Using Trace Data for Run-Time Optimization of Parallel Execution in Real-Time Multi-Core Systems

Florian Schade, Timo Sandmann, Jürgen Becker
Institut fuer Technik der Informationsverarbeitung (ITIV)
Karlsruhe Institute of Technology
Karlsruhe, Germany
{schade, sandmann, becker}@kit.edu

Henrik Theiling
SYSGO GmbH
Klein-Winternheim, Germany
hth@sysgo.com

***Abstract*—In recent years, multi-core processors are becoming more and more common in embedded systems, offering higher performance than single-core processors and thereby enabling both computationally intensive embedded applications as well as the space-, weight-, and energy-efficient integration of software components. However, real-time applications, for which meeting certain deadlines must be guaranteed, do not profit as much from this transition. This is mainly due to interference between the processing cores of commercial-off-the-shelf multi-core processors at shared resources, hampering the predictability of task execution times. An effective approach to avoid this is running the critical tasks exclusively on one core while pausing execution on all other cores. This, however, reduces the overall system efficiency since parallel execution potential remains unused. In this work we present a novel approach to managing shared and exclusive execution in such systems. By on-line observation of the critical task progress via the on-chip trace infrastructure, we reduce the time of exclusive execution when it is safely possible and thereby increase the overall system efficiency. Using trace information allows for early detection of parallelization potential and does not require modifications to the critical application, which helps avoiding re-certification of the critical application. We present an implementation on a heterogeneous multi-processor system-on-chip using a state-of-the-art hypervisor for critical systems and evaluate its performance. Our results indicate that a performance gain of 37 % to 41 % over approaches focused on exclusive execution can be reached in low-interference situations.**

***Index Terms*—Multicore processing, Real-time systems, Concurrency control, Timing isolation, Worst-case execution time**

I. Introduction

The amount and complexity of functionality implemented in embedded systems such as automotive E/E architectures and avionics control systems has increased rapidly over the recent years. At the same time, a trend of consolidating embedded processing units has emerged, leading to the integration of formerly distributed functionality in few powerful processing platforms to help reduce space, weight, and power consumption. This is enabled by the availability of powerful multi-core processors and heterogeneous Systems-on-Chips (SoC). With this trend, however, the issue of integrating functionality of different criticality became apparent. In mixed-criticality systems, where low- and high-criticality functionality is implemented on the same processing platform, sufficient isolation needs to be put in place to guarantee freedom from interference between these components as demanded by safety standards. In the resulting software architectures, this isolation is often achieved by embedded hypervisors, which control software access to platform resources in a certifiable manner. Here, relevant resources include memory, peripherals, and CPU cores.

In the avionics domain, the ARINC 653 standard [1] defines a real-time operating system (RTOS) interface to facilitate the Integrated Modular Avionics (IMA) concept, targeting this integration of functionality on a single execution hardware. Its concepts are supported by a variety of industrial hypervisors such as PikeOS [2] or XtratuM [3]. These hypervisors use TDMA scheduling approaches to ensure temporal isolation of software in different partitions. A periodic sequence of fixed-length time windows are defined for each core, to which partitions are mapped for execution.

In embedded systems found in the automotive, avionics, or industrial domain, real-time control functionality is mainly implemented in the form of periodic tasks. In each period, they process input data and output their results either by controlling actuators or forwarding information to other tasks. For critical functionality, it needs to be ensured that every processing iteration finishes before a defined deadline. During system design this means that sufficient processing resources need to be reserved for the task. For non-real-time tasks this is not necessary since late or missing task results are acceptable and do not negatively affect the system in an intolerable way.

