Predictable Vision for Autonomous Systems

Michael Balszun*, Martin Geier* and Samarjit Chakraborty†
*Technical University of Munich, Germany †UNC Chapel Hill, USA

***Abstract*—In this perspective cum case-study paper, we argue the need for designing timing-predictable vision processing algorithms for autonomous systems. Many core functions in systems like autonomous vehicles involve computer vision within a control loop. Designing such closed-loop controllers and guaranteeing their performance requires the vision processing to be predictable. But this is challenging given the multitude of choices when implementing vision processing algorithms, and the heterogeneity of the architectures (involving GPUs and FPGAs) on which such algorithms are implemented. Towards this, we report a tracing and measurement infrastructure we have been building and illustrate its potential utility using a case study.**

I. Introduction and Related Work

Vision is an integral part of many autonomous systems, ranging from autonomous cars, to robots and industrial automation systems. Autonomous cars are today equipped with multiple cameras, sometimes lidars, and otherwise radar sensors. Data from these devices are processed to estimate the distance of cars and other obstacles in the vicinity and also their speed and direction of motion. These estimates are then fed into control applications, that determine control inputs which in turn determine the speed, braking and driving direction of the autonomous car.

Given the safety-critical nature of the various controllers in autonomous cars, it is imperative that their real-time behavior is crucial for their correct functioning. In particular, both their design, and also their testing, certification, and debugging relies on their *end-to-end* timing behavior from sensing to actuation.

Timing analysis of embedded systems: There is a large volume of literature on various aspects of timing analysis of distributed embedded systems. The most basic, yet still a very difficult problem, is that of estimating the worst-case execution time (WCET) of a piece of code running on a processor. This problem involves a static analysis of the code along with modeling the microarchitecture of the processor on which the code is running, and is referred to as the WCET analysis problem. There is a long history of work on software timing analysis, including WCET analysis [1], [2], [3], [4], [5], [6], [7] and commercial WCET analyzers such as those from AbsInt GmbH [8] and Rapita Systems [9] are now routinely used in the automotive industry.

Building on the WCET analysis of individual code blocks, *system-level timing analysis* aims to determine the overall or end-to-end timing behavior of a full system or a large sub-system [10], [11]. There have been multiple studies focusing on system-level timing analysis of automotive hardware/software architectures [12] and system-level timing analysis tools such as those from Inchron [13] and SymtaVision (now Luxoft) [14] are now routinely used by automotive OEMs and suppliers for both timing-aware design and also verification of timing constraints.

The timing analysis problem for automotive architectures, in-vehicle communication buses and protocols [15] and automotive software [16], [17], [18], [19] has received considerable attention both because of its complexity and also its practical/industrial relevance. While the need for timing guarantees stemming from the safety-critical nature of the automotive domain is obvious, it is complicated by the highly cost-sensitive nature of the domain which necessitates tight resource-dimensioning. The cost-sensitivity also rejects any pessimism in the timing analysis results. This makes many of the static timing analysis techniques difficult to apply in the automotive domain, although they might be usable in other domains like avionics which are less cost sensitive.

Beyond static analysis, relying on measurements: Hence, in spite of this large volume of work on timing analysis techniques and tools, including those specifically targeting automotive architectures and software, there are still a number of pending technical challenges that need to be overcome. The first is the scope of commonly used task models that can be analyzed using techniques from Real-time Systems (RTSs). In order to keep the analysis tractable, the models need sacrifice some of their expressibility thereby limiting their use in real-life applications. To address this, there have been efforts to use more generic verification techniques such as model checking [20], but they suffer from limited scalability. A more pragmatic approach, which while widely followed in practice, has more recently also attracted considerable academic interest, and is referred to as *measurement-based timing analysis* [2], [21], [22], [23], [24]. It collects a large number of execution times of smaller code blocks, program paths, or program instructions. This is done using measurements that compared with static analysis circumvents the challenges associated with accounting for the effects of the microarchitectural features of processors on execution time, albeit while losing some accuracy. These results are then combined using static analysis and probabilistic methods to estimate higher-level timing properties, which could be worst-case execution times of code or system-level timing properties.

Timing analysis of computer vision algorithms: Timing analysis of automotive control applications has been studied over the past couple of years [25], [26], [27], [28], including how the control signal delays influenced by scheduling algorithms might impact control performance. However, the assumption in all of these studies has been that the timing behavior *can* be estimated or measured. How to do this estimation has not been the focus of these studies.

While significant advancements have lately been made in designing autonomous systems, a major roadblock in their widespread deployment lies in *verifying* their correctness, or in other words *certifying* that they would work correctly under *all* possible scenarios. In fact **verifying timing correctness**

is considered to be one of the major challenges to be overcome in autonomous driving [29]. In addition to the timing analysis challenges already outlined above, this problem is exacerbated by the heavy reliance on computer vision processing in autonomous systems, and in particular in autonomous vehicles. Whereas the need for designing *timing predictable* embedded systems [30] is well understood, the concept of timing predictability is virtually unknown in the domain of computer vision. The goal has always been to process as “fast as possible”, instead of guaranteeing any timing bounds, of the form that are required when such vision processing is on the path of a feedback control loop. In addition to the algorithmic complexities of computer vision algorithms and their reliance on libraries such as OpenCV, which have not been designed with timing predictability in mind, a further layer of complexity in terms of timing analysis is added by hardware accelerators like GPUs and FPGAs, which vision processing algorithms often rely on.

Given the high workloads associated with vision processing and often also the high frame rates required by *visual servoing* (vision-based control) applications, purely software solutions are mostly not feasible. Hence, architectures of autonomous cars already today and also in the foreseeable future will be heterogeneous and will heavily rely on GPUs [31], [32] and FPGAs [33], [34]. Currently, there are no known static analysis techniques for estimating the timing behavior of either of these two classes of accelerators, and this situation is likely to remain so in the future. Hence, measurement-based techniques constitute the only potential solutions for both timing analysis and also timing debugging of vision-based control applications for autonomous systems.

