12.1 Motivation

Automated driving (A.D.) requires concurrent execution of multiple complex driving functions on automotive embedded platforms. In general, such systems can be partitioned into early stages including sensor processing, individual perception, and cognition functions and into later, more centralized stages that perform data fusion, planning, and decision making. In this chapter, we exemplarily concentrate on automotive embedded processing systems for perception and cognition problems, however, we expect similar problems also on later stages such as data fusion. For perception and cognition, one can observe a wide gap between required processing power and the achievable embedded realizations which have to fulfill non-functional requirements such as low power and small cost. Furthermore, these systems must perform all processing under strict safety requirements that guarantee deadlines and provide high system robustness.

Regarding their development, there is a lack of efficiency when applying established methodologies to modern heterogeneous architectures that employ hardware acceleration to meet performance requirements. Finally, there is a lack of hardware architectures that can scale towards the ever-growing processing requirements. In this chapter, we first identify these gaps based on a case study of a single perception and cognition function. Afterwards we present novel solutions that tackle these gaps.
individually. Finally, we introduce a powerful framework for modern automotive platform design based on these solutions.

12.2 State-of-the-Art

This section consists of two parts: First, we give an overview on current heterogeneous automotive processing hardware and discuss state-of-practice for automotive timing analysis and multitasking. Then we discuss embedded prototyping. Focusing on perception and cognition, we then present a case study based on our previous work [7] to identify current deficiencies of embedded automotive processing architectures and their development.

12.2.1 Overview on Current Embedded System Design

We first give an overview on current heterogeneous embedded platforms for advanced driver assistance systems (ADAS) and automotive processing before we summarize current automotive software development with regards to timing analysis, automotive multitasking, and prototyping. The shortcomings of current design methods are then identified based on the case study.

12.2.1.1 Current Embedded Automotive Hardware

Currently, automotive embedded architectures rapidly change in favor of complex heterogeneous system-on-chips (SoCs) that do not only integrate multiple embedded CPU cores but also hardware accelerators. Table 12.1 gives a summary of

Table 12.1 Overview on current high-performance embedded ADAS system-on-chips (selection)

| Product          | Core types                                                                 | Accelerators                                      |
|------------------|-----------------------------------------------------------------------------|---------------------------------------------------|
| TI TDA2x [40]    | 2x ARM Cortex-A15, 2x ARM Cortex-M4, 2x TI C66x DSP                        | 4x TI EVE Vision accelerator                      |
| TI TDA3 [41]     | 2x ARM Cortex-M4, 2x TI C66x DSP                                            | 1x TI EVE Vision accelerator                      |
| nVidia Tegra K1 [25] | 4+1 ARM bigLittle Cortex-A15                                                | 6x12 nVidia Kepler Shaders                        |
| nVidia Tegra X1 [26] | 4+4 ARM bigLittle Cortex-A57/A53                                           | 2x16x8 nVidia Maxwell Shaders                     |
| Mobileye EyeQ3 [38] | 4x MIPS32 1004K [31]                                                        | 4x Vector Microcode Processors                    |
| Mobileye EyeQ4 [23] | 4x MIPS InterAptive, 1x MIPS M5150 peripheral core                         | 6x Vector Microcode Processors, 2x Multithreaded Processing Cluster, 2x Programmable Macro Array |
prominent architectures on the market and their architectural features. Examples for these complex SoCs come from vendors such as nVidia or Texas Instruments but also from more specialized companies such as Mobileye [21]. Heterogeneity differs in type and number of cores or accelerators. Accelerators include arrays of GPU shaders as well as fixed-function and programmable accelerators. Regarding safety qualification, in contrast to nVidia’s chips, EyeQ4 is designed to provide ISO 26262 ASIL-B(D) [23] while Texas Instruments’ TDA2x provides AEC-Q100 qualification [2].

12.2.1.2 Timing Analysis for Automotive Systems

Timing analysis and verification is a crucial task in automotive system development. Such a timing analysis is typically already integrated in model-driven and component-based development workflows on highest abstraction level. The AUTOSAR timing extensions [4] allow for specification of timing properties on component level but also the specification of end-to-end timing requirements over component boundaries. By modeling dependencies and annotating periodicity, early schedulability analyses can already be executed on such high-level models. One prominent automotive tool is SymTA/S [39] which integrates low-level analysis results into a model- and component-based flow and can be coupled to ECU software development tools such as Ascet [12]. Early timing behavior can be validated using provided simulation models for generic schedulers, bus systems, and RTOS, such as the AUTOSAR OS. In model-based flows, for early analysis, manual estimation of timing properties is still state-of-the-art (even for space missions [14]), while timing models can be refined iteratively based on updated results from low-level static analysis or timing measurements (e.g., execution on ECU hardware during a test drive to extract timing data) on generated code.

For tighter, lower level, estimates, timing analysis that considers application knowledge can be done manually (see analysis of the pipeline of Fig. 12.1 in [8]), or more generalized based on timing-annotated graphs [13]. Manual analysis is cumbersome and not generalizable. Here, guaranteed worst-case execution times can only be given when largely overestimating execution time although theoretical worst-cases practically never occur on real executions (cf. Sect. 12.3.2.2).

Graph-based modeling such as scenario-aware-dataflow graphs (SADFG) [13] considers data dependencies and control flow dynamics on a more generic level, e.g., by modeling them as scenarios. Both, model-based analysis and graph-based

![Fig. 12.1 Structure of the software pipeline for camera-based assistance systems (adapted from [24])](image-url)
analysis, share the fact that they require the annotation of timing properties which are (iteratively) extracted from lower level tools from implementation results: For example, SymTA/S allows the import of timing properties from the Absint aiT \[1\] static analysis tool or from recorded trace data from execution on real hardware.

Measurement-based extraction of timing data introduces big uncertainty \[3\] due to the small coverage of real-world scenarios. For automotive manufacturers, this manifests in the state-of-practice of many required test drive kilometers. On the other hand, commercially available static timing analysis tools such as Absint aiT \[1\] are not available for complex microarchitectures with features such as out-of-order execution or SMT, both of which can be found in current, commercially available, high-performance automotive platforms (cf. Sect. 12.2.1.1).

Furthermore, the trends towards automotive multicores and hardware acceleration introduce side-effects that complicate such analyses. Examples are common access to shared resources such as shared Level 2 caches but also the side-effects of DMA transfers from other masters in the system, such as accelerators \[3\]. Schmidt et al. \[33\] showed that such effects already occur on current automotive multicores. Also, low-level static analyses usually require application knowledge, such as bounds on loop iteration counts, which can be hard to establish for complex A.D. algorithms while limiting overestimation to reasonable margins (cf. Sect 12.3.2.2).

12.2.1.3 Automotive Multitasking

Multitasking systems for automotive processing have to consider strong timing requirements. As apparent from the previous section, those requirements can be annotated early when using a component-based or model-based workflow. Then, tools can provide schedulability analyses or simulation of real-time schedulers such as earliest deadline first \((EDF)\). Static (cyclic) scheduling is also common when using AUTOSAR \[22\]. Here within a scheduling hyperperiod, multiple AUTOSAR runnables are scheduled at fixed offsets within smaller time-slots \[22\]. Such static scheduling can especially be observed for near-sensor or data fusion driving applications which are typically tightly synchronized to input data capturing rates. Here, the length of scheduling slots is usually coupled to the input sensor update rates, such as camera frame rates for perception problems. Scheduling within Mobileye EyeQ1 is an example for such static scheduling \[37\]. In EyeQ1, timeliness of execution times against the static slot periods was verified using simulation on large amounts of recorded data from test drives. While static scheduling provides highest predictability, it limits concurrent utilization of processing resources, especially with growing number of concurrent pipelined applications (cf. Sect. 12.2.3.2).

12.2.1.4 Current Embedded Prototyping

Prototyping by utilizing simulations of a full system, so-called virtual prototypes, has emerged as a potential solution for many challenges in general embedded system
development. However, the approach is yet to be adopted by many automotive software developers and system designers. The key issue of virtual prototyping is approximating system properties such as accurate timing behavior of target software at a simulation performance that makes a practical application viable. The needed trade-off between performance and accuracy can vary over designs and development progress. For example, exact timing provides little advantage in early stages of development when a system and thus its timing will still undergo large changes. Here we give a brief overview of simulation abstraction levels during an idealized development process.

During pre-development or early development, abstract prototypes can be constructed using tools such as Simulink [43] or usual desktop programming. These prototypes enable activities such as algorithmic development or training of models for machine learning. However, due to the high level of abstraction, an accurate prediction of most product properties is impossible.

For automotive embedded real-time systems, timing behavior is as crucial as functional behavior. Therefore a target hardware platform has to be selected considering its timing but also its cost. Virtual prototypes that provide timed simulations enable a systematic exploration of the design space, such as potential platforms and hardware/software partitions. SoC vendors rely on detailed hardware models (e.g., of the CPU cores) that closely approximate hardware at this stage [9]. However, this approach is unrealistic for application development—for example, by an automotive OEM—due to two reasons: Firstly, the abstraction of these models is typically orders of magnitude too low to allow a fast timed simulation of real applications. Secondly, if the sole purpose is selecting a suited target platform, the high licensing cost of such detailed architectural models is not justified, as a large library of many of those models would be required.

Pressler et al. [29] propose a solution based on the estimation of platform independent source code and the computational cost of operations on a given target. While this approach, due to the higher level of abstraction, certainly has a lower accuracy than more detailed simulations, it is less subjective and much faster than alternatives (e.g., selecting based on benchmark results).

System-level simulations approximate a full system by co-simulating hardware and software models that have been compiled for the simulation host. They can be applied during most development stages and offer a wide range of possible trade-offs between simulation accuracy and performance. Software is simulated based on either source or binary code. Source-code-based approaches (source-level simulations, SLS) rely on heuristics [9, 17, 35, 36] to accurately reflect non-functional properties such as timing of target binary code. This step can introduce an additional error compared to approaches based on binary code, the so-called binary-level simulation (BLS). Hardware is usually modeled in SystemC [15], a standard language for system-level simulation. Abstraction of hardware can range from techniques such as TLM+ [11], where bus transfers are simulated for whole application-level object (e.g., a full video frame), to a detailed simulation of internal hardware signals.
12.2.2 **Challenges in Hardware/Software Co-design for Automated Driving: A Case Study**

In this section, we exemplarily demonstrate the deficiencies of the current design of automotive processing architectures, with a bias towards perception and cognition, based on a case study (also see our previous work [7]) that covers the development of a heterogeneous embedded realization of a complex driver assistance function, using a state-of-the-art approach. We then discuss the shortcomings of such classic development and propose novel solutions.

12.2.2.1 **A Vision-Based Traffic Light Detection**

Our case study is concerned with an efficient implementation of a camera-based traffic light recognition (TLR) system. The targeted algorithm [24] consists of a multi-level image segmentation for finding light points, a search for the surrounding traffic light boxes and a subsequent classification of found candidates by a support vector machine (SVM). Candidates are tracked over time and classification results are fused temporally over multiple frames. The application’s output are the detected light positions and their states. Such a result can be used by subsequent applications such as an intersection assistance which is a key component in automated driving. On 1280 × 720 color video data it could achieve about 25 frames per second when executed on an Intel Core i7-2820QM quad-core general-purpose development PC, outperforming state-of-the-art approaches in both, detection robustness and speed (when executed on a PC platform). Figure 12.1 shows the generic algorithmic pipeline structure that is used by the TLR and also applies to other perception and cognition applications.

12.2.2.2 **Systematic Hardware/Software Co-design**

As the algorithm evaluates the results from a history of several frames before deciding on a recognition result, a high average frame rate, ideally near 24 fps, is targeted. Our workflow for realizing the recognition under embedded constraints is depicted in Fig. 12.2. Based on past experience together with the performance results from executing the application on the high-performant developer platform, we concluded that we most probably would not be able to realize the application at required performance on an embedded platform, when executed in software only.

Therefore, the first design decision was choosing the Xilinx Zynq SoC [47] as target platform. It does not only contain a powerful ARM Cortex-A9 dual core CPU but it also allows for custom hardware accelerators by providing an on-die FPGA logic fabric. Also, for maximum possible software performance and minimized overhead, we additionally decided to use no operating system but only a small hardware abstraction layer that contains basic library and I/O support (bare metal programming). Thus we save the overhead from device drivers, virtual memory, and also keep memory footprint low.
Starting point was the x86 architecture implementation from [24] of the assistance function on a PC. First, an initial port to the target ARM CPU architecture had to be done, which included replacing a large number of libraries by custom implementations as they were relying on an operating system or were not available for the ARM architecture. Then, preparations for profiling this software port were done. For being able to receive test drive video data from a host PC, a hardware interface for GigE had to be provided together with adding a UDP software stack.

Additionally, the software port was instrumented by adding calls to the ARM CPU timers around individual functional blocks to allow for timing measurements. This port of the software was then executed for profiling, where input video data was fed over gigabit Ethernet.

While memory consumption was fair (about 22 MiB), the performance profiling results of this solution were orders below the targeted goal: Processing one single video frame could take multiple seconds, resulting in a average-case throughput of 0.5 frames per second. As a major bottleneck we identified the classification step which uses a computationally high-demanding SVM. Also, it could occur that several tens of traffic light box candidates would be selected for classification by the segmentation stage. The selection itself was performed by solving a complex optimization problem which discarded candidates based on iterative scoring. We replaced this approach by a simpler plausibility check that processes a list of rules that includes color and geometry checks and still showed robust detection while lowering complexity of decision making at similar detection rates.

We also performed a manual data type analysis for the SVM’s feature vector elements, testing different data resolutions against the introduced error. It was
revealed that we could replace the double data types by an 8-bit type without affecting overall classification results (see [7] for the detailed error analysis). Still, iterative profiling runs showed that hardware acceleration would be needed for image segmentation and classification. The design was then manually partitioned into hardware and software components where hot-spots are tackled by dedicated hardware accelerators whereas uncritical program paths remained implemented in software. For segmentation we developed a generic filter accelerator structure that allows for on-the-fly image processing by only intermediately storing local image lines. This generic architecture is then instantiated multiple times to form a structure that implements a more complex morphological top-hat transform. The formula of the radial basis function (RBF) kernel within our SVM was reformulated [7] to express the kernel (see [30] for an introduction on SVMs) as a large running multiplication, where the argument is a function that—due to the 8-bit feature elements—can only assume 256 different values. We implement a fully pipelined approach processing eight vector elements in parallel within each clock cycle using four dual-port on-chip SRAM lookup tables followed by a multiplication tree and an accumulator register. Also, we hold the candidate vector in local memory preventing costly fetches from external DRAM for every single vector comparison. Figure 12.3 shows the overall accelerator design. Synthesizing hardware accelerators, interconnect and compiling the software finally led to our executable hardware/software platform which then could be evaluated with regards to performance, functional equivalence, and power consumption.

Fig. 12.3 Hardware accelerator for the RBF kernel of the support vector machine. Four dual-port lookup tables are employed together with a multiplier tree, forming a fully pipelined design
Using this step-by-step methodology, we achieved 12 fps minimum and a 46 fps average detection performance while only consuming approx. 5 Watts total system power at similar recognition results as the original x86 implementation.

### 12.2.3 Obstacles to Efficient Realizations of Automated Driving

Guided by our experiences from the case study of a single perception and cognition function, we now identify major shortcomings of current development of automotive processing architectures.

#### 12.2.3.1 Performance Gap

As apparent from our case study, even for single-application systems, there is a large gap between the available processing power of embedded CPUs and the computational demands of modern assistance/A.D. functions. This can already be seen in the trend towards heterogeneity in current automotive SoCs which answer those demands via hardware acceleration, e.g., GPU shader arrays.

While heterogeneity may ease the performance gap, growing heterogeneity also leads to changes in automotive software development. Programmable accelerators may enforce usage of non-standard or proprietary programming languages, which makes code platform-specific and much less portable than (MISRA—[20]) C often used in homogeneous embedded automotive platforms. For example, for general-purpose programming the GPU shaders of the nVidia Tegra K1/X1 SoCs, their custom CUDA programming model is mandatory, as at the time of writing, no openCL support was available.

#### 12.2.3.2 Utilization Gap

As also apparent from our case study, hardware acceleration may be mandatory to efficiently implement complex future A.D. functions. Costs for chip area of those accelerators have to be justified by high utilization as for a given process technology, manufacturing cost of a die exponentially increase with die area (cf. [46, Eqs. (14.5), (14.7)]). Guaranteeing such high utilization might already be difficult when accelerators are assigned to single applications and when static scheduling is used as it is state of the art in many automotive systems. However, with the growing number of functions needed for automated driving, it has to be made sure that accelerators can be utilized by multiple applications to avoid costly over-design and over-provisioning of resources. This is especially true when accelerators can only be used within certain stages of application pipelines, running idle in other stages (as our accelerator for classification, Fig. 12.3, used within our pipeline model, Fig. 12.1).
Static scheduling as introduced in Sect. 12.2.1.3 gives highest predictability, but due to the inherent dynamics of input data and subsequent execution time, big over-provisioning has to be applied to the common static slot period, leaving slots highly underutilized in the average case. Again, this dilemma grows with growing number of concurrently executed applications.

12.2.3 Development Gap

Prevailing approaches to developing automotive embedded systems, usually electronic control units (ECUs), rely on target hardware to evaluate implementation properties such as performance. This has numerous drawbacks: The effort required for a preliminary implementation restricts the number of evaluated platforms. In the case of our case study, development took several months due to the system complexity. A preselection can often only be based on imprecise factors such as benchmark results or developer experience.

Furthermore, observability in modern SoCs is restricted, as classical debugging aids such as in-circuit emulation are not feasible anymore.

12.2.3.4 Scalability Gap

While current automotive systems employ multiple embedded CPU cores and accelerators, their symmetric multiprocessing together with currently used interconnect will eventually hit a scalability gap. Regarding interconnect, buses, crossbars (as currently found on automotive SoC, cf. Sect. 12.2.1.1), but also ring interconnects do not scale well towards large numbers of processing elements (PE) such as CPU or accelerators [16, 32]. As authors of [32] point out: Buses are limiting due to the serialization of concurrent requests. Crossbars, while limiting serialization, suffer from quadratical area and power demand wrt. number of PEs, while average hop count for rings grows proportional to the number of PEs. The current crossbars and shared memory architectures of current SoC will eventually not be sufficient anymore to fulfill the increasing demand of concurrent driving functions.

12.3 Concepts for Efficient Future Automated Driving

In this section we present solutions to individually tackle the challenges discussed in Sect. 12.2.3: overcoming the performance gap, the utilization gap, as well as the development gap and the scalability gap.
12.3.1 Bridging the Performance Gap with Heterogeneity

As successfully implemented in our case study, for bridging the gap between demanded and available processing power in embedded realizations, we propose hardware acceleration. For not equally enlarging the utilization gap, these accelerators should not be application-specific but should allow for reconfiguration towards usage by a large number of concurrent applications. For example, for our RBF Kernel accelerator this means that the application-specific lookup tables (cf. Fig. 12.3) can be reloaded at runtime to contain different trained classification models.

For design reuse and scalability towards multiple instantiations, hardware accelerators (HWA) should share a common interface separating configuration from input/output data transfers. Thus, only the communication channels have to be adapted when changing from a classic bus interconnect to a modern network-on-chip (NOC). This also allows for simple addition of novel accelerator types.