To ensure that critical tasks finish in time, their worst-case execution time (WCET) needs to be determined. While this is considered a solved problem for single-core processors, it poses a major challenge in multi-core systems [4]. This is due to interference between applications running on parallel cores, caused by contending access to shared resources. One prominent example is the memory infrastructure, including shared caches and memory controllers. Due to high cache miss penalties, shared cache line evictions caused by one application can cause significant delays in applications running on a parallel core. Since the delay caused by interference on a resource heavily depends on the behavior and timing of the contending applications, detailed knowledge on the applications and their execution would be necessary to calculate a tightly-bound WCET for parallel applications running on multi-core processors. Therefore, in practice, critical tasks are often scheduled exclusively on multi-core processors by disabling parallel cores for a time window matching the WCET of the critical task and thereby avoiding interference.

In this work, we present a novel approach to scheduling mixed-criticality systems on hypervisor-managed multi-core processors with TDMA scheduling. The concept increases the parallel execution of critical tasks and non-critical best-effort tasks by monitoring the critical task progress at run-time using CPU trace data, switching to exclusive execution only if meeting the deadline is at risk. It requires no prior knowledge on the non-critical tasks and critical tasks do not need to be modified.

II. Related Work

The integration of mixed-criticality functionality in multi-core processors has been subject to intensive research [5], [6]. Interference between cores, caused by contending access to shared resources, poses major challenges in the development of real-time and safety-critical systems. In [6], Maiza et al. summarize that the memory bus is considered the main source of inter-core interference. Since interference can lead to significant slowdown in task execution [7], a multitude of approaches have since been proposed, ranging from interference-aware execution models over tracking and bounding shared resource access to hardware measures to reduce interference.

Considering interference bounding in hypervisor-partitioned multi-core systems, Nowotsch et al. [8] present a mechanism that limits shared resource access by guest applications and evaluate it on the P4080 multi-core processor. The hypervisor is extended by a monitor keeping track of each process's accesses to the shared system network-on-chip. When a process exceeds its budget, it is suspended until the budget is replenished.

Other approaches focus on temporarily suspending task execution on parallel cores to facilitate the completion of critical tasks without interference. Lara et al. [9] propose the use of a dedicated hardware module to detect the start and end of the critical task execution. Aligned to its deadline, it ensures that a time window equal to the single-core WCET of the critical task is available for exclusive execution. This is realized by disabling other cores' access to the memory bus. When the critical task finishes before its WCET, it switches back to shared operation.

Freitag et al. [10] propose a more detailed monitoring and control approach considering the critical task progress at run-time. This progress is determined by a pattern matching approach on performance counter signal curves, called *Fingerprinting*. By comparing signal samples to pre-recorded model samples, the slowdown of a critical task is determined. A hardware module external to the multi-core processor implements the monitoring functionality and throttles execution on the other cores as necessary. Several approaches for throttling are proposed and compared concerning their performance. While the idea presented in [10] is related to our work, we propose a different way of tracking the critical task progress.

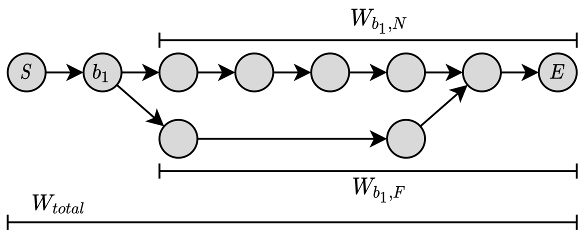

Fig. 1. Exemplary control flow of a critical task

We investigate the use of trace data generated by the on-chip tracing subsystem, which allows for direct detection of decisions taken at conditional branch instructions. This yields information on the critical task progress and its current path within its control flow, which can be exploited for obtaining information on the remaining execution time and thereby for early parallel execution.

III. Interference Control using Trace Data

A. Systems under Consideration

In this work we consider embedded software systems where tasks of different real-time criticality are executed on a single multi-core processor as found in commercial-of-the-shelf (COTS) SoCs. We assume that there are periodic *critical tasks* which implement real-time functionality. These tasks are characterized by their period interval p, their start time t_s at which they start their execution, and deadline t_d before which they need to complete in each period. We assume that critical tasks can be analyzed to obtain a WCET for single-core execution on the target processor while all other cores are idle (*exclusive execution*). Furthermore, as depicted in Figure 1, we assume that there are multiple walks across a critical task's control flow graph (CFG), starting from the first instruction executed in an iteration (S) to the last instruction in the final vertex E. We assume that among these walks some differ significantly in execution time. Therefore, a set B of conditional branch instructions with significant influence on the overall execution time of the task can be identified. Each conditional branch instruction $b \in B$ leads to two execution alternatives (*decisions*), for which the partial WCETs $W_{b,F}$ and $W_{b,N}$ can be determined. They equal the remaining worst-case execution time when starting from the first instruction following b, assuming that the conditional branch was followed (F) or not followed (N).

While we assume that there is only one critical task scheduled at the same time on the processor, there can be any number of non-critical *best-effort* tasks scheduled in parallel on the other cores. Best-effort tasks are not subject to real-time requirements and may be interrupted at any time without affecting critical system functionality. Besides this, we assume that there is no prior knowledge on these tasks.

B. Goal and System Overview

Based on the assumptions described in Section III-A, the presented approach aims to increase the overall system performance by maximizing the use of parallelism in the multi-core processor. We assume that in the average case maximizing the time available for the parallel execution of tasks will lead to a higher system performance since more computational work can be done. Therefore, we attempt to maximize the time during which critical and best-effort tasks run in parallel. At the same time, the approach must guarantee that critical tasks meet their deadlines in every iteration.

Figure 2 shows the main components of the proposed system. Critical tasks and best-effort tasks are running on an application multi-core processor managed by a hypervisor. The hypervisor enforces resource access constraints to provide isolation between the tasks. With respect to scheduling, it implements a TDMA-based cyclic schedule. External to the application processor runs an *execution controller*. It is synchronized with the hypervisor scheduler and controls best-effort task execution on the hypervisor while the critical task is running. To do so, it determines the progress of the critical task by extracting information on branch decisions from trace data emitted by the application processor. It keeps track of the remaining worst-case execution time of the critical task and ensures that slowdown introduced by the best-effort tasks does not lead to deadline misses.

C. Scheduling Approach

The proposed scheduling approach requires a detailed analysis of the critical task. Besides the overall WCET W_{total}, conditional branch instructions which significantly influence the task execution time need to be identified for monitoring. As indicated in Section III-A, for each of the identified branch instructions $b \in B$, the partial WCETs $W_{b,F}$ and $W_{b,N}$ have to be determined and configured in the execution monitor.

When configuring the hypervisor schedule, the critical task is assigned a time window $W = \{c, s, D\}$ on one CPU core. It is characterized by the CPU core c, the offset s within the hyperperiod, i.e., the scheduling frame, and its duration D. The time window is aligned with the critical task offset and deadline so that $t_s = s$ and $t_d = s + D$. To guarantee completion of the critical task, the time window duration D needs to be chosen so that $D \geq W_{total}$. We suggest that it is extended by D_{slack} to allow for some slowdown due to interference. This results in $D = W_{total} + D_{slack}$.

At run-time, the hypervisor enforces the schedule as configured. At the start of the critical task time window in iteration i, at $t_{s,i}$, the critical task is scheduled on its assigned CPU core.

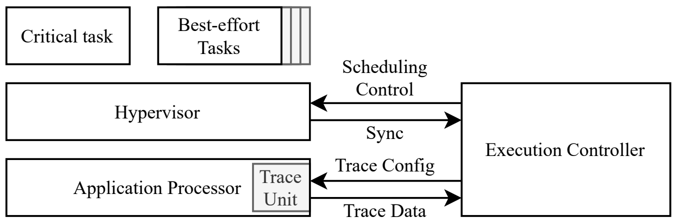

Fig. 2. System component overview

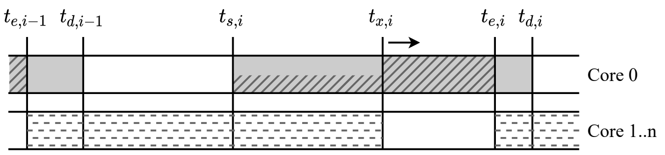

Fig. 3. Scheduling approach. The critical time window is highlighted in gray, critical task execution is indicated by stripes, best-effort task execution by dashes.

Best-effort tasks are executed on the remaining cores. This is referred to as *parallel execution*. A synchronization event is communicated to the execution controller, which then starts its control functionality and tracks the elapsed time as well as the critical task progress. It calculates the moment $t_{x,i}$ at which the system will have to switch to exclusive execution:

$$t_{x,i} \leftarrow t_{d,i} - W_{total} \tag{1}$$

Reserving a timespan that equals the task's WCET before its deadline for exclusive execution ensures that the critical task will finish in time.

As the critical task proceeds, the execution monitor continuously monitors the incoming trace data. When a conditional branch instruction $b_j \in B$ is passed, trace data indicates the branch decision $d_{j,i} \in \{F, N\}$, i.e., which path was taken. This information is used to update $t_{x,i}$ based on the remaining WCET for the path taken:

$$t_{x,i} \leftarrow t_{d,i} - W_{b_j,d_{j,i}} \tag{2}$$

When $t_{x,i}$ is reached, the execution monitor triggers the hypervisor to switch to exclusive execution mode. The critical task is still monitored so that it is possible to switch back to shared mode in case that $t_{x,i}$ is postponed back to the future.

When the execution monitor detects that the critical application has finished, it also switches the system to shared execution, allowing best-effort applications to run for the remaining part of the time window.

Figure 3 visualizes the proposed approach: At $t_{s,i}$, the critical application execution starts. However, due to interference, it is slowed down (indicated by half striped filling). At $t_{x,i}$, the system switches to exclusive execution, preventing interference and allowing the critical task to run at single-core performance. When observed branch decisions lead to a reduction of the remaining WCET, $t_{x,i}$ is postponed. When the critical task ends at $t_{e,i}$, best-effort tasks are enabled again.

D. Rationale

The delay caused by interfering tasks running in parallel largely depends on the task behavior. Typically, memory-intensive tasks can be expected to cause more interference than CPU-intensive tasks. Thus, best-effort applications shall be enabled to run in parallel as long as their effect on the critical task does not endanger meeting the deadline. Since no prior knowledge on the best-effort task is available, its effects on the critical task need to be determined by measurement at run-time.

Using branch information for progress monitoring allows for the early detection of changes in the critical task's remaining run-time. This is in contrast to conservative approaches which enforce exclusive execution for (1) the whole duration of the critical task execution or for (2) a duration matching its WCET. When a branch decision at runtime leads to a reduction of the execution time by Δt, in conservative approaches of type (1), this will lead to a gain in best-effort execution time of Δt, since the best-effort tasks will be scheduled as soon as the critical task finishes. For approaches of type (2), this will not influence best-effort execution time at all.

In the proposed concept, the early detection of a reduction in critical task remaining (worst-case) execution time is used to allow for early parallel execution. Early parallel execution is further supported by defining an appropriate D_{slack} since this prevents the system from switching to exclusive operation at the beginning of the critical time window. In case of low interference between the critical and best-effort tasks, parallel execution of a best-effort task will barely increase the execution time of the critical task. At the same time, it allows for best-effort workload to be processed. With early parallel execution, this allows for achieving a gain in best-effort execution time that is greater than Δt and thereby enhancing the overall system performance.

In an optimal case, branch monitoring is laid out in a way that leads to frequent postponing of $t_{x,i}$ so that for low-interference scenarios $t_{x,i}$ always lies in the future, leaving the system in shared execution mode.

The proposed approach postpones $t_{x,i}$ only after critical task progress was observed. Thereby, it is ensured that a sufficient phase of exclusive execution can be realized before the critical task deadline. Even in cases of high interference this ensures that the deadline will be met. Furthermore, the trace subsystem is not required to fulfill real-time requirements in order to give critical task real-time guarantees, since trace system latencies will only delay postponing $t_{x,i}$.

IV. Implementation on COTS Hardware

A. System overview

The concept was implemented on the ZCU102 Zynq UltraScale+ MPSoC evaluation board [11]. The SoC includes an Application Processing Unit (APU) comprising 4 Cortex-A53 processing cores, a Real-time Processing Unit (RPU) including a dual-core Cortex-R5 processor, as well as programmable logic, among other components. It features an ARM CoreSight tracing subsystem which supports instruction tracing on the processing cores. With regards to the memory infrastructure, the APU provides separate 32 kB L1 caches for instructions and data for each core as well as a 1 MB shared L2 cache with 64 B line length. The RPU comprises 128 kB of tightly coupled memory which can be used for predictable instruction execution and load/store timing. Components in the SoC are connected via AXI interconnects. A DDR controller interfaces 4 GB DDR4 RAM located on the evaluation board.

An overview on the implementation is given in Figure 4. The APU is used as application processor, hosting the PikeOS

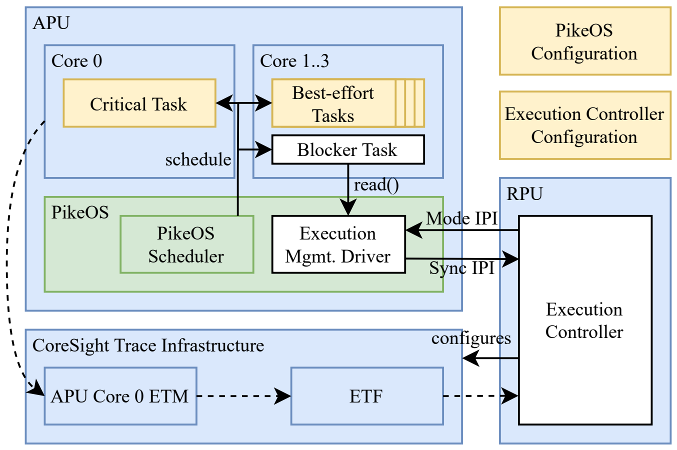

Fig. 4. Implementation of the trace-data-based execution control concept. SoC components are highlighted in blue, commercially available software in green, concept-specific implementations in white and application-defined artifacts in orange. Trace data is indicated using dashed lines.

embedded hypervisor. Critical and best-effort tasks are implemented in separate PikeOS partitions which are mapped to a cyclic TDMA schedule. The critical task is mapped to core 0, best-effort tasks can be be mapped to the other cores. The PikeOS kernel is extended by an execution management driver which can inhibit application processing on cores 1 through 3 using blocker tasks, thereby realizing exclusive execution. The execution controller is implemented as a bare-metal application running on the RPU. It communicates to PikeOS via inter-processor interrupts (IPI) and interfaces the platform's trace subsystem. The trace subsystem is configured to forward trace data generated by the APU core 0 Embedded Trace Macrocell (ETM) into an Embedded Trace FIFO (ETF) for buffering, which is polled periodically by the execution monitor.

B. PikeOS

PikeOS is a hypervisor and operating system for embedded real-time systems that is certifiable up to the highest safety levels. It enforces separation of applications and guest OSs both in the space and the time domain, using a concept of partitions. In the PikeOS configuration, tasks are assigned to resource partitions. Resource partitions are logical containers, for which access limitations to system resources such as memory and peripherals can be configured and are then enforced by the hypervisor's separation kernel. With regard to CPU scheduling, the concept of time partitions and time windows is used. A time window represents a period of time during which only threads of resource partitions which are mapped to the time window are allowed to be executed. A sequence of time windows for each core forms a scheduling frame, which is repeated periodically by the scheduler.

In our implementation the critical task is mapped to one resource partition (*critical partition*), while best-effort tasks are mapped to one or multiple other partitions to facilitate isolation. The critical partition is mapped to a time window on core 0 (*critical time window*) while the other partitions are mapped to other cores. Time windows are defined so that initially the best-effort partitions are active in parallel to the critical partition. This configures the hypervisor for parallel execution.

To realize exclusive execution we make use of the prioritization mechanism in the PikeOS scheduler. To prevent best-effort task execution, a blocker thread is started on each core. It is assigned a higher priority than the maximum priority configured for the best-effort resource partition threads. Thereby, the blocker thread can be used to control other thread's execution on the core by switching between running and blocking thread state. When the blocker thread runs, it prevents other threads from running on the core. When it blocks, lower-priority threads are scheduled. While running, the blocker thread spins on a wait-for-interrupt instruction in user space to put the CPU in low-power mode and not cause interference itself.

To prevent best-effort tasks from influencing the blocker thread, blocker threads are placed in a separate resource partition. This resource partition is mapped to time partition 0, which is intended for service tasks and always eligible to run.

In the PikeOS kernel space, an execution management driver module controls the execution of all blocker threads. It provides a `read()` function that uses the PikeOS kernel service API to put the calling thread into blocked mode. When an inter-processor interrupt (IPI) is received, the driver identifies whether shared or exclusive execution is requested and wakes the corresponding blocker threads as necessary.

C. Execution Monitoring and Control Application

The execution controller application running on the RPU implements the trace-data-based critical task progress monitoring, derives $t_{x,i}$ and notifies the execution management driver when a mode change is necessary.

Its configuration comprises the task deadline, its total WCET, final code block instruction address, and trace-specific information such as the trace context ID. Monitored branch instructions are stored in a tree structure, each element comprising a branch instruction address filter, $W_{b,N}$, $W_{b,F}$, as well as a list of branches to monitor once a branch is passed.

When configuring the execution control application, care must be taken that the processing latency of the implemented control system is considered when it comes to switching to exclusive execution. This delay includes IPI latencies for synchronization and mode change notification as well as latencies for timer interrupt handling and issuing of the mode change notification. In addition, the latency for IPI handling and blocker thread activation in PikeOS needs to be considered as well as slowdown in critical task execution caused by cache line evictions before the switch. These delays can be deducted from the configured deadline or added to the branch-specific remaining WCET values to ensure a sufficiently early switchover to exclusive operation.

The execution controller synchronizes with the PikeOS scheduler at the beginning of each critical time window and then starts a hardware timer as a local time reference. As

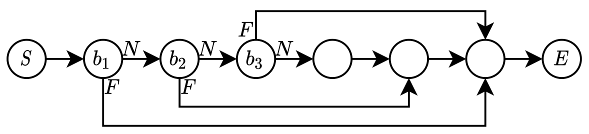

Fig. 5. Evaluation task structure. Circles indicate CFG blocks, b_i identifies the branch instruction located at the end of the CFG block.

described in Section III-C, it then calculates $t_{x,i}$ and programs a timer interrupt. When this interrupt occurs, it notifies the PikeOS kernel module by IPI to switch to exclusive execution. After synchronization, the application configures the trace infrastructure to emit trace data when relevant conditional branch instructions are processed. Here, instruction address filter modules provided by CoreSight Embedded Trace Macrocells are used to limit trace generation to relevant instruction address ranges, which reduces the trace data processing load. By polling the ETF buffer, trace data is then received and parsed. When relevant branch decision information is received, a new $t_{x,i}$ is calculated, the timer interrupt is updated, and the trace infrastructure filters reconfigured as necessary. When appropriate, the execution mode switch is triggered. In parallel, the trace hardware is configured to indicate when the final code block is reached, indicating the end of the critical task. When this is detected, a mode switch back to shared execution is triggered.

V. Evaluation

In this section, we present a first evaluation of our concept. We investigate the performance of the proposed scheduling approach for various interference scenarios and measure the gain in system performance over conservative approaches. For this evaluation we consider system performance to be the processing rate of best-effort workload while the critical task completes in all iterations before its deadline.

The evaluation setup is based on the implementation described in Section IV. Two application tasks are present in the system: A critical task is executed periodically during the critical time window in the hypervisor schedule. It contains several conditional branch instructions which represent decisions made on input data (cf. Figure 5), resulting in four possible execution paths through the application. Between these branch instructions, it executes a pre-defined number of iterations of a worker function. Next to it, a best-effort task runs continuously whenever it is scheduled by the hypervisor. It also loops on a worker function that processes a pre-defined amount of workload. One iteration of the worker function is referred to as a *work package*. In addition, the task keeps track of the number of work packages processed.

In order to simulate different interference situations and account for CPU- and memory-intensive applications, multiple worker function implementations were created. To simulate low-interference processing, a CPU-intensive worker function was created, mainly executing arithmetic operations by calculating the square root of random numbers. A memory-intensive

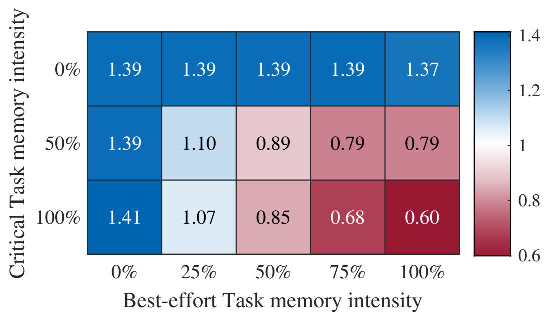

Fig. 6. System performance gain based on memory intensity of the critical and best-effort tasks

worker function writes and reads a sequential memory range of $1\,\mathrm{MB}$ with a $64\,\mathrm{B}$ stride. Based on preliminary experiments on the hardware platform, this function causes significant interference and thereby slowdown when executed on multiple cores in parallel. This becomes clear considering that the parameters match the L2 cache size and line length. To simulate mixed workloads, a worker function implementation was created that randomly executes single work packages of the CPU- and memory-intensive worker function. One work package was tuned to take approx. $1\,\mathrm{ms}$ of execution time during exclusive operation.

Based on these tasks and worker function implementations, different system configurations were evaluated, which are characterized by different *memory intensity* of the tasks. For each critical task configuration, the overall WCET as well as partial WCETs were determined. This was done experimentally on the target platform by measuring the maximum execution time of 500 iterations in exclusive execution mode, determining the maximum (partial) run-time and adding a margin of $5\,\%$. Based on this, the hypervisor schedule was defined. The critical time window was defined on core 0 and its duration set to W_{total}. For the low-interference critical task ($0\,\%$ memory intensity), this led to duration of $140\,\mathrm{ms}$, while the other implementations led to $141\,\mathrm{ms}$. In parallel, on core 1, a best-effort time window of the same duration was defined, while the other cores were not used.

As a reference, we implemented a more conservative scheduling approach where the critical task is executed exclusively for its complete run-time (cf. type (1) described in Section III-D). This is achieved by assigning both the critical and best-effort task to the same time partition on core 0 while setting the critical task to a higher priority. Thereby, as soon as the critical task finishes, the best-effort task is executed for the rest of the time window.

Table I shows the number of best-effort work packages processed over 80 schedule iterations (20 per path) for the presented setup. The resulting gain factors with respect to the reference implementation are plotted in Figure 6. The measurements indicate that the proposed approach leads to a significant gain in system performance as long as the memory intensity of either one of the contending applications is low. As long as one task is purely CPU-intensive, a gain of $37\,\%$ to $41\,\%$ could be observed. With increasing memory intensity of both tasks, this gain decreases. At $50\,\%$ memory workload in the critical task and $25\,\%$ in the best-effort task, the gain is still $10\,\%$. However, as soon as both the critical task and best-effort task exceed $50\,\%$ memory-intensive workload, the proposed approach shows a negative gain, leading to a performance loss of up to $40\,\%$ at $100\,\%$ memory workload.

This loss in performance is to be expected since we employed worker tasks that maximize L2 cache usage in their memory-intensive phase. When such tasks contend for memory resources, this leads to a high number of L2 cache misses, resulting in RAM accesses. Since RAM access latencies exceed L2 cache access latencies by a large factor, this leads to an extensive slowdown of the task execution. Typically, this can even lead to parallel execution times exceeding sequential execution of the tasks. Yet, for all measured scenarios, the proposed approach and the priority-based approach achieved meeting the deadline in all measurements.

VI. Discussion

Our evaluation shows that the proposed approach successfully increases parallel execution in the system compared to a conservative approach. The resulting effect on the overall system efficiency therefore heavily depends on the nature of the applications running in parallel. As long as little interference is caused between the tasks, a significant gain in processing performance can be expected. This is the case when either the critical task or the best-effort tasks mostly access core-local resources. When both tasks make intensive use of limited shared resources such as shared caches or RAM, significant delays can be observed. While not explicitly evaluated in this work, this may also be the case when contending for peripherals and other shared system components. To ensure that best-effort tasks cannot cause critical task deadline misses by locking system components beyond the control of the hypervisor, system designers need to make sure that access to such resources is limited appropriately. In cases of heavy interference, scheduling approaches realizing exclusive execution for the run-time of the critical tasks can lead to better performance due to sequential execution of the high-interference workload. This is a well-known phenomenon inherent to parallel execution of such tasks on COTS multi-core processors. As identified in [7], the execution time of tasks under heavy interference can be multiple times slower compared to their execution time when running exclusively. In that case, sequential execution leads to an earlier completion of the two tasks. One possible approach to reduce this interference is the application of cache management techniques such as cache coloring.

At maximum interference, the proposed approach still ensures that the critical task deadline is met. While priority-based approaches lead to significantly higher best-effort task performance in this case, our results indicate that the proposed approach still achieves approx. $44\,\%$ of the workload that would be achieved at minimum interference, that is, for a CPU-intensive critical task. In comparison, the safe approach of completely disabling other CPU cores for the full duration

TABLE I
BEST-EFFORT WORK PACKAGES PROCESSED OVER 80 SCHEDULE ITERATIONS BASED ON CRITICAL TASK (CT) AND BEST-EFFORT TASK (BT) MEMORY INTENSITY

BT memory intensity	Proposed approach			Priority-based scheduling		
	CT 0 %	CT 50 %	CT 100 %	CT 0 %	CT 50 %	CT 100 %
0 %	7361	7522	7739	5294	5417	5475
25 %	7549	6025	5873	5430	5462	5501
50 %	7749	5008	4793	5576	5601	5647
75 %	7996	4547	3967	5734	5764	5819
100 %	6319	3669	2805	4622	4644	4677

of the critical time window, i.e., critical task WCET, would result in no best-effort task progress during that time window.

While the presented implementation manages a single critical task, the concept can be applied to any number of critical tasks as long as they are scheduled sequentially. Parallel execution of critical tasks is not supported, since the concept relies on disabling parallel tasks to achieve single-core performance.

The proposed approach holds the potential to increase system performance on the application processor. However, its overhead needs to be considered as well. In the current implementation, a second CPU is needed for trace data processing and execution control. In the future, this software could be ported to an RTOS to enable the parallel execution of other tasks. Since the timely execution of the critical tasks depends on the correctness and timing of the execution control application, however, this needs to be done carefully. