The Best of Both: High-performance and Deterministic Real-Time Executive by Application-Specific Multi-Core SoCs

Steffen Vaas*, Peter Ulbrich†, Marc Reichenbach* and Dietmar Fey*
Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg, Germany
*Chair of Computer Science 3 - Computer Architecture
†Chair of Computer Science 4 - Distributed Systems and Operating Systems
{steffen.vaas, peter.ulbrich, marc.reichenbach, dietmar.fey}@fau.de

Abstract*—Embedded multi-core processors improve performance significantly and are desirable in many application-fields. This in particular includes safety-critical real-time systems, which typically require a deterministic temporal behavior. However, even tasks without dependencies running on different cores can interfere due to, sometimes hidden, shared hardware resources, such as common memories or buses. Consequently, only a pessimistic assumption of the *worst-case execution time* (WCET) that incorporates interference can be given. Hence, the aspired performance gain fizzles out in the poor temporal analyzability. Based on the fact that in safety-critical systems all tasks and their dependencies are known at compile-time, this paper presents an approach to generate application-specific, deterministic multi-core processor architectures for these systems. Thereby safety-critical tasks are executed on dedicated *Deterministic Execution Units* (DEUs) including lightweight, deterministic processor cores, bus systems, memories and peripherals. The remaining soft real-time tasks are executed on a general purpose multi-core processor that offers performance over determinism. Consequently, timing analysis for hard real-time tasks is significantly simplified, since interferences caused by shared resources and scheduling are effectively eliminated. To show the benefits of our approach, an application-specific architecture for a flight controller was generated and compared to an *ARM Cortex-A9* dual-core as reference. Overall, we were able to significantly improve temporal properties of safety-critical tasks while preserving the overall performance for soft real-time tasks.

***Index Terms*—Safety-Critical, Reconfigurable, Reliable, Deterministic, Distributed, NoC, SoC**

I. Introduction

A characterizing feature of real-time systems is that they have to react to physical events within given deadlines. These can be more or less strict, for which reason real-time systems are typically classified into hard and soft real-time (i.e., high and low criticality). In a car, for example, the airbag always has to meet hard timing constraints, whereas missed deadlines in the navigation system are ultimately just annoying.

Therefore, a fundamental requirement in real-time system engineering is the sound estimation of *worst-case execution times* (WCETs) for safety-critical, hard real-time tasks. However, contemporary processors typically pack a whole series of performance-enhancing measures, such as pipelines, out-of-order execution, and tiered memory architectures jeopardizing this aim. To give an example, a simple addition may take between 3 (best) and 321 cycles (worst case) in a 2001 PowerPC 755 [1]. Consequently, any sound WCET analysis will be pessimistic to a certain extent and provide only over-approximations (upper bounds) of the actual WCET. In the past, real-time applications of different criticality (i.e., hard and soft) where either treated equally according to the higher criticality or partitioned on dedicated execution platforms. For example, in a car each function is rendered by a single *electronic control unit* (ECU), which made it relatively easy to isolate and satisfy timing and safety requirements. Recently two important factors have changed: On the one hand, autonomous driving is just around the corner, making software much more complex as well as more parts safety-critical. On the other hand, multi-core embedded processors have hit the market, supplying an amount of processing power unthinkable in the past. Market pressure forces the industry to seize this opportunity by consolidating ECUs. Therefore, software of lower safety requirements now co-resides with applications of high safety demands, thus increasing interference, complexity and the certification hurdle occur at the same time.

Although considerable progress has been made in WCET and especially hardware analysis for single core real-time

This work is partially supported by the German Research Foundation (DFG) under grants no. SCHR 603/9-2 and the Bavarian Ministry of State for Economics under grant no. 0704/883 25 (EU EFRE funds).