Furthermore, hardware safety measures are employed: Configuration of the DMA’s memory addresses is only possible via privileged access, not allowing the DMAs to be accidentally configured to overwrite foreign memory areas. The accelerator structure is depicted in Fig. 12.4.

12.3.2 Bridging the Utilization Gap

Hardware acceleration for demanding computational tasks can only be cost-efficient when high utilization is ensured. Otherwise additional chip area cost may not be justified (cf. Sect. 12.2.3.2). However, sharing resources in the context of usually safety-relevant systems has to be carefully designed and verified towards safety requirements. We now present means on how such high utilization can be achieved.
12.3.2.1 Runtime Resource Management

In our previous work [8], we give a twofold approach to ease the utilization gap in heterogeneous embedded systems for automated driving. Our approach consists of two main contributions:

1. Decoupling of input data capturing from processing to increase utilization while at the same time shortening system response time.
2. Inter-application resource sharing that also considers safety aspects.

Decoupling of Input Data Capturing from Processing Our approach targets applications that decide on a vehicle reaction based on observing the environment over a certain period of time. Practically, this means that such a decision is based on a temporal fusion of results from past application iterations (i.e., multiple executions of the full application’s pipeline, see Fig. 12.5). Many driver assistance applications follow such a scheme.

In our approach, applications follow a common software pipeline model similar to that of Fig. 12.1. Adhering to this pipeline scheme, we use two camera-based applications, a TLR (also see Sect. 12.2.2 for details) and a traffic sign recognition (TSR) which performs a circle-detection-based segmentation into Region-of-Interest, which are then classified by an SVM against a trained model using a radial-basis-function kernel.

Both algorithms further track results over time and can robustly decide on a reaction based on the last 15 iterations. Deadlines of the system’s response times are derived from functional requirements (see [8]). Our first contribution now lies in the fact, that by decoupling input data capture from processing, in contrast to state-of-the-art automotive static scheduling (cf. Sect. 12.2.1.3), we do not have leave resources idle until the next capturing period is started but can immediately start processing of a fresher frame. By capturing at higher rates than classic automotive designs we thus can minimize the per-application idle time and shorten reaction time by earlier availability of the results from 15 iterations. The scheme is depicted in Fig. 12.5 for a two-application setting executed on a dual core processor.

Fig. 12.5 Decoupling data acquisition from processing allows for better utilization of processing resources at reduced system response time
Sharing of Common Processing Hardware Under Safety Requirements Our second measure for increased efficiency is the definition of sharable clusters of processing resources of a common type. A cluster may consist of multiple CPUs or (programmable) hardware accelerators. We define tasks as being sub-computations within each application pipeline that can be offloaded to the processing resources located within such a cluster of shared resources.

Access to shared resources is restricted: Applications cannot directly issue tasks to such a cluster but only indirectly to a unit we name strategy controller, controlling resource access. Following this scheme, utilization of these resources can be granted in a way, so that application deadlines are fulfilled. Each cluster is equipped with such a strategy controller which in our implementation is a small synthesizable Xilinx MicroBlaze CPU core. The decision on using a programmable CPU was made for flexibility in evaluating different scheduling algorithms. As soon as the scheduling algorithms are decided on, in a series production, one would rather replace that CPU by custom hardware for chip area efficiency. The strategy controller does not only arbitrate requests to shared resources, it also collects status information at runtime from each executed application using a heartbeat scheme:
At each completion of a software pipeline stage, the controller is accordingly notified. Also, individual resources within a cluster notify their controller when a task has been successfully processed. Different scheduling strategies such as EDF can be implemented that consider application deadline requirements over hardware/software boundaries. For an evaluation of scheduling algorithms, please see [8]. A strategy controller can additionally act as safety manager when heartbeat rates of an application drop below a threshold, indicating a malfunction.

12.3.2.2 A Runtime Fail-Operational Mechanism

For individual driving functions, response-time requirements can be derived from functional requirements and parameters such as camera optics or supported vehicle speed range, and sometimes, also from regulations (cf. [8]).

However, giving tight execution time bounds for complex applications such as perception applications is difficult on architectural and application-level. For the former case, this is due to (micro)architectural execution time variations and side-effects found in modern automotive multicore processing platforms (cf. Sect. 12.2.1.2). For example, for the target platform of our case study (see Sect. 12.2.2), microarchitectural timing ambiguity in low-level static analysis is additionally increased on architecture level by shared Level 2 caches and DMA transfers from accelerators. On application-level, specifying bounds for iterative and looped computations is mandatory when using static analysis. However, in perception problems, for example, at the early segmentation pipeline stages, execution time dynamics directly relate to the content of the input video data. This results in extreme overestimations in static analysis when assuming the theoretical worst-case, even if by far not observable during real drives. On the other hand, assuming worst-cases based on statistical data or from observations within test
drives are inherently unsafe, as they give no formal guarantees. For later pipeline
stages limitations might be easier (see [8]).

As confidence on such timing analysis has to be considered low, dynamic analysis
and monitoring is additionally required to guarantee safe behavior on the road.

We propose a runtime monitoring approach coupled with a fail-operational
mechanism: For each application to be concurrently executed we define a set of
deadlines of different criticality where the criticality levels reflect different vehicle
parameterizations (e.g., comfortable braking versus emergency braking). Within
such a deadline, an application’s pipeline is executed multiple times (iterations).
Dividing it by the number of needed iterations that are required to decide on a
reaction leads to an average per-iteration deadline that we use as an indicator of
the current conformity of the real observed execution time versus the static per-
application deadlines. In the case of our traffic light and TSR we require 15 iterations
for deciding on a reaction and define two deadlines of different criticality, the
first deadline allows for a comfortable vehicle reaction (here soft braking), while
a shorter second deadline still allows for a safe reaction but at much less comfort (in
our two-state case, this means parameterizing the vehicle for emergency braking).

Figure 12.6 shows the approach: A slight violation of the per-iteration average
deadline which would allow for a comfortable vehicle reaction is interpreted as
a signal, that an application might not meet its multi-iteration execution time
requirement that would allow for a comfortable reaction. This leads to a state
transition where the vehicle is prepared for a stronger, less comfortable, reaction
that can still be executed within the remaining time period. Additionally, while such
situation should never happen on the road, an overload indication period should
also be implemented, detecting system failure and potentially triggering a limp-
home/-aside mechanism. While we just implement two states. In a real deployment,
more fine-granular vehicle parameterization with more states would be used. This
allows for trading-off guaranteed performance against resource sharing potential, to
provide safe behavior even when observing high system load (cf. Sect. 12.4.2.1).