Infrastructure for timing measurements for FPGA-based vision processing: Very recently, there have been efforts to provide timing guarantees for computer vision algorithms on GPUs relying on OpenVX [35]. Along similar lines, running convolution neural network-based computer vision algorithms on GPUs with the aim of timing predictability has also been investigated [36].
In this paper, we report our efforts to build a measurement infrastructure that can provide end-to-end timing measurements for FPGA-accelerated real-time systems. In particular, we describe a case study of a visual servoing system that implements a multi-camera-based real-time control application that can serve as a surrogate for a vision-based control application in an autonomous car. Our measurement infrastructure can provide detailed traces of events in the system, that can help in timing, and timing-assisted functional debugging.

We argue that timing measurements using such an infrastructure can be used to certify the functionality of control algorithms that rely on cameras and vision-based processing. The complexity and the black box nature of vision processing algorithms and the hardware they rely on, make it very difficult to extract timing guarantees otherwise. Our infrastructure provides detailed insights into the different processing stages in such a system, which is helpful for timing debugging and the selection of appropriate parameters of controllers lying at the core of any autonomous functionality.

Summary and outline: The main contribution of this paper lies in highlighting the challenges associated in estimating the timing behavior of vision processing algorithms that lie at the heart of many autonomous systems, and especially autonomous vehicles. In particular, we describe the various design choices that vision processing involves and what workload and timing impacts they might have (Sec. II). Next, we outline a measurement infrastructure that we have been developing for timing analysis and debugging of vision processing algorithms running on FPGA-based heterogeneous architectures (Sec. III). Finally, in Sec. IV we illustrate the utility of this infrastructure towards designing timing-predictable vision processing systems, using a case study that is representative of vision-based closed-loop control applications found in many autonomous systems.

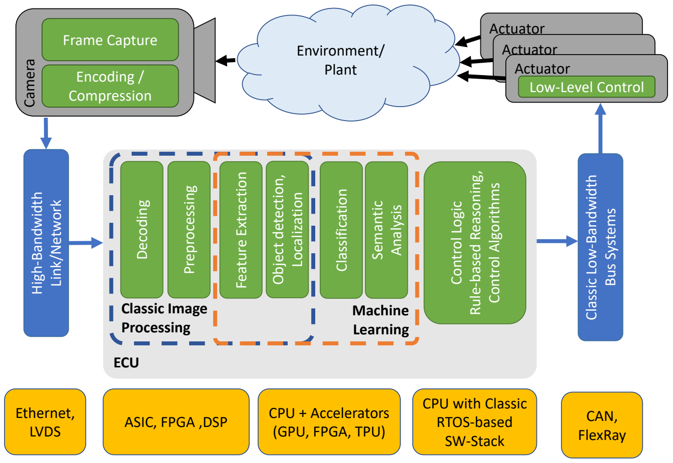

Fig. 1. Physical Plant with Sensors/Actuators (top) and Processing (bottom)

II. Design and Analysis Challenges for Vision-based Control Systems

The processing and bandwidth demands of vision-based applications are generally much higher than that of classic closed-loop control systems. For example, the necessary calculations for a simple control law or the filtering of low-dimensional Inertial Measurement Unit (IMU) data are negligible in comparison to the computations necessary for running a simple convolutional filter over a single full HD image with 2 million pixels. In addition, the processing pipeline of a visual servoing application (VSA) is usually much deeper and more complex and involves a multitude of different hardware and software systems (Fig. 1 gives an overview of a typical pipeline) and as a final challenge for timing analysis techniques, the run times of many involved algorithms are inherently dependent on the data. At the same time, most of the tasks based on such vision data – from lane control and path planning (including overtake maneuvers) to emergency breaking in a vehicle – usually impose hard limits on the acceptable worst-case response time (WCRT) and the design needs to ensure that they are satisfied for all relevant operating conditions. In the remainder of this section, we provide a brief overview over the different stages typically involved in a VSA. We also discuss the impact that various possible vision processing design choices have on the temporal characteristics of the data stream and the predictability of the end-to-end latency in the system, as it impacts the performance of the closed-loop control system.

Image capture: This is the process of transforming light falling onto the camera sensor into a sequence of individual frames. In most cases, camera systems in the automotive domain can be assumed to run at a fixed frame rate, as opposed to being triggered by events as is common in some manufacturing scenarios (e.g. quality control, or sorting). Still, depending on the camera technology and configuration, the output can have different temporal properties in terms of how the image data is streamed and the temporal coherency of the image data itself: If the camera works in rolling shutter mode (common e.g. in classic CCTV cameras), the image data of different pixels in a single frame (usually grouped by lines) corresponds to a different point in time during the frame period, whereas in global shutter mode, all pixels represent the same moment in time. From a classic timing analysis perspective the temporal coherency of the data in a frame itself may not be important as long as the stream properties at the camera interface are known, however, a proper end-to-end timing analysis and control logic may need to take this effect into account. A rolling shutter design usually also implies a continuous stream of pixel data whereas global shutter mode creates bursts of data packages.