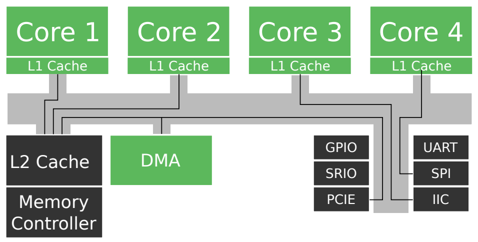


Fig. 1. The structure of an exemplary multi-core processor architecture: tasks running on different cores share hardware resources, as the common memory controller, which leads to contention. Even if tasks do not access to same peripherals, interferences occur caused by the common bus system.

systems, multi and many-core systems still represent an unresolved challenge to be tackled. Here, the major show-stoppers are interferences caused by inter-task dependencies as well as shared resources and, even worse, hidden yet inherent interdependencies within the actual hardware architecture, as illustrated in Fig. 1. Overall, these massively increased interferences typically bloat analysis estimates to the point where they become unfeasible in practice.

Consequently, the real-time community pursues various other ways to deal with the problem: with partitioning, novel scheduling techniques, and deterministic hardware being the most prominent representatives. The first, partitioning, certainly is the most intuitive and widespread approach. It is, for example, state-of-the-art in the automotive domain, where individual functions are tied to dedicated cores. This works well, as long as the problem fits the hardware granularity. Another approach is to pigeonhole the cumbersome WCET overestimates by resorting to mixed-criticality scheduling. Here, optimistic WCET estimates are used as long as budgets are not exceeded. Otherwise, the system gradually switches to hard upper bounds for critical tasks with hard deadlines by omitting less critical tasks to mobilize resources. Although being a captive concept it requires multiple WCET estimates per task, which can be tricky to acquire, and restricts the verifiability in practice due to the added scheduling complexity. Finally, deterministic architectures, such as PATMOS or RISC-V, can be used to again simplify the WCET analysis. However, this is also accompanied by a substantial loss in performance due to the absence of the aforementioned hardware techniques.

A. Problem Statement

Recent developments in *field programmable gate arrays* (FPGAs) and *systems on a chip* (SoCs) allow for a further approach: Instead of relying on *commercial off-the-shelf* (COTS) hardware, the architecture can be specifically tailored to meet the application's needs while maintaining determinism and analyzability. In earlier work [2], we have therefore investigated the automated partitioning and tailoring of automotive real-time systems on dedicated, deterministic soft-cores within contemporary FPGAs. However, this approach comes at the cost of flexibility and performance as the entire hardware architecture is synthesized within the FPGA. As mentioned earlier, real-time systems are heterogeneous in practice, with performance and safety requirements varying between individual tasks. We assume that the predominant part of real-time systems is of low criticality (e.g., comfort and diagnost functionality). Moreover, tasks typically exhibit substantial dependencies and precedence constraints, for example between sensing, computing and actuating. Consequently, synthesis of dedicated soft cores for individual tasks quickly reaches its limits and is not generally applicable.

To leverage FPGAs in heterogeneous settings a combination of performance-oriented general-purpose cores and task-exclusive real-time cores is missing that allows for interdependencies and precedence constraints between critical and non-critical tasks and, at the same time, prevents harmful interference by strong isolation concepts.

B. Our Contribution

In this paper, we present the *reconfigurable, reliable, deterministic, distributed* (R2-D2) toolchain, an approach for the automated generation of application-specific multi-core architectures for heterogeneous real-time systems. We claim the following key contributions to improve:

- Improved *scalability* by an automated mapping of soft and hard real-time tasks to COTS processors and *Deterministic Execution Units* (DEUs), respectively.
- Enhanced *performance* due to an optimized utilization of the available FPGA resources.
- Strong *isolation* between deterministic and non-deterministic execution units by tailored data exchange and synchronization mechanisms.

The remainder of this paper is structured as follows: First, we detail our R2-D2 approach, the automated architecture generation and shared resources in Sec. II. Subsequently, we exemplify our approach by a flight control showcase in Sec. III and IV before discussing related work in Sec. V and concluding the paper in Sec. VI.

