conclusion

We explained in this document how we used a MBD methodology to define a monitoring system for a distributed execution platform in order to validate their temporal behaviour during their execution. Using a high-level dataflow language, i.e. Polygraph, the correct temporal behavior is specified. From these specifications, source code is generated or written to embed software monitors. These monitors are compiled into the application, and inserted into the application execution flow, and track dataflow-relevant events. At the binary level, two types of worst-case timing analysis are considered. For the first one, static analyses are performed on the control flow graph as a high level abstraction of the binary code. The second one a measurement-based approach where the binary is actually executed on the architecture and the results are interpreted with respect to these measurements. Finally, at the architecture level, execution is profiled using tools such as Performance Monitoring Unit (PMU). We have shown that these various monitoring layers provide complementary data, especially useful to modelling the actual execution time in presence of inter-task interference.

References

[1] https://www.absint.com/ait/index.htm
[2] https://github.com/ARM-software/gator
[3] https://www.rapitasystems.com
[4] Mihail Asavoae and Claire Maiza and Pascal Raymond, "Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective", In 13th International Workshop on Worst-Case Execution Time Analysis (WCET), pp. 32–41, 2013.
[5] Ballabriga C., Cassé H., Rochange C., Sainrat P. (2010) OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: Min S.L., Pettit R., Puschner P., Ungerer T. (eds) Software Technologies for Embedded and Ubiquitous Systems. SEUS 2010. Lecture Notes in Computer Science, vol 6399. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-16256-5_6
[6] W. L. Bircher and L. K. John, "Complete System Power Estimation Using Processor Performance Events," in IEEE Transactions on Computers, vol. 61, no. 4, pp. 563-577, April 2012, doi: 10.1109/TC.2011.47.
[7] Malcolm Bourdon and Eric Alata and Mohamed Kaaniche and Vincent Migliore and Vincent Nicomette and Youssef Laarouchi, "Anomaly detection using hardware performance counters on a large scale deployment", In ERTS 2020.
[8] J.-L. Colaço, B. Pagano and M. Pouzet, "SCADE6: A Formal Language for Embedded Critical Software Development", in 11th International Symposium on Theoretical Aspects of Software Engineering (TASE), pp 1-11, 2017
[9] J. Dabney, T. Harman, "Mastering Simulink", Pearson Ed, 2004
[10] Marianne De Michiel and Armelle Bonenfant and Hugues Cassé and Pascal Sainrat, "Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation", The Fourteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 161–166, 2008
[11] Dubrulle P., Gaston C., Kosmatov N., Lapitre A., Louise S. (2019) A Data Flow Model with Frequency Arithmetic. In: Hähnle R., van der Aalst W. (eds) Fundamental Approaches to Software Engineering. FASE 2019. Lecture Notes in Computer Science, vol 11424. Springer, Cham. https://doi.org/10.1007/978-3-030-16722-6_22

[12] Francalanza, A., Pérez, J.A. and Sánchez, C., 2018. Runtime verification for decentralised and distributed systems. Lectures on Runtime Verification, pp.176-210.
[13] Nicolas Halbwachs, "A synchronous language at work: the story of Lustre", International Conference on Formal Methods and Models for Co-Design, MEMOCODE, pp.3–11, 2005.
[14] Nicolas Halbwachs and Fabienne Lagnier and Pascal Raymond, "Synchronous Observers and the Verification of Reactive Systems", in Algebraic Methodology and Software Technology (AMAST), pp. 83–96, 1993
[15] Hamelin, Etienne and Ait Hmid, M and Naji, Amine and Mouafo-Tchinda, Yves, "Selection and evaluation of an embedded hypervisor: Application to an automotive platform", European Congress of Embedded Real Time Software and Systems, 2020
[16] P. Hintjens, "ZeroMQ: Messaging for Many Applications", O'Reilly Media, 2013
[17] R. Kirner, R. Lang, G. Freiberger and P. Puschner, "Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models", in 14th Euromicro Conference on Real-Time Systems (ECRTS), pp 31-40, 2002
[18] Ghislain Landry and Tsafack Chetsa and Laurent Lefevre and Jean-Marc Pierson and Patricia Stolf and Georges Da Costa, "Exploiting performance counters to predict and improve energy performance of HPC systems", in Future Gener. Comput. Syst, pp. 287–298, 2014.
[19] Claire Maiza and Pascal Raymond and Catherine Parent-Vigouroux and Armelle Bonenfant and Fabienne Carrier and Hugues Cassé and Philippe Cuenot and Denis Claraz and Nicolas Halbwachs and Erwan Jahier and Hanbing Li and Marianne De Michiel and Vincent Mussot and Isabelle Puaut and Christine Rochange and Erven Rohou and Jordy Ruiz and Pascal Sotin and Wei-Tsun Sun, "The W-SEPT Project: Towards Semantic-Aware WCET Estimation", In 17th International Workshop on Worst-Case Execution Time Analysis (WCET), pp. 9:1–9:13, 2017.
[20] Corey Malone and Mohamed Zahran and Ramesh Karri, "Are hardware performance counters a cost effective way for integrity checking of programs", In Workshop on Scalable trusted computing, STC@CCS, pp. 71–76, 2011
[21] Cristian Maxim and Adriana Gogonel and Irina Mariuca Asavoae and Mihail Asavoae and Liliana Cucu-Grosjean, "Reproducibility and representativity: mandatory properties for the compositionality of measurement-based WCET estimation approaches", In SIGBED Rev., pp. 24–31, 2017.
[22] Jan Nowotsch and Michael Paulitsch and Arne Henrichsen and Werner Pongratz and Andreas Schacht, "Monitoring and WCET analysis in COTS multi-core-SoC-based mixed-criticality systems", In Design, Automation & Test in Europe Conference & Exhibition (DATE) pp. 1–5, 2014
[23] Navabpour S., Bonakdarpour B., Fischmeister S. (2015) Time-Triggered Runtime Verification of Component-Based Multi-core Systems. In: Bartocci E., Majumdar R. (eds) Runtime Verification. Lecture Notes in Computer Science, vol 9333. Springer, Cham.
[24] Reinbacher T., Függer M., Brauer J. (2013) Real-Time Runtime Verification on Chip. In: Qadeer S., Tasiran S. (eds) Runtime Verification. RV 2012. Lecture Notes in Computer Science, vol 7687. Springer, Berlin, Heidelberg.
[25] R. Rodrigues, A. Annamalai, I. Koren and S. Kundu, "A Study on the Use of Performance Counters to Estimate Power in Microprocessors," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 60, no. 12, pp. 882-886, Dec. 2013, doi: 10.1109/TCSII.2013.2285966.
[26] Norman Scaife and Christos Sofronis and Paul Caspi and Stavros Tripakis and Florence Maraninchi, "Defining and translating a "safe" subset of Simulink/Stateflow into Lustre", In EMSOFT, pp.259–268, 2004
[27] Siddharth Sharma, Advanced Lane Lines Detection, https://github.com/sidroopdaska/SelfDrivingCar/
[28] Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A. and Zadok, E., 2011, September. Runtime verification with state estimation. In International conference on runtime verification (pp. 193-207). Springer, Berlin, Heidelberg.
[29] Ingomar Wenzel and Raimund Kirner and Bernhard Rieder and Peter P. Puschner, "Measurement-Based Timing Analysis", In Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, (ISoLA), pp. 430–444, 2008.
[30] Reinhard Wilhelm and Sebastian Altmeyer and Claire Burguière and Daniel Grund and Jörg Herter and Jan Reineke and Björn Wachter and Stephan Wilhelm, "Static Timing Analysis for Hard Real-Time Systems", In Verification, Model Checking, and Abstract Interpretation, 11th International Conference, (VMCAI), pp. 3–22, 2010.
[31] Hongkun Yu and Chen Chen and Xianzhi Du and Yeqing Li and Abdullah Rashwan and Le Hou and Pengchong Jin and Fan Yang and Frederick Liu and Jaeyoun Kim and Jing Li, TensorFlow Model Garden, https://github.com/tensorflow/models, 2020
[32] Euphilia, the systems modeling tool, ALKALEE, https://www.alkalee.fr/products/euphilia/
[33] Receef, the embedded features orchestration software, ALKALEE, https://www.alkalee.fr/products/receef/
[34] Daniel Casini and Tobias Blaß and Ingo Lütkebohle and Björn B. Brandenburg, Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling, ECRTS, 2019