Encoding and Compression: For cameras that are not directly attached to an Electronic Control Unit (ECU) but are rather connected via one of the vehicle's bus systems (see next paragraph), there is no question that some degree of video compression is needed. For example, a single full-HD video stream at 60 frames per second with a naive 8 bit RGB encoding would already by far exceed the bandwidth of a Gigabit Ethernet link (60 fps times 1920 x 1080 pixels/frame times 24 bits/pixel $\approx$ 3Gbit/s). Video and image encoding has been a field of intensive research for decades and there exists a huge variety of different encodings for video data that often build on top of each other. Starting from classic per-pixel encodings in different color spaces like RGB or HSV to spatial sub-sampling schemes and compression algorithms for single images (e.g., JPEG) and finally to modern video compression schemes like the ubiquitous H.264/AVC and the more modern H.265/HEVC standards. Unfortunately, virtually any non-trivial video compression algorithm will introduce non-determinism in one dimension or another: Compression ratio, run time and image quality are the three most important metrics and not all three can be kept constant on a per-frame basis. E.g. lossless compression schemes like Huffman or Arithmetic Encoding – which are also an important post-processing step in most lossy compression schemes – provide a constant quality but varying compression ratios and differences in run time. For maximal compression efficiency, most lossy video encoding schemes have usually even more unpredictable run time behavior, but at the same time provide many configuration options that allow the development of encoders that produce constant or quasi constant data rates (constant bit rate (CBR) and average bit rate (ABR) rate control modes) at the cost of having a high variability in the image quality – in particular if the encoder has to work a single pass mode which is desirable in low latency systems.

Another important aspect to consider is the use of intra-frame prediction. Video encoding schemes usually use information from the previous frame to more efficiently encode the next frame, where frames that can be decoded independently are called I-frames (Intra-coded frame) and those that depend on previous image data P-frames (Predicted picture). On the one hand this can greatly reduce average data rate but on the other it does not help with the worst-case data rate and it also poses questions related to reliability – if an I-frame gets corrupted or lost, all consecutive P-frames can no longer be decoded. For those reasons, most implementations for hard RTSs only use intra-frame prediction techniques.

Data transmission: While Low-Voltage Differential Signaling (LDVS) cables are still a common method to connect cameras directly with the Advanced Driver-Assistance Systems (ADAS) ECUs in current generation cars, there is a clear trend to use Ethernet-based bus protocols like Time-Triggered Ethernet (TTE) and Audio Video Bridging (AVB) for the transportation of video and other high bandwidth data streams from different places in the chassis to the central processing units [37]. Both protocols provide the ability to allocate fixed time slots for time critical data such as a video streams. Consequently, similar timing analysis techniques can be applied as in other cases of time triggered bus systems. However, the main difference when transmitting digital video data compared to data from other sensors is that the bandwidth requirements are generally very high and – as just discussed – the amount of data to be transmitted may vary between individual samples. As a single image will usually not fit into a single Ethernet frame, multiple time slots have to be allocated for each image. As a result the transmission might get interleaved with other high priority data streams, which increases the transmission latency of a single frame. At the same time this interleaving of different data streams also offers an opportunity for the use of P-frames: If two data streams A and B that have to share a common bus connection can be synchronized, those streams can be interleaved in such a way, that every time stream A sends an I-frame, stream B sends a P-frame and vice versa. This way, the combined bandwidth requirement would remain relatively constant and still be lower compared to a scenario where both streams only use I-frames. In any case, the migration to an Ethernet-based infrastructure with IP-based protocols, like GigE Vision [38], used on top introduces further overhead in the pipeline, and complexity in the timing analysis.

Decoding: Whatever choices are made on the encoding side of course also effects the decoding side. However, generally decoding is a lot less problematic. Among other reasons, the decoder does not have to analyze the data and/or make ad-hoc decisions on what quality level and meta-parameters to use for encoding, but can simply follow. Furthermore there is much more compute power available on modern ECU for autonomous cars – in particular in form of special circuitry, GPUs and FPGAs that can be used to accelerate the decoding process. As a result, variations in the decoding time – if any – tend to be negligible.

Image Processing: Once the image data arrives at the ECU, the actual process of analyzing the image content starts. The end goal is usually the detection and classification of various objects and features. That information is then – usually after various coordinate transformations – either forwarded to a higher level control intelligence for building maps and a general awareness of the environment or directly fed into specific control algorithms for e.g. lane control. Even more than the specific field of video compression, image processing in general is a wide research area that lately has been dominated by machine-learning-based approaches – at least for object detection and classification. Discussing all common image processing algorithms that find application in autonomous cars is outside the scope of this paper, but we will point out some properties important for timing analysis and along which the various algorithms from different domains can be classified:

- Streamability: While some algorithms require access to the whole image before processing can start, many classic image processing algorithms like encoding or convolutional algorithms (including the individual layers of Convolutional Neural Networks (CNNs)) can directly operate on individual pixels or at least sub-region of a single frame.
- Parallelizability: Can the algorithm run in parallel on multiple compute nodes and if so, up to what granularity. Parallelizability usually implies Streamability, but the converse is not true, as a streamable algorithm might need information that as been extracted from a previous chunk.
- Content-dependent run time: The run time of many algorithms (as e.g. the previously mentioned CNN) only depends on the size of the input image and hyper-parameters of the algorithm itself, which are fixed at design time. On the other hand, the run time of other algorithms like the detection and aggregation of contours [39]. If the number of iterations an algorithm has to perform is in principle unbounded, some upper limit has to be hard coded – either in form of the number of iteration steps or in terms of run time, but a timing analysis still has to be performed in order to determine what those limits should be.
- Whole image vs. region of interest (ROI): This is less a property of the algorithm itself and more about its application and implementation: Some operations need only be performed on one or multiple subregions of the image defined by an ROI value. The source of the ROI can either be a previous algorithm that ran over the same image, the result from the processing of a previous frame or from an external source (e.g. a completely different sensor or even the environment). The size, shape and number of ROIs in a given image are usually content-dependent, but contrary to the previous dimension, they are known before the algorithm is applied. So, if we e.g. run a simple CNN over a set of subareas of the image, the number of operations is known in advance, contrary to a contour detection algorithm.
- Suitability for processing on HW accelerators or even dedicated hardware circuitry: The computational demands of many computer vision algorithms by far exceed the capability of embedded CPUs and in many cases even those of high-performance desktop CPUs. For that reason, modern ECUs may be augmented by GPUs and other hardware accelerators like FPGAs. The latter can perform typical image processing operations like matrix multiplications much more efficiently due to their massively-parallel and highly configurable logic resources (Sec. III). On the one hand, this can significantly speed up the image processing and many algorithms based on (convolutional) neural networks in particular, but the additional transfers between different compute architectures poses new challenges for the design, traceability and formal analysis.

Composition of the individual stages: A recurring property across all stages is the question of streamability. I.e. can a particular stage work on individual chunks of an image one after the other or does it need access to the full image first and if it works on individual chunks at a time, what size and form do they have (e.g. compression algorithms usually operate on quadratic blocks of pixels, whereas a camera in rolling shutter mode would usually stream the data line by line). This can have a significant impact on end-to-end latency: In the worst case, where each stage starts operating only once the full image is available, the end-to-end delay from the start of the image acquisition to the end of the image processing pipeline is the sum over the full processing delays of the individual stages. It also means that between each stage there needs to be a big enough buffer to store the full image data. On the other hand, in the ideal pipeline case, where each stage can operate on chunks of the same size and the delays are the same, the total delay is only the sum over the stage delays for a single chunk plus the full processing time of the last stage. In practice, it is unlikely that each stage in the pipeline can really operate at the same rate, chunk size and chunk shape, which makes the calculation more complex. This is even more true, when processing delay and/or the amount of data produced by a stage depends on the image content (e.g. transmission delay depends on the outcome of the previous compression stage). This is further aggravated by the fact that the individual stages often need to share access to the underlying hardware resource with other tasks. This adds scheduling effects to the situation, making a perfect match of two adjacent stages even less likely.

All those effects make exploring, analyzing and, in particular, validating the different design possibilities a challenging task. Thus, detailed insights into the behavior under realistic work loads are required.

III. Processing Platform and Measurement Setup

With today's complex real-time application pipelines relying on a combination of various *parallel and sequential subtasks* to reach the performance goals, traditional CPU-only systems are often no longer sufficient. Instead, massively-parallel platforms such as GPUs and FPGAs are increasingly utilized, albeit with vastly different design and integration principles. Whilst GPUs by current design are traditional *hardware accelerators* relying on CPUs to offload individual subtasks and handle I/O, FPGAs can integrate the *entire real-time pipeline* from data acquisition via processing to signal transmission. On the other hand, GPUs are programmed on a (relatively) high level of abstraction, e.g., via NVIDIA's CUDA or tools above – whereas FPGAs require a design flow close to that of an Application-specific Integrated Circuit (ASIC). Although this yields fine-grained (i.e., register-level) control over the resulting hardware pipeline – in contrast to mostly black-boxed GPUs – and its timing, the development effort/time for FPGA-accelerated RTSs is significantly higher.

Even though not required from an operational point of view, i.e., to execute proprietary vendor drivers as in case of a GPU-accelerated system, most FPGA-based RTSs still incorporate at least one *CPU for the sequential parts* of the real-time processing pipeline and/or other application software. Often, this also includes a Real-time Operating System (RTOS) to manage the multitude of software tasks, whilst ensuring that lower-priority (e.g., management) functions do not interfere with the timing-critical subtasks of the processing pipeline mapped to software. Traditionally, such CPUs were implemented on FPGAs like all the Intellectual Property (IP) cores that comprise the hardware pipeline – i.e., by mapping them to the device's reconfigurable fabric. Whilst such *softcore* CPUs both are readily available in all major FPGA design tools and can flexibly be configured to match the application scenario at hand, using the fabric comes at a cost. Firstly, complex CPU-driven System-on-Chip (SoC) structures can require a significant amount of *FPGA resources* (e.g., logic blocks or memories), which then are unavailable to the real-time pipeline. Secondly, even a modern FPGA fabric is still considerably less efficient in terms of *integration* densities and *energy* consumption compared to a functionally equivalent ASIC implementation. In other words, mapping a fully-fledged SoC – with the same CPUs, memory controllers, interconnects and I/O peripherals – to an FPGA requires a larger device and results in higher energy consumption than an equivalent ASIC.

To combine the efficiency of ASICs with the flexibility of an FPGA, its vendors offer a range of heterogeneous devices that integrate a fixed-function SoC with one or multiple CPUs and a flexible FPGA fabric. Such *Programmable SoCs* (pSoCs) are readily available from all major FPGA vendors and range from small single-core to (mixed-architecture) many-core SoCs that are tightly coupled to FPGA fabrics of varying complexity and size – enabling the system designer to select the most suitable combination for the RTS in question. We thus introduce Xilinx' Zynq pSoC as a representative of such devices, followed by both concepts and details of our measurement methodology to capture various temporal and functional aspects of the RTS.

Based thereon, Sec. IV presents and analyzes the distributed, mixed-hardware/software processing pipeline that implements a high-speed VSA with two cameras, one heterogeneous Zynq pSoC and a traditional CPU-only node, all networked together. As our measurement infrastructure precisely covers the FPGA-accelerated RTS and its interactions over the network, both the temporal and functional implications of changes in parameters and/or vision algorithms on the control loop can be measured.