II. The R2-D2 Approach

This section introduces our R2-D2 approach: starting with basic design principles, we subsequently detail our approach to automated architecture generation. Finally, we tackle the challenging issue of unavoidable intra-task dependencies and shared resources.

A. The Basic Approach

COTS multi-core architectures typically include shared resources such as common memories or buses. Multiple tasks may attempt to access them at the same time, which results in interference between these tasks. To avoid this issue from the first, we propose an application-specific processor architecture with dedicated, spatially separated hardware resources for safety-critical tasks. The detailed structure of an exemplary architecture generated by the R2-D2 toolchain is illustrated in Fig. 2. Every safety-critical task is exclusively executed by one DEU that includes all required components to run as an independent basic microcontroller system. The corresponding CPU core is customizable, different architectures from 8-bit to 64-bit can be selected. Additionally, some architectures are parametrizable, thus for example floating point units or hardware dividers can be optionally instantiated. The core is connected as sole master on one AXI bus to the corresponding slave peripherals. By default a timer for periodical execution of code, a watchdog as fail-safe option, a controller for *inter-processor communication* (IPC) and a memory are instantiated. The abolition of a common memory for all processor cores furthermore eliminates a well-known bottleneck of COTS multi-core systems. Moreover, the distributed, spatially separated memories of safety-critical tasks increase the reliability of the system. Corrupt write accesses from uncritical tasks are inhibited by design. Thus, a *memory protection or management unit* (MPU/MMU) can be omitted. Since only one task is executed on one DEU, no additional scheduling mechanisms are required. As a consequence, an operating system is not necessary any longer. Tasks can run on bare-metal. This reduces the complexity of the system drastically, because no preemption of tasks with higher priority can occur. Only the source code itself and data dependencies between tasks have to be regarded for timing analysis of the safety-critical tasks. Therefore, tasks running on dedicated DEUs have the same behaviour as they would be executed on bare-metal single-core systems. For such single-core systems there exist a range of scientific [3] and commercial [4] static timing analysis methods, which can now be applied to this application-specific multi-core system. Hence, the verification of the system can be simplified significantly.

However, implementing a complete DEU for every task is inconvertible for large systems with a considerable amount of tasks due to limited chip area. Since for uncritical tasks within a system deterministic behaviour is not necessary, all non-safety-critical tasks can be executed on a COTS multi-core processor architecture, which is only loosely coupled with the DEUs for data transfers. Empiric studies show [5] that only a few tasks within an application system are safety-relevant, so only a low number of dedicated DEUs will be required.

B. Automated Architecture Generation

In most cases safety-critical applications consist solely of static software code, so all tasks within a system are already known at compile time. To design such systems as a top-down approach and to decouple dependencies of hardware and software developments, system specifications as *AUTOSAR* for automotive applications and *ARINC 653* in the domain of avionics are applied. Typical information, which are stored in such system descriptions are number, priorities, dependencies, shared variables, stimulus and allocations of tasks. Usually system specifications have a similar structure, in which all properties and dependencies are stored in an XML file. To receive a better compatibility, the open source system description format of *APP4MC* [6] was taken as input format to generate the processor architecture.

For an automated application-specific hardware generation the parameters of the application system have to be extracted from the corresponding *APP4MC* description. The detailed stages of the toolchain are illustrated in Fig. 3.

1) APP4MC Hardware Description: In the first step, parameters of the software system stored in the *APP4MC* description are extracted. Therefrom, additional parameters configuring the hardware architecture and the allocation of tasks were derived and added to the system specification. For each safety-critical task a complete DEU gets instantiated and the task gets exclusively allocated to it. The architecture of every DEU can be optionally configured. Various processor cores are available and further peripherals can be added. Uncritical tasks get allocated to the COTS multi-core processor and will be scheduled by a *real-time operating system* (RTOS). Currently the toolchain is limited to *Xilinx Zynq* and *Zynq UltraScale+* devices, low-priority tasks can therefore be scheduled on an *ARM Cortex-A9* dual-core, an *ARM Cortex-R5* or *ARM Cortex-A53* quad-core.

2) Hardware Generation with Vivado TCL: After the *APP4MC* system description is updated with hardware-specific parameters, a set of TCL commands for *Xilinx Vivado* can be derived from the XML system description. Therefrom, an instance of this application-specific architecture gets implemented in *Vivado*.

At first, all DEUs defined in the *APP4MC* were generated successively. Each one includes a customized processor core and default peripherals, as well as further ones that were custom-added in the system description. The related peripherals are connected by a dedicated AXI bus system with the processor core as only bus master, as shown in Fig. 2. At this stage, the DEUs are completely isolated, which results in a deterministic structure, so every DEU can be regarded and analyzed independently. However, data exchange between tasks running on the DEU is mandatory, so in the following step, the communication links were implemented.

The R2-D2 toolchain supports two communication types: *shared variables* providing common access to specific values and *message queues*, which include a FIFO storage for buffered data transfers. Moreover, data transfers between non-critical tasks executed on a general-purpose multi-core and data transfers with safety-critical tasks have to be distinguished. Uncritical tasks running on the COTS multi-core processor can simply share data by the common memory. Message queues are realized in software by an RTOS. For safety-critical tasks running on an DEU, a common memory between all cores in the system is not an option. This would result in a bottleneck when accessing shared data and reduce the performance. Furthermore, it would destroy the gained reliability by the spatially separated execution of tasks. As all data transfers between tasks are already defined in the system description, dedicated communication links can be implemented in hardware. Shared variables are realized as dedicated hardware registers. To transfer data, one task writes values in the dedicated register. The DEUs of all tasks requiring read access to this variable are physically connected to the corresponding register. This principle is realized in the IPC peripheral of the system, as illustrated in Fig. 2. Message queues are realized by hardware FIFO blocks, which can either be mapped to RAM blocks or registers in the FPGA, depending on the buffer size of the message queues. Moreover, message queues own two AXI slave ports, which are directly connected to the bus system of the corresponding DEUs. Thus, exclusive access for the involved tasks is provided.

After the automated implementation of the architecture instance in *Vivado*, further post configurations can be done, as pinout and timing constraints. In a final step, the hardware bitstream is generated.

3) Task Frames with Eclipse TCL: As the decentralized, irregular structured multi-core architecture instance owns separated DEUs with different processor cores, different memory maps and no common memory, each safety-critical task has to be treated as separated software project. Every task executed on a corresponding DEU requires a dedicated linker script and has to be compiled with a customized compiler toolchain. However, one software project per task causes poor usability. To simplify the programmability of the system, a TCL script for the *Eclipse* IDE from *Xilinx* is derived from the *APP4MC* system description. Thereby one software project is generated for every DEU. Each one includes a basic bare-metal application and the software drivers for all connected peripherals. For the general-purpose multi-core processor an RTOS project with raw task frames is generated. To keep all source files in a common folder structure, the code sections for the different tasks running on various DEUs have to be delimited by *define* symbols. Thus, the different compiler toolchains from various projects generate binaries for all DEUs from a common source-code base. Furthermore, all shared variables have globally the same addresses, so the usage of the shared variables behaves the same as on a single project.

C. Shared Hardware Resources

The basic principle of the presented approach is to avoid shared hardware resources by an application-specific multi-core architecture to prevent interferences between tasks. Shared hardware resources, as common SPI controllers of multiple cores, can be resolved by instantiating DEUs explicitly for managing shared peripherals. Thus, no locking mechanisms are necessary. Tasks requiring access to this common resource can transfer data via message queues to the DEU maintaining the peripheral. However, if legacy designs shall be ported or there is not enough space on the FPGA device for additional DEUs, it is also possible to connect a shared hardware peripheral to multiple DEUs. To keep the spatial separation of the DEUs, an additional bus in a lower hierarchy level is implemented, as shown in Fig. 4. This bus has only one slave port, which is connected to the peripheral, and several master ports. The master ports are connected to the bus systems of the DEUs. Hence, tasks only influence each other during read/write accesses on the shared peripheral.