![Per-application fail-operational state machine: the indication of a potential overload causes a state change, still allowing for an emergency reaction](image)

Fig. 12.6
Additionally, between deadlines of different criticality, a recovery mechanism can be implemented, setting vehicle parameterization to a less critical level when observing deadline conformity for a number $n$ of subsequent iterations.

### 12.3.3 Bridging the Development Gap Using Virtual Prototypes

Virtual prototypes promise to bridge the gap between the low efficiency of classic design processes and the complexity of modern systems. In this section, we present our simulation framework, which has already been applied to automotive ADAS production code and can be applied during most development stages.

An overview on the framework is shown in Fig. 12.7. Software execution on a processor, including a highly accurate timing approximation, is simulated by specialized software simulations. They are integrated with our hardware simulations. We discuss both in the following.

#### 12.3.3.1 Simulation of Software Execution

We support binary code emulation using QEMU [5], which utilizes just-in-time compilation to achieve a high performance and supports a wide range of instruction sets. Furthermore we integrated a custom source level instrumentation tool into our framework. Both are augmented with a custom timing simulation library.

As in most prevailing timing simulations, our library advances time by consuming a certain number of processor cycles when a block of instructions, similar to a basic block, is executed. A novel concept of our approach is utilizing multiple

---

**Fig. 12.7** Overview of our virtual prototyping framework
possible cycle counts for a single block. During simulation, values are selected based on the current context, an abstraction of preceding control flow. For example, for a function called from a loop, our simulation can differentiate the timing of blocks inside the function based on the loop iteration the function was called from.

For an ARM Cortex-M3 processor we have compared our approach to a context-insensitive simulation (i.e., the same cycle count is used on every block execution). Our results [27] demonstrate that even for this relatively simple processor, context-insensitive simulations can induce large errors in the simulated timing, whereas our context-sensitive simulation is highly accurate. More recent [28] results for an ARM Cortex-A9 suggest that our context abstraction is powerful enough to implicitly model caches. This suggests another advantage over context-insensitive simulation, which requires a dynamic cache model. These cache models can be a significant simulation performance bottleneck, slowing down BLS by 200% [42] and SLS by nearly a factor of ten [18]. The context-sensitive simulation causes an overhead of only 70% in comparison to a simulation without timing, which is only 20% more than the 50% [42] for a (extremely inaccurate) context-insensitive simulation that does not include a cache model.

Timing information is stored in the so-called timing database (TDB), which simplifies running multiple simulations (e.g., for different program inputs) with one set of timing data. The timing data itself can be obtained by static analysis or by tracing a single program execution on target hardware.

12.3.3.2 Simulation of Hardware

We build on the SystemC [15] standard for our hardware simulation. Therefore, many academic approaches as well as commercial models can be integrated with our simulations. For our in-house application we rely on the reference implementation of the SystemC kernel.

For software-focused simulations we often implement simplified models of the necessary peripherals, that exclude internal details of peripherals as well as functionality that is not utilized in a project. For hardware-focused simulation, we usually rely on more detailed models and in some cases even integrate RTL simulations when higher accuracy is required.

Transaction level modeling (TLM) [10] is applied to model inter-module communication in order to provide a flexible and reasonable trade-off between simulation accuracy and performance. We usually employ loosely timed synchronization, where modules may simulate without synchronization for a certain amount of time, called a quantum. Further synchronization is enforced on particular interactions with peripherals, for example, when software changes the interrupt mask in an interrupt controller. To accurately handle asynchronous events such as interrupts, a quantum must usually be chosen such that at most one instance of such an event can occur within each simulation quantum.

The software simulations are wrapped by specialized SystemC models to integrate them with the hardware simulation. They expose signals for asynchronous
events (e.g., interrupts) and TLM interfaces for bus accesses. Multiple processors can simply be simulated by instantiating multiple processor modules in SystemC, which are each backed by an independent instance of our software simulation.

### 12.3.4 Bridging the Scalability Gap for Future Platforms

As discussed in Sect. 12.2.3.4, the lack of scalability of current automotive embedded architectures with regards to the growing number and nature of future applications may pose a serious threat for bringing highly or fully automated driving to a mass market soon. For general-purpose computing, the answer to scalability problems has been introducing more parallelism which eventually leads to large scalable multicore or even many-core solutions that contain some to several hundreds of CPU cores. As interconnect, the so-called NoCs are commonly used, connecting a regular structure of CPU tiles over a packet-based on-chip interconnect. Using such a regular structure together with sophisticated routing algorithms allows for large many-cores, where available chips easily contain 48 [19] or 64 cores [44].

Homogeneous many-cores, i.e., only containing one type of processing elements, have been evaluated for automotive perception and recognition problems in research work [6, 48] or [34]. While they all showed that in general complex single automotive functions can be run on such platforms, their absolute results lacked efficiency wrt. performance/watts and performance/#cores (performance per chip area), making them unfavorable for future A.D. functions.

Success of heterogeneous automotive SoCs suggests that instead, heterogeneous multi- to many-core solutions might be better suitable for bridging the scalability gap. An example for such a research architecture is the HeMan (Heterogeneous Many-core) architecture developed at FZI (see Fig. 12.8). It is a tiled multi- to many-core research platform that incorporates heterogeneous tiles each containing classic embedded cores with individual RAM and ROM memories but also allows for domain-specific (potentially programmable) hardware accelerators. Each tile has a special local memory used for message passing and providing data to the hardware accelerators. External memory is accessed by special tile types.

Specific measures in per-router message queues enforce message routing that can give latency guarantees for end-to-end message transfers between tiles based on providing virtual communication channels on top of the physical links.

HeMan employs privileged communication channels for system control messages such as heartbeats (see Sect. 12.3.2) where specific hardware measures prevent communication from non-privileged tiles on those channels. On top of these virtual communication channels, prioritization of messages is used. Such measures also provide the means for securing system-relevant communication, such as the boot process where the program code for application tiles is deployed by a system controller whereas for regular tiles, (accidentally) overwriting the program code of other tiles is forbidden. As the NoC is a shared resource, a guaranteed throughput and latency for individual communication can only be provided when access to this
Fig. 12.8 Scalable heterogeneous automotive processing architecture: Processing tiles are connected via an NOC. Hardware accelerators provide required performance—while the interconnect allows for scaling towards future demands. Off-chip I/O and DRAM access is provided by special tile types.

shared resource remains restricted: Thus the conformity of a tile to a previously assigned data plan (communication schedule) is controlled in hardware by specific traffic shaping components within each tile’s network interface.

Hardware accelerators within tiles share a common interface as proposed in Sect. 12.3.1. The runtime approach from Sect. 12.3.2 can also be applied to such a tiled architecture: Then, tiles are grouped into a cluster that is augmented by one strategy controller. Also, redundancy can be applied such as adding lockstep cores for the CPU cores within each tile that execute the same instruction stream as the original core with a online self-tested comparator unit detecting mismatches that might occur because of hardware faults (dual-mode redundancy). In such a case, NoC messages to a safety management tile can be triggered automatically.

This could especially be useful regarding the demanded further integration of more functionality per chip [22], where safety-relevant tiles, e.g., data fusion, planning or decision making, are executed on specifically hardened processing tiles, while functions which are not safety relevant are executed on generic processing tiles. On top, safety concepts on higher levels such as soft lockstep can be implemented where multiple tiles perform redundant calculations on the same input data whereas other tiles take the role of comparators. Also, hybrid approaches are possible: Here, the results from hardware accelerators are regularly checked by a software thread that runs on the tile’s CPU during idle periods (self-test). For all these mechanisms, mixed system topologies are possible where only safety-relevant tiles provide redundancy, while processing-optimized application tiles are use the soft approaches.
12.4 Modern Platform-Based Automotive System Design

We now present our tooling framework, that shows how our individual solutions for realizing embedded A.D. functions from Sect. 12.3 can be efficiently combined. Additionally, we evaluate the discussed runtime concepts of Sect. 12.3.2 and demonstrate the accuracy of our timing simulation of Sect. 12.3.3 on an automotive function.

12.4.1 Tooling Framework

Creating and developing a complex heterogeneous target platform which integrates the concepts of Sect. 12.3 requires an efficient workflow, which can only be achieved with powerful tooling. Our tooling and workflow to achieve this goal is depicted in Fig. 12.9 and essentially consists of three steps: generating intermediate sources from a platform description (Platform Generation), creation of the actual simulation and FPGA platforms (Platform Creation), and finally, execution of the platforms (Platform Execution). These steps are performed iteratively during development.

Generating both, a virtual and an FPGA-based prototype, automatically from the very same description, enable a highly efficient development. We use the virtual

![Diagram](image_url)

Fig. 12.9 Our tooling framework: simulation binaries and RTL code are generated from the same specification. The approach of Sect. 12.3.3 is used to simulate timing within tiles
prototype for early design space exploration beyond the chip area limitations of a physical FPGA and the time-consuming design of hardware and its debugging.

Furthermore, the virtual prototype is an ideal platform for efficient development of application software and, due to the integrated timing simulation, allows a continuous verification of timing requirements.

The hardware prototype allows extraction of non-functional properties, such as timing and power information which can be back-annotated to the virtual prototype. Additionally, we use the physical prototype for low-level hardware development and optimization, which is not possible on the virtual prototype due to the high level of abstraction, for example, to develop low-level interfaces to external components, such as CMOS sensors. Those results are then integrated into our RTL code template library making them available for automated platform generation.

### 12.4.1.1 Platform Generation

Platform Generation starts with a common platform description in XML. This description covers the topology of the mesh, together with parameters such as the desired number of virtual channels and the number, positions, and tile types for each individual tile. From a templated library of SystemC, Verilog, and VHDL source files, a custom tool called *Topology Generator* creates intermediate inputs for both, the virtual prototype and the hardware prototype. This includes the creation of the specified interconnect between tiles and the mesh. Two sets of source files are produced, SystemC sources and the RTL hardware descriptions: For SystemC, wrapper files integrate the simulations of individual CPU cores within tiles and links to our custom library that provides timed SystemC/QEMU co-simulation. Also, NoC code is generated, that provides the local links of each individual tile to the mesh. Finally, a simulation top-level acts as a simulation testbench.

RTL code includes the requested mesh topology in Verilog RTL code which is used for both, simulation and hardware platform, in the subsequent Platform Creation step. The RTL code for individual tiles, including the CPU core instantiations, is generated in VHDL, supported by templates from the library.

### 12.4.1.2 Platform Creation

Based on the intermediate results generated in the previous step, the actual platform creation can be performed. For the virtual prototype, first, the topology-dependent Verilog NoC description is translated into a cycle-accurate, yet fast SystemC model using the Verilator [45] tool. Together with top-level testbench Verilog code, this is compiled and linked into an executable simulation binary. For the FPGA prototype, the individual tiles, given in VHDL, and the mesh, given in Verilog, is synthesized, placed and routed using the vendor-specific synthesis toolchain.
12.4.1.3 Platform Execution

With the simulation and the hardware platform available, one can now utilize both: The FPGA prototype is executed by uploading first the bitstream, and then a software binary that includes the software of each individual tile. Our hardware implementation can fit up to nine tiles on our FPGA development board.

Providing target software binaries, the virtual prototype can be executed. Stimuli such as input data can be provided via dedicated testbench models. The SystemC simulation does not only provide a dramatic speedup compared to RTL (RTL running at approx. 100 simulated clock cycles per wall-clock seconds), but at the same time it also increases observability, as we can use target debugger instances to connect to each simulated processor and arbitrarily trace signals of the mesh. The simulation allows for systematic and fast software development as well as design space exploration, where topologies not fitting the FPGA can be explored.

If timing simulation is required, the co-simulation library reads context-sensitive timing information from a previously generated Timing Database that can either be extracted from recorded instruction traces of an execution on the physical prototype or from static analysis (cf. Sect. 12.3.3).

12.4.2 Evaluation

We now evaluate our dynamic resource management from Sect. 12.3.2 including our fail-operational strategy from Sect. 12.3.2.2 and our timing simulation from Sect. 12.3.3.

12.4.2.1 Runtime Resource Management

Our approach on runtime resource management has been evaluated on a Xilinx Zynq SoC, using it’s two ARM Cortex-A9 cores and a hardware accelerator cluster of four SVM RBF kernels. However, it is retargetable to other architectures. The two applications from Sect. 12.3.2 each get assigned to one Cortex-A9 core for the software parts of their pipeline. As strategy controller, we use an embedded Xilinx MicroBlaze soft core. Signaling of heartbeats, requests for shared resources, and completion of accelerator executions to the corresponding strategy controller is done using interrupts to the strategy controller. We implemented different scheduling strategies (cf. [8]). For evaluation we use a combination of static allocation together with earliest deadline first where each application is statically assigned one accelerator exclusively while sharing the remaining ones using an EDF schedule. This provides a good trade-off between throughput and sharing potential. EDF always schedules the runnable tasks of an application that has least slack towards its deadline. For doing so, the already consumed time budget of an application is calculated from the heartbeat signals issued on each completion of a pipeline stage.
Our FPGA-based evaluation system either can be used with a real CMOS sensor, where image preprocessing such as demosaicing is done in hardware, or as in-lab prototype where videos from test drives are provided using UDP. In the latter case, evaluation was performed subtracting the extra overhead introduced. Figure 12.10 shows the evaluation platform.