A. Current FPGA/pSoC Platforms for Real-time – Xilinx Zynq

As indicated in Fig. 2, Xilinx Zynq pSoCs comprise a fixed-function Processing System (PS) and an FPGA-equivalent Programmable Logic (PL). The PS (dark gray) features a SoC-like architecture with two ARM Cortex-A9 CPUs sharing their L2 cache, a high-speed On-Chip Memory (OCM) with 256 kByte, several bus interconnects and a variety of peripheral controllers for, e.g., DDR memory or Gigabit Ethernet (GigE). In contrast to previous heterogeneous FPGA architectures (such as Xilinx' Virtex-4 FX series), Zynq devices are far more software-driven and can boot without enabling the reconfigurable portion of the device (light gray), i.e., the Programmable Logic (PL), at first. Like within comparable parts from Intel (i.e., those with HPS),

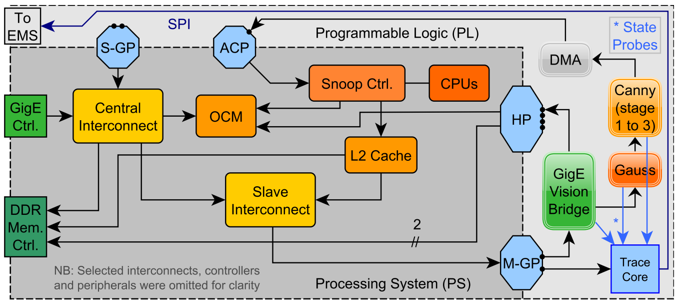

Fig. 2. Zynq pSoC: PS (left) & PL (light gray) with Image Processing Pipeline

several AXI bus ports facilitate high-speed data transfers from the PS to the PL, or vice versa. Four General-Purpose (GP), an Accelerator Coherency Port (ACP) and four High-Performance (HP) ports with integrated clock domain crossing (CDC) logic are available – although only the two Master (M) GP interfaces carry transfers from PS to PL. The remaining HP, ACP and the two Slave (S) GP ports enable custom IP cores in the PL to use the highly efficient PS components, e.g., for external storage to large DDR memories, or I/O via the PS's two GigE controllers.

The PL contains configurable logic blocks (CLBs) with flip-flops and look-up tables, BlockRAM (BRAM) and DSP slices to implement arbitrary digital functions from various types of design sources (e.g., a hardware description language, netlists or prepackaged IP cores). A programmable signal interconnect links the individual fabric blocks to each other, and to the I/O pins at the edge of the device. The clock signals to drive flip-flops, BRAMs and the other synchronous blocks, however, are distributed independently by means of global, regional and I/O clock trees. In, e.g., autonomous systems using many external sensors with high data rates, careful I/O assignment is crucial to avoid congested or unrouteable designs because of clocking.

The tight coupling between PS and PL not only enables the realization of complex heterogeneous (i.e., hardware/software) real-time pipelines, but also opens opportunities for novel data acquisition solutions for networked high-speed VSSs (Sec. IV) and our temporal-functional measurement method (Sec. III-B).

B. Hybrid Power/State-Tracing: Capture Timing and Function

Even for measurement-based analysis/verification, the complexities of the *heterogeneous hardware architectures* (Sec. I) and *processing pipelines* (Sec. II) required in, e.g., autonomous systems pose a number of challenges. Apart from synchronization to gain a holistic view of the temporal behavior across the distributed processing nodes, precise measurements on a single node are crucial for both verification and optimization. Due to their increasing heterogeneity, however, existing software-only solutions are even less sufficient, as their reach intrinsically is limited to CPUs and software. With more and more processing steps being mapped to hardware such as GPUs or FPGAs, their temporal-functional behavior has to be acquired *together* with that of the software system, necessitating a new methodology.

For FPGA-accelerated systems like the one used in our case study (Sec. IV), we thus deploy our *hybrid power/state-tracing* methodology introduced in [40]. Aiming at monitoring on the application-level (in contrast to, e.g., single CPU instructions), it combines the following concepts to enable unified temporal, functional and (if required) energy measurements on FPGA- or pSoC-based RTSs. Firstly, the device-under-test (DUT) at hand is *instrumented* to capture various temporal and functional data points whilst the real-time pipeline is running. On the software side acquired via insertion of lightweight memory-mapped I/O (MMIO) accesses, this includes, e.g., application information (such as timestamps of individual processing steps, frame IDs, or control or actuation data), CPU load or scheduling statistics. On the hardware (i.e., PL) side, sequence counters, IDs, state-machine transitions or interrupt request (IRQ) information has already been used successfully. The instrumentation is realized by means of a highly resource-efficient *Trace IP Core* that will be instantiated in the PL, as shown in Fig. 2 (bottom right). It not only receives a variety of state signals from the hardware processing pipeline in the PL (e.g., Gauss and Canny filtering stages) or the IRQ line of, e.g., a direct memory access (DMA) controller, but also features an optional bus interface. The latter enables low-overhead instrumentation of software applications and/or RTOS via uncached MMIO writes to the core's internal logging registers. The Trace Core then converts captured state information from the PL state probes (light blue in Fig. 2) and the software instrumentation into a serial data stream, which is then sent to an *External Measurement System* (EMS) attached to the DUT using a three-wire serial peripheral interface (SPI).

Apart from precisely timestamping and storing the incoming state stream, the EMS may also capture voltages and currents on the supply rails of DUT and external I/O components such as GigE PHYs. Based thereon, the tracing coverage can further be increased to timing-relevant *I/O events* (detected via current changes on selected I/O rails) and to *DUT Energy Monitoring*. Even though commercial data acquisition (DAQ) systems such as National Instruments's PXI series can be used as EMS, care has to be taken during generation of the state timestamps (due to the jitter of their often software-based SPI implementations). For our evaluations, however, we rely on a purpose-built EMS implementation that features an FPGA Mezzanine Card (FMC) interface to connect to the DUT. Apart from two GigE PHYs, our FMC-sized EMS primarily integrates an 18-channel analog DAQ subsystem with a resolution of 16 bits and sampling rates beyond 200 kSPS (kilo-samples per second) for analog and, at the moment, 25 kSPS for states [41]. The ratio between analog and state samples can be adjusted, although the current choice of 8:1 turned out to be adequate for all measurement scenarios we encountered so far. A timer with sub-μs resolution is used to generate a hybrid power/state trace that is eventually stored on a μSD card, enabling acquisition durations of over an hour.