To control shared peripheral devices, further locking mechanisms are required. Therefore one semaphore unit is implemented in hardware for each shared resource. This hardware component has one AXI slave port for each DEU, that requires access to the shared peripheral. To send a lock request to the semaphore, tasks can set a flag in the control register of the semaphore unit. Since only one task is executed on a DEU, no scheduling takes place and the status register can simply be polled until the task obtains access to the critical section. The slave ports of the semaphore own static priorities to guarantee a correct real-time behaviour.

III. Quadcopter Controller Showcase

To show the advantages of the R2-D2 toolchain practically on an use case, an architecture of a quadcopter flight controller was generated and implemented on a *Xilinx Zynq 7020* device. The structure is shown in Fig. 5. The flight controller combines multiple functionalities: data acquisition from sensors, position estimation in space and motor control. For more complex flight operations the *Mission Control* task can navigate the quadcopter by GPS coordinates. The *MAVLink* task provides an interface over the identically named widespread UAV communication protocol to configure settings, as PID parameters. Additionally an image processing unit is implemented that scans the ground for landing marks and automatically activates a landing sequence.

The tasks of the system are partitioned in real-time critical and non-critical sections, as it is proposed in [7]. Main critical section of the flight controller is the control loop consisting of sensor-, position- and motor control. Therefore these three tasks are executed on isolated DEUs. Data between the *Sensor Control*, *IMU* and *PID Control* tasks are exchanged by shared variables, so all calculations are executed with the latest sensor data available, which is common practice to avoid synchronization locks. Since sensors and tasks deliver continuous values, deviations caused by delayed data updates are negligible. The tasks *Mission Control* and *MAVLink* have no hard deadline requirements, but they are classified as safety-critical, as their outputs directly influence the control loop. Therefore, those tasks are also executed on DEUs. The non-critical part of the system includes the detection of the landing mark, since it is an optional feature and the landing sequence can also be started manually. Therefore the related tasks were executed on the *ARM Cortex-A9* dual-core. Furthermore, this image processing application is comparably compute intensive, so it benefits from the performance of the firm implemented dual-core. Due to the combination of multi-core and FPGA on one chip, the landing application can be additionally improved by a hardware accelerator for the *Sobel* algorithm, which processes the edge detection of an image. The feature detection is described in [8]. Detected landing sections are signaled to the mission control task, which will start the landing sequence.

The deterministic structure of the generated architecture simplifies the later verification of safety-critical sections with static timing analysis tools. The control loop can be considered separately without influences of an RTOS or non-critical tasks, which were spatially separated. Uncritical parts can further be developed completely separated, since the DEUs executing the safety-critical tasks do not have the same code base.

IV. Results

To show the performance values of the architecture generated in III, the execution times of the 5 critical and 25 further non-critical tasks were measured. In Fig. 6 the minimum, maximum and average execution times of all tasks are shown. Task 1 to 5 have to fulfill a deadline of 1 ms. The other uncritical tasks were executed with a periodical repetition of $1\ ms * Task\ \#$. To compare the results, two reference systems have been implemented. In the first one all tasks were implemented on the *ARM Cortex-A9* dual-core. Tasks with odd ID are allocated on core 1, the others on core 2. On the second reference system all tasks were executed on DEUs.

TABLE I
REQUIRED SPACE OF VARIOUS PROCESSOR ARCHITECTURES ON THE *Xilinx Zynq-7020* FPGA DEVICE. AN ADDITIONAL LIMITATION IS THE INTERNAL MEMORY OF 560 KB, WHICH HAS TO BE SEGMENTED FOR ALL DEUS.