Our decoupling of processing from input data acquisition shows vast improvements compared to a static schedule where the time-slot period is set to the maximum observable execution time of a pipeline iteration during test drives. For the TSR, relative improvements range from a min./avg./max. of 37, 39, and 42 %, respectively. The traffic light application still shows minimum 8 % improvement with 36 % avg. and 49 % max. improvements. Total system power is below 6 Watts, including DRAM, HDMI, and GigE.

**Runtime Fail-Operational Mechanism** We evaluate our runtime fail-operational mechanism from Sect. 12.3.2 using the platform of Fig. 12.10. Heartbeat signals are used to notify completion of a pipeline stage in order to monitor the per-iteration execution time of each individual application, steering the per-application state machine. Dynamics in the TSR is much less as in the TLR. Driving from an outer area into the city, we could successfully demonstrate our approach using a ten-iteration soft-recovery, once switching to an emergency parameterization and, as expected, never reaching failure state (Fig. 12.11).
12.4.2.2 Platform Timing Simulation

As a preliminary evaluation of the timing simulation, we conducted a case study for the TSR application [28]. We simulate the execution of the TSR on an ARM Cortex-A9 using the framework presented in Sect. 12.3.3. The necessary timing database was generated by tracing the execution on real hardware. As simulation input we utilize several real pictures of traffic situations and 630 distinct combinations of the segmentation parameters.

The input used during hardware tracing influences the timing of the traced execution, which in turn influences the results of a simulation based on a TDB generated from that traced execution. To investigate this factor we generated several TDBs which were generated using default segmentation parameters and all but one image. One image was excluded, as it does not include any circle detected by the segmentation and a resulting TDB would thus not cover the classification code.

We use each TDB with all simulation inputs and evaluate the results to represent the following practical scenarios: (1) Arb: An execution for an arbitrary image is traced and the resulting TDB is used for all simulations. (2) Del: An execution for an image with all supported traffic signs is traced and the resulting TDB is used for all simulations. (3) PerI: Executions for all images are traced and each TDB is only used in simulations for the same image that was used in tracing. For an image with no signs, a visually similar image is used. Figure 12.12 shows our results. Even the Arb strategy produces results that allow for a practical application. Therefore in practice only straightforward factors need to be considered when applying our trace-based approach. For example, the TSR should be traced using an input image that includes a traffic sign. Putting more effort into tracing input selection can lead to further accuracy improvements.
12.5 Conclusions

In this chapter, we presented solutions that address the deficiencies in state-of-the-art development and realization of embedded automotive driving functions that pose a barrier towards automated driving. We believe that our contributions will bring significant improvements not only to individual challenges but especially when combined, where they might eventually lead to more efficient architectures for highly or fully automated driving, accelerating their introduction to a mass market.

Acknowledgements This work was partially funded by the State of Baden-Württemberg, Germany, Ministry of Science, Research and Arts within the scope of Cooperative Research Training Group EAES, and by the ITEA2/BMBF project MACH under grant 01IS13016B.

References

1. AbsInt Angewandte Informatik GmbH: aiT Worst-Case Execution Time Analyzers (2015), http://www.absint.com/ait/. Accessed 18 March 2015
2. ADAS Applications Processor TDA2x System-on-Chip Technical Brief (2013), http://www.ti.com/lit/ds/symlink/tda2.pdf. Accessed 18 March 2015
3. S. Altmeyer, B. Lisper, C. Maiza, J. Reineke, C. Rochange, WCET and mixed-criticality: what does confidence in WCET estimations depend upon? in OASIcs-OpenAccess Series in Informatics, vol. 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
4. AUTOSAR, Specification of Timing Extensions, Release 4.2.2, Document 411 (2015). doi:411. http://www.autosar.org. Accessed 20 Oct 2015
5. F. Bellard, QEMU, a fast and portable dynamic translator, in USENIX Annual Technical Conference, FREENIX Track (2005)
6. J. Borrmann, A. Viehl, O. Bringmann, W. Rosenstiel, Parallel video-based traffic sign recognition on the intel SCC many-core platform, in Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP) (2012), pp. 1–2
7. J. Borrmann, F. Haxel, D. Nienhüser, A. Viehl, J. Zöllner, O. Bringmann, W. Rosenstiel, STELLaR - a case-study on SysTEmaticalLy embedding a traffic light recognition, in Proceedings of the 17th IEEE International Conference on Intelligent Transportation Systems (ITSC) (2014), pp. 1258–1265. doi:10.1109/ITSC.2014.6957860
8. J. Borrmann, F. Hasel, A. Viehl, O. Bringmann, W. Rosenstiel, Safe and efficient runtime resource management in heterogeneous systems for automated driving, in *Proceedings of the 18th IEEE International Conference on Intelligent Transportation Systems (ITSC)* (2015).

9. O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneider, P. Sasidharan, S. Singh, The next generation of virtual prototyping: ultra-fast yet accurate simulation of HW/SW systems, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (EDA Consortium, Grenoble, 2015), pp. 1698–1707.

10. L. Cai, D. Gajski, Transaction level modeling: an overview, in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (ACM, New York, 2003), pp. 19–24.

11. W. Ecker, V. Ensen, R. Schwencker, T. Steininger, M. Velten, TLM+ modeling of embedded hw/sw systems, in *Proceedings of the Conference on Design, Automation and Test in Europe* (European Design and Automation Association, Dresden, 2010), pp. 75–80.

12. ETAS GmbH, ASCET-SynTAS End-to-end Timing Analysis for Electronic Control Systems (2015), [http://www.etas.com/download-center-files/products_ASCET_Software_Products/ASCET_SYMTAS_flyer_en.pdf](http://www.etas.com/download-center-files/products_ASCET_Software_Products/ASCET_SYMTAS_flyer_en.pdf). Accessed 01 June 2016.

13. M. Geilen, S. Stuijk, Worst-case performance analysis of synchronous dataflow scenarios, in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (ACM, New York, 2010), pp. 125–134.

14. R. Henia, L. Rioux, N. Sordon, G.E. Garcia, M. Panunzio, Integrating model-based formal timing analysis in the industrial development process of satellite on-board software, in *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (2014), pp. 619–625.

15. IEEE Standard for Standard SystemC Language Reference Manual (2012). IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005).