Benefits of Hybrid Tracing: Based on all the voltage, current and state information contained in the hybrid traces, numerous crucial temporal, functional and energy-related characteristics of an FPGA-accelerated RTS can be extracted. A Python-based measurement tool first reads analog and state samples – relying on the precise timestamps captured for each data point. Based on the application (i.e., PS/PL) information stored in the state trace, *functional RTS characteristics* such as application states, adaptive control gains or the sequence number of the currently processed image frame are then found. As state changes often correlate with crucial *temporal application events* such as the start of a new iteration, the DUT-internal timing is accurately reproduced. Together with additional events from monitoring of I/O components, the *end-to-end processing latencies* – from reception of a sensor signal to transmission of a corresponding actuation value – are also covered. Jointly, the temporal events enable an identification of individual processing phases within the application that, once combined with fine-granular energy readings (generated by integration via the precise timestamps), yield *per-phase/component energy* values helpful for optimization of the RTS. In case the DUT is communicating with other nodes over the network, even inter-node (e.g., sensor) latencies can be captured, as demonstrated by our case study in Sec. IV.

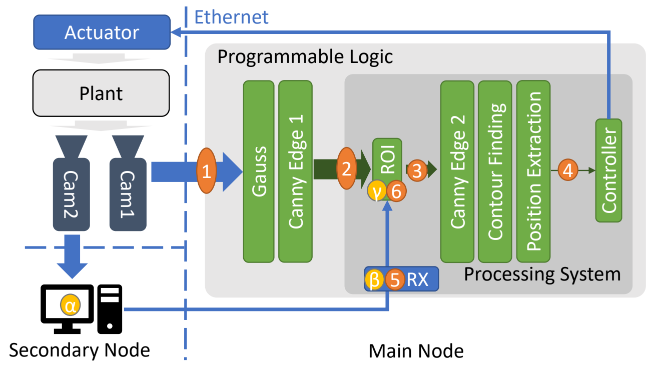

Fig. 3. Case Study: Vision-based control

IV. Case Study

To demonstrate the capabilities of our proposed RTS tracing infrastructure, we extended the Visual Servoing System (VSS) originally presented in [42] towards a representative workload for distributed autonomous systems with multiple cameras. It combines a heterogeneous Zynq-based RTS (implementing the main control pipeline) with a software-only secondary node, each driven by a GigE Vision camera over a shared network. This setup serves as a surrogate for an autonomous vehicle, where the camera data is used in closed-loop real-time control.

A. VSS Application Scenario

A hollow steel hemisphere is kept afloat in a variable magnetic field generated by a current-controlled coil. The position of the hemisphere is continuously captured by a GigE Vision camera that sends the video stream via UDP over an Ethernet link to our instrumented Zynq-based processing node (Sec. III-A). It processes the individual camera frames in a multistage pipeline, which is implemented partially in the Programmable Logic (PL) and partially running in software on the Processing System (PS), and has both convolutional and iterative stages. The output of the pipeline is the hemisphere's estimated position, which is then fed into a standard PID controller that calculates the required current in the coil and sends that value – again via UDP – to the control circuit of the electromagnet. The main difference to the setup in [42] is that we now have a second camera that also tracks the hemisphere, albeit at a lower resolution and its processing is happening on a separate compute node. The feature data from this secondary node are also transmitted to the processing node (PN) and used to cut out an ROI for the software steps in the primary image processing pipeline. While this exact scenario might not have a direct equivalence with applications in current autonomous cars, we believe the workload to be comparable to many common tasks in the

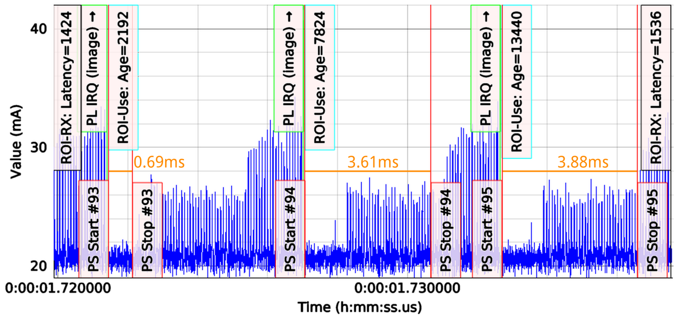

Fig. 4. Hybrid Power/State Trace of mixed-hardware/software VSS on Zynq

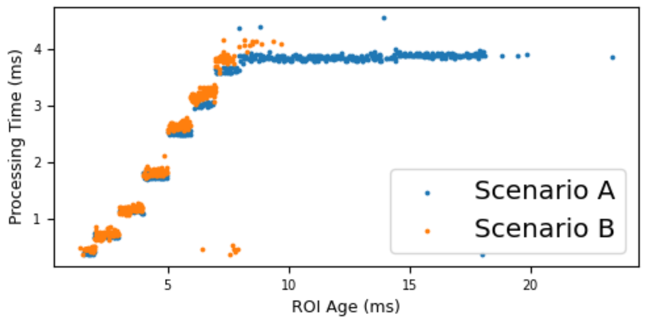

Fig. 5. PS Processing Time vs. ROI Age ($\gamma - \alpha$) for individual Frames

ADAS domain, such as lane assists and object detection. Here, the second stream of data could, for example, represent sensor information that is provided by the car in front, which can provide the trailing car with a ”glimpse into the future” or just film the same object from a different perspective.

As is to be expected from externally provided sensor data in the real world, the two cameras are not synchronized, may run at different frequencies and the second compute node is not a real-time system, which adds jitter to the ROI stream. Consequently, the ROI data can be of varying timeliness and, hence, accuracy with respect to the frame currently processed on the primary system. The main control pipeline accounts for that by adapting the size of the ROI window depending on the age of the ROI data (based on the time stamps taken by the application logic at (α), (β) and (γ) in Fig. 3 which are). The time stamp (α) is propagated along the ROI data packet, (β) is taken when that package arrives (for instrumentation purposes) and (γ) is taken every time that information is actually used to cut out an ROI from an incoming frame.

B. *Captured Trace Data*

For this case study, we instrumented various events on the main node, marked (1) to (6) in Fig. 3. At (1), we monitor the PHY's power draw that directly correlates with the incoming camera Ethernet frames. At (2), (3) and (4), we generate events (in software or hardware) when one stage of the processing pipeline is complete and the data are handed over to the next. (5) traces the arrival of new ROI data and (6) the moment that ROI data get actually used in the image pipeline. In addition to generating an event, (5) and (6) also embed the ROI's age as estimated by the application ($\beta - \alpha$ and $\gamma - \alpha$, respectively) in the trace. A short excerpt of such a trace is shown in Fig. 4. The blue curve in the background shows the current on the PHY's I/O rail, identifying the incoming Ethernet frames (56 per camera frame). The first label (ROI-RX) corresponds to an incoming ROI packet (5), the next three events (PL-IRQ (2), ROI-Use (6) and PS Start (3)) are generated almost simultaneously when the processing on the PL is finished and the PS starts to work on the ROI subarea. In this particular scenario, the ROI data are only transmitted at 60 Hz. When work begins on the first frame shown (#93), the ROI information is fresh, and, thus, the processing time on the PS (PS Start to PS Stop) is very short. For the next frames, however, the ROI information is getting older and, hence, the ROI gets expanded, in turn causing longer processing times.

There are a number of methods and tools for formal modeling and analysis of control algorithms, both at design and at run time. However, all of these tools need timing-annotated models. Getting accurate timing annotations, when complex vision processing is a part of the control loop is difficult, and this is one of the several ways in which our tracing and measurement infrastructure can be useful. The timing-annotated events in Fig. 3 can decorate formal models, e.g., to ensure that certain invariants are satisfied.

Fig. 5 shows a scatter plot of the recorded processing latencies against the respective ROI age for two configurations of the system. Scenario A is the same that Fig. 4 was generated from, with an ROI update rate of 60 Hz. In scenario B, the ROI data were updated at 160 Hz (and thus approaching the frequency of the main control loop running at 178 fps). We clearly see how the processing latency increases with the age of the ROI. The clustering in the y dimension happens because the algorithm computing the size of the ROI operates at a millisecond resolution, while the tracing tool and the age computation work at a much higher resolution. The maximum observed execution time of about 4 ms is reached when the ROI extents to the whole image. As expected, this happens much more rarely in the 160 Hz case, but surprisingly, we can also see that for the same ROI age there are more incidents of higher processing times. This is indicated by several orange clusters residing above their 60 Hz blue counterparts in Fig. 5. Such non-intuitive observations provided by our tracing and measurement infrastructure will be useful for design and verification, and would be difficult to obtain otherwise.

In summary, we get detailed quantitative insights into the sequence of steps contributing to the overall image processing latency and how they are coupled to external influences and with each other. This data can either be used for directly debugging and evaluating a given design or as a basis for statistical models in temporal verification tools, which significantly helps to overcome the challenges outlined in Sec. II.

V. Conclusion

Response times of autonomous systems are highly dependent on the time for acquisition, transmission, processing and interpretation of high-resolution video data streams. Not only pose many of those steps difficult problems for formal timing analysis techniques in isolation, their interdependent and data-dependent run times make tight estimates on the WCRT next to impossible. We believe that in order to fully understand emergent temporal properties, possible design trade-offs and optimization opportunities, coherent measurement and tracing facilities are necessary to enable an easy correlation of events across various software and hardware components. In this paper, we presented a low-overhead tracing solution for FPGA-accelerated RTSs and demonstrated how it can be used to analyze and correlate the events on a VSS's main control node. An interesting next step in this work will be the efficient and effective combination of trace data from multiple different instrumented nodes across a distributed systems.

References

[1] R. Metta, M. Becker, P. Bokil, S. Chakraborty, and R. Venkatesh, "TIC: a scalable model checking based approach to WCET estimation," in *17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, 2016.

[2] M. Becker, R. Metta, R. Venkatesh, and S. Chakraborty, "Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors: a comeback of model checking," *STTT*, vol. 21, no. 5, pp. 515–543, 2019.

[3] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty, "Performance debugging of esterel specifications," *Real-Time Systems*, vol. 48, no. 5, pp. 570–600, 2012.

[4] ——, "Performance debugging of esterel specifications," in *6th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2008.

[5] S. Chakraborty, T. Erlebach, and L. Thiele, "On the complexity of scheduling conditional real-time code," in *7th International Workshop on Algorithms and Data Structures (WADS)*, ser. Lecture Notes in Computer Science, F. K. H. A. Dehne, J. Sack, and R. Tamassia, Eds., vol. 2125. Springer, 2001, pp. 38–49.

[6] W. Zhao, W. C. Kreahling, D. B. Whalley, C. A. Healy, and F. Mueller, "Improving WCET by applying worst-case path optimizations," *Real-Time Systems*, vol. 34, no. 2, pp. 129–152, 2006.

[7] J. Coffman, C. A. Healy, F. Mueller, and D. B. Whalley, "Generalizing parametric timing analysis," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2007.

[8] "aiT WCET analyzer from AbsInt Angewandte Informatik GmbH," https://www.absint.com/ait/.

[9] "RapiTime Aero and Auto WCET analyzers from Rapita Systems Ltd." https://www.rapitasystems.com/products/rapitime.

[10] M. Jersak, K. Richter, R. Henia, R. Ernst, and F. Slomka, "Transformation of SDL specifications for system-level timing analysis," in *10th International Symposium on Hardware/Software Codesign (CODES)*, 2002.

[11] S. Chakraborty and L. Thiele, "A new task model for streaming applications and its schedulability analysis," in *Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2005.

[12] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein, "System-level timing feasibility test for cyber-physical automotive systems," in *11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016.

[13] "chronVAL timing analysis tool from Inchron GmbH," https://www.inchron.com/tool-suite/chronval/.

[14] "LuxTrace timing analysis tool from Luxoft," https://www.luxoft.com/pr/launch-of-luxtrace-accelerates-trace-timing-analysis-by-10x-in-the-automotive-sector/.

[15] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty, "Generalized asynchronous time-triggered scheduling for flexray," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 2, pp. 214–226, 2017.

[16] M. Becker, S. Mohamed, K. Albers, P. P. Chakrabarti, S. Chakraborty, P. Dasgupta, S. Dey, and R. Metta, "Timing analysis of safety-critical automotive software: The AUTOSAFE tool flow," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2015.

[17] L. Zhang, R. Schneider, A. Masrur, M. Becker, M. Geier, and S. Chakraborty, "Timing challenges in automotive software architectures," in *36th International Conference on Software Engineering (ICSE)*, 2014.

[18] M. Lukasiewycz, M. Glaß, J. Teich, and S. Chakraborty, "Exploration of distributed automotive systems using compositional timing analysis," in *Embedded Systems Development, From Functional Models to Implementations*, A. L. Sangiovanni-Vincentelli, H. Zeng, M. D. Natale, and P. Marwedel, Eds. Springer, 2014, pp. 189–204.

[19] R. Schneider, U. D. Bordoloi, D. Goswami, and S. Chakraborty, "Optimized schedule synthesis under real-time constraints for the dynamic segment of flexray," in *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2010.

[20] M. Kauer, S. Steinhorst, R. Schneider, M. Lukasiewycz, and S. Chakraborty, "Automata-theoretic modeling of fixed-priority non-preemptive scheduling for formal timing verification," in *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 812–817.

[21] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time," *Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, 2017.

[22] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[23] L. Kosmidis, E. Quiñones, J. Abella, T. Vardanega, C. Hernández, A. Gianarro, I. Broster, and F. J. Cazorla, "Fitting processor architectures for measurement-based probabilistic timing analysis," *Microprocess. Microsystems*, vol. 47, pp. 287–302, 2016.

[24] I. Wenzel, R. Kirner, B. Rieder, and P. P. Puschner, "Measurement-based timing analysis," in *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2008.

[25] S. Chakraborty, M. D. Natale, H. Falk, M. Lukasiewycz, and F. Slomka, "Timing and schedulability analysis for distributed automotive control applications," in *11th International Conference on Embedded Software (EMSOFT)*, 2011.

[26] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty, "Multi-objective co-optimization of flexray-based distributed control systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[27] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty, "Constraint-driven synthesis and tool-support for flexray-based automotive control systems," in *9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

[28] D. Goswami, P. Seshadri, U. D. Bordoloi, and S. Chakraborty, "A DECOMSYS based tool-chain for analyzing flexray based automotive control applications," in *IEEE Conference on Automation Science and Engineering (CASE)*, 2009.

[29] R. Ernst, "Automated driving: The cyber-physical perspective," *IEEE Computer*, vol. 51, no. 9, pp. 76–79, 2018.

[30] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi, "Building timing predictable embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, pp. 82:1–82:37, 2014.

[31] U. Karkaria, "New heart of the car: The GPU, Automotive News, July 30, 2018," https://www.autonews.com/article/20180730/OEM10/180739942/new-heart-of-the-car-the-gpu.

[32] "NVIDIA solutions for self-driving cars," https://www.nvidia.com/en-us/self-driving-cars/.

[33] J. Yoshida, "Where do FPGAs stand in Auto IC race? EETimes, June 26, 2018," https://www.eetimes.com/where-do-fpgas-stand-in-auto-ic-race/.

[34] "Automotive FPGA solutions from Intel," https://www.intel.com/ content/www/us/en/automotive/products/programmable/overview.html.

[35] G. A. Elliott, K. Yang, and J. H. Anderson, "Supporting real-time computer vision workloads using openvx on multicore+gpu platforms," in *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[36] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[37] L. L. Bello, "Novel trends in automotive networks: A perspective on ethernet and the IEEE audio video bridging," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, A. Grau and H. Martínez, Eds. IEEE, 2014, pp. 1–8.

[38] AIA, "GigE Vision Main Page - AIA Vision Standards," http://www.visiononline.org/vision-standards-details.cfm?type=5.

[39] S. Suzuki and K. Abe, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, pp. 32–46, 1985.

[40] M. Geier, M. Brändle, D. Faller, and S. Chakraborty, "Debugging fpga-accelerated real-time systems," in *2020 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[41] M. Geier, D. Faller, M. Brändle, and S. Chakraborty, "Cost-effective energy monitoring of a zynq-based real-time system including dual gigabit ethernet," in *2019 27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[42] M. Geier, F. Pitzl, and S. Chakraborty, "GigE vision data acquisition for visual servoing using sg/dma proxying," in *2016 14th ACM/IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*.