Architecture	**Bits**	**LUTs**	**Registers**	**DSPs**	**MHz**
Microblaze min.	32	1893 (3.6%)	1660 (1.6%)	0 (0%)	148.0
Microblaze max.	32	3543 (6.7%)	3018 (2.8%)	6 (2.7%)	138.9
Picoblaze	8	189 (0.4%)	145 (0.1%)	0 (0%)	123.1
NEO430	16	833 (1.6%)	567 (0.5%)	0 (0%)	62.2
Rocket RISC-V	64	29640 (53.8%)	13484 (12.7%)	24 (10.9%)	28.6
Custom *MIPS I*	32	2918 (5.5%)	2373 (2.2%)	0 (0%)	91.2

During the measurements all deadlines were met. Safety-critical tasks executed on DEUs have a larger, but deterministic execution time with less jitter, which increases the quality of the control loop. Moreover, the jitter of the uncritical tasks running on the *Cortex-A9* could be reduced by outsourcing the tasks with higher priority to DEUs. However, for non-critical tasks the execution on DEUs is not convenient. The execution time gets increased, which is mainly caused by the lower clock frequency of soft-core processors. Moreover, the selected device did not provide enough programmable logic resources to implemented one DEU for every task. It was only possible to implement 13 DEUs with a *Microblaze* core in extended configuration including a floating-point unit. Nontheless, the number of DEUs strongly depends on the selected architecture of the core. Therefore, the required resources for the different architectures are shown in Table I.

Furthermore, this approach profits from the fact that real-time critical tasks require rather low latencies than computing power and because of the high repetition rates they only need a few of KB memory for instructions. This favours the use of small soft-core processors, which have only a low performance in comparison to firm implemented cores. However the overhead of an RTOS for scheduling is not required, so low latencies can be realized. The instruction code is small enough to fit into the internal memory of the chip, so the dedicated memory can be realized.

V. RELATED WORK

There are various classes of approaches determining the timing behaviour of real-time systems [9], as simulation techniques, software analysis and deterministic hardware approaches. All of them take shared hardware units into account. Those completely different approaches have pros and cons.

Simulation techniques, as shown in [10], estimate the WCET by a large number of simulation runs. Different parameter values at the beginning of each simulation trigger interferences between tasks executed in parallel. Simulators deliver results in short time, but the systems are too complex to simulate all existing states. This is not relevant for non-safety-critical real-time applications, but for safety-critical applications, which have to be certified, approaches employing simulation techniques cannot be applied. Simulation results rely on statistical assumptions for the absence of failures, which is not an acceptable argument for authorities [11].

A further approach is the static timing analysis of applications at system level. All possible interferences between tasks caused by shared hardware resources have to be considered. The advantage here is that the determined WCET does not rely on statistics. However, the resulting WCET has pessimistic assumptions. All possible interferences are considered, even if most of them might never occur concurrently. The detection of interferences, which cannot occur at the same time, takes a lot of effort. Thus, plenty approaches consider only specific problems within real-time systems, as caches [12], contention of bus resources [13], optimized allocation of tasks to cores [14] or scheduling techniques [15]. Main problem always remains a too pessimistic WCET. Hence, there are publications with hybrid approaches trying to combine measured or simulated results with static timing analysis [16].

Further approaches use deterministic processor architectures. A comparison of various approaches is discussed in [2]. Often those architectures provide only low performance or create new issues, as complex data transfers between cores with low communication bandwidth by a regular network-on-chip (NoC) structure. Therefore the presented approach combines architectural parts of general-purpose and deterministic multi-core processors to gain deterministic behaviour for safety-critical tasks and more performance for uncritical code sections.

VI. CONCLUSION

In this paper we presented an approach to execute safety-critical and uncritical tasks in parallel on one chip by generating an application-specific multi-core architecture. The safety-critical tasks of a demonstration system were isolated and executed on dedicated, deterministic hardware units. Hence, timing analysis for those tasks get simplified significantly, as hardware interferences were avoided by design. Further measurements have shown that the selected FPGA device does not provide enough resources to implement all tasks on DEUs, so a combination of COTS multi-cores and DEUs is required. Thereby the jitter of the tasks was reduced, as well as the average execution time of the uncritical tasks executed on the COTS multi-core.

REFERENCES

[1] C. Ferdinand *et al.*, “Towards model-driven development of hard real-time systems,” in *Model-Driven Development of Reliable Automotive Services*. Springer, 2008, pp. 145–160.

[2] S. Vaas *et al.*, “The R2-D2 toolchain – Automated porting of safety-critical applications to FPGAs,” in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.

[3] P. Montag and S. Altmeyer, “Precise WCET calculation in highly variant real-time systems,” in *2011 Design, Automation Test in Europe*, pp. 1–6.

[4] “AbsInt aiT.” [Online]. Available: https://www.absint.com/ait/

[5] S. Fürst, “BMW_20151005.pdf.” [Online]. Available: http://image.mdstec.com/mail/SES/BMW_20151005.pdf

[6] “APP4MC.” [Online]. Available: http://www.eclipse.org/app4mc/

[7] S. Schreiner *et al.*, “Autonomous Flight Control Meets Custom Payload Processing: A Mixed-Critical Avionics Architecture Approach for Civilian UAVs,” in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 348–357.

[8] B. Pfundt *et al.*, “Teaching heterogeneous computer architectures using smart camera systems,” in *2016 11th European Workshop on Microelectronics Education (EWME)*.

[9] G. Fernandez *et al.*, “Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art,” in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 31–42. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4602

[10] P. K. Valsan, H. Yun, and F. Farshchi, “Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[11] “Position Paper CAST-32A,” certification Authorities Software Team (CAST). [Online]. Available: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf

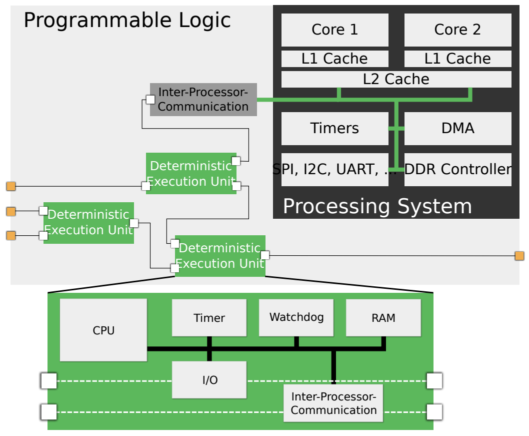
[12] N. Guan *et al.*, “FIFO cache analysis for WCET estimation: A quantitative approach,” in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 296–301.

[13] D. Dasari and V. Nelis, “An Analysis of the Impact of Bus Contention on the WCET in Multicores,” in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*.

[14] M. Panić *et al.*, “RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores,” in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[15] T. Kelter, H. Borghorst, and P. Marwedel, “WCET-aware scheduling optimizations for multi-core real-time systems,” in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 67–74.

[16] J. Nowotsch *et al.*, “Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement,” in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 109–118.


Programmable Logic
Core 1
Core 2
L1 Cache
L1 Cache
L2 Cache
Inter-Processor-Communication
Timers
DMA
SPI, I2C, UART, ...
DDR Controller
Processing System
Deterministic Execution Unit
Deterministic Execution Unit
Deterministic Execution Unit
CPU
Timer
Watchdog
RAM
I/O
Inter-Processor-Communication

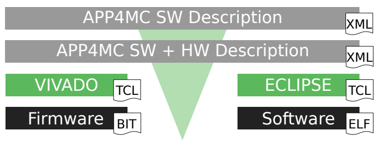
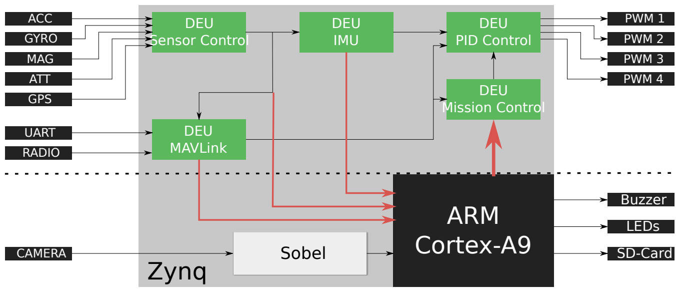


Fig. 3. The tool-flow for the automated hardware generation. An *APP4MC* system description is required as input and will be extended by additional parameters defining the application-specific hardware. Then TCL instructions for *Xilinx Vivado* and *Eclipse* are created to generate a hardware architecture instance and raw task frames for the application.



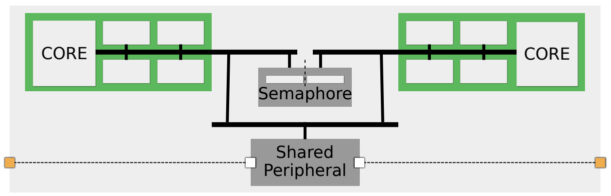


Fig. 4. To support shared peripherals and to avoid a common bus system between DEUs, shared hardware resources are connected by a bus system with a lower hierarchy. Thus, the DEUs stay spatially separated. For correct execution of applications, locking mechanisms are required. Therefore, a semaphore unit is implemented in hardware.

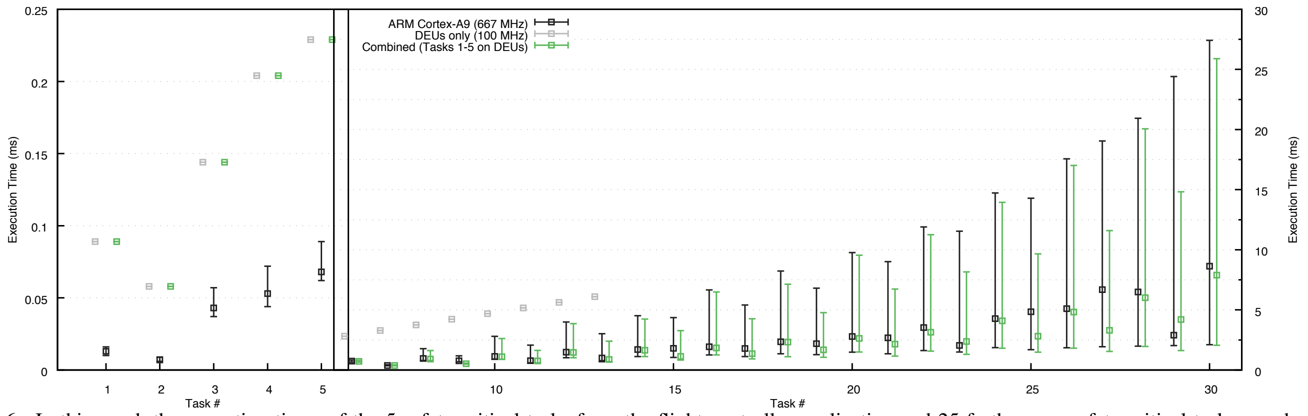


Fig. 6. In this graph the execution times of the 5 safety-critical tasks from the flight controller application and 25 further non-safety critical tasks are shown. The DEUs have a highly deterministic behaviour, but the FPGA device does not provide enough space for 30 of them. By applying the COTS dual-core only, non-deterministic behaviour is not excluded. A combination of both architectures unifies the advantages of them and further reduces the jitter of all tasks.