16. R. Kumar, V. Zyuban, D.M. Tullsen, Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling, in *Proceedings of the 32nd International Symposium on Computer Architecture, 2005. ISCA’05* (IEEE, New York, 2005), pp. 408–419.

17. K. Lu, D. Muller-Gritschneider, U. Schlichtmann, Hierarchical control flow matching for source-level simulation of embedded software, in *Proceedings of the 2012 International Symposium on System on Chip (SoC)* (IEEE, New York, 2012), pp. 1–5.

18. K. Lu, D. Muller-Gritschneider, U. Schlichtmann, Fast cache simulation for host-compiled simulation of embedded software, in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013 (IEEE, New York, 2013), pp. 637–642.

19. T.G. Mattson, M. Riepen, T. Lehnerg, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, The 48-core SCC processor: the programmer’s view, in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2010, pp. 1–11. [http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5644880](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5644880).

20. MIRA Ltd, MISRA-C:2004 Guidelines for the use of the C language in critical systems (2004). [www.misra.org.uk](http://www.misra.org.uk).

21. Mobileye (2015), [mobileye.com](http://mobileye.com). Accessed 18 March 2015.

22. A. Monot, N. Navet, B. Bavoux, F. Simonot-Lion, Multisource software on multicore automotive ECUs - combining runnable sequencing with task scheduling. IEEE Trans. Ind. Electron. 59(10), 3934–3942 (2012).

23. Moving Closer to Automated Driving, Mobileye Unveils EyeQ4 System-on-Chip with its First Design Win for 2018 (2015), [http://www.mobileye.com/blog/press-room/moving-closer-automated-driving-mobileye-unveils-eyeq4-system-chip-first-design-win-2018/](http://www.mobileye.com/blog/press-room/moving-closer-automated-driving-mobileye-unveils-eyeq4-system-chip-first-design-win-2018/). Accessed 18 March 2015.

24. D. Nienhüser, Kontextsensitive Erkennung und Interpretation fahrrelevanter statischer Verkehrselemente. Ph.D. thesis, KIT Karlsruhe (2014).

25. nVidia, Tegra K1 Technical reference manual (2014), [https://developer.nvidia.com/tegra-k1-technical-reference-manual](https://developer.nvidia.com/tegra-k1-technical-reference-manual). Accessed 18 March 2015.
26. nVidia, Tegra X1 Super Chip (2015), http://www.nvidia.com/object/tegra-x1-processor.html. Accessed 18 March 2015
27. S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, O. Bringmann, Context-sensitive timing simulation of binary embedded software, in Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2014)
28. S. Ottlik, J.M. Borrmann, S. Asbach, A. Viehl, W. Rosenstiel, O. Bringmann, Trace-based context-sensitive timing simulation considering execution path variations, in 21st Asia and South Pacific Design Automation Conference (ASP-DAC) (2016)
29. M. Pressler, A. Viehl, O. Bringmann, W. Rosenstiel, Execution cost estimation for software deployment in component-based embedded systems, in Proceedings of the 17th international ACM Sigsoft Symposium on Component-Based Software Engineering (ACM, New York, 2014), pp. 123–128
30. S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2. edn. Prentice Hall Series in Artificial Intelligence (Prentice Hall, Upper Saddle River, NJ, 2003)
31. Safer SoCs for safer driving. SemiWiki.com, the Open Forum for Semiconductor Professionals (2014), https://www.semiwiki.com/forum/content/3844-safer-socs-safer-driving.html. Accessed 18 March 2015
32. D. Sanchez, G. Michelogiannakis, C. Kozyrakis, An analysis of on-chip interconnection networks for large-scale chip multiprocessors. ACM Trans. Archit. Code Optim. (TACO) 7(1), 4 (2010)
33. K. Schmidt, J. Harnisch, D. Marx, A. Mayer, A. Kohn, R. Deml, Timing analysis and tracing concepts for ECU development. Technical Report, SAE Technical Paper, 2014
34. T. Schonwald, A. Koch, B. Ranft, A. Viehl, O. Bringmann, W. Rosenstiel, Stereo depth map computation on a Tilera TILEPro64 embedded multicore processor, in Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP) (2012)
35. S. Stattelmann, Source-Level Performance Estimation of Compiler-Optimized Embedded Software Considering Complex Program Transformations (Verlag Dr. Hut, Munich, 2013)
36. S. Stattelmann, O. Bringmann, W. Rosenstiel, Fast and accurate source-level simulation of software timing considering complex code optimizations, in Proceedings of the 48th Design Automation Conference (DAC) (2011)
37. G. Stein, E. Rushinek, G. Hayun, A. Shashua, A computer vision system on a chip: a case study from the automotive domain, in IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Workshops, vol. 3 (2005), p. 130. doi: 10.1109/CVPR.2005.387. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1565445
38. STMicro and Mobileye are Developing Third-Generation SoCs for Driver Assistance. John Day’s Automotive Electronics - Insight for Engineers (2011), http://johndayautomotiveelectronics.com/stmicro-and-mobileye-are-developing-third-generation-socs-for-driver-assistance/. Accessed 18 March 2015
39. Syntavision GmbH: Leading in real time - overview flyer (2015), https://www.syntavision.com/downloads/Flyer/Syntavision_Overview_Flyer_2015_SEPT.pdf. Accessed 20 Oct 2015
40. TDA2x SoC for Advanced Driver Assistance Systems (ADAS) (2015), http://www.ti.com/tda2x. Accessed 18 March 2015
41. TDA3 SoC Processor Advanced Driver Assistance Systems (ADAS) (2015), http://www.ti.com/product/tda3. Accessed 18 March 2015
42. D. Thach, Y. Tamiya, S. Kuwamura, A. Ike, Fast cycle estimation methodology for instruction-level emulator, in Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE) (2012)
43. The MathWorks, Inc. (2015), http://www.mathworks.com. Accessed 18 March 2015
44. Tilera goes pro with TILEPro64 (2008), http://www.tgdaily.com/business-and-law-features/39408-tilera-goes-pro-with-tilepro64. Accessed 18 March 2015
45. Veripool Verilator (2015), http://www.veripool.org/. Accessed 18 March 2015
46. N. Weste, D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 4th edn. (Addison-Wesley Publishing Company, New York, 2010)
47. XILINX: UG585 - Zynq-7000 all programmable SoC technical reference manual (2014), http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Accessed 18 March 2015

48. J. Zimmermann, Applikationsspezifische Analyse und Optimierung der Energieeffizienz eingebetteter Hardware/Software-Systeme. Ph.D. thesis, Universität Tübingen, Germany (2013). Website http://nbn-resolving.de/urn:nbn:de:bsz:21-opus-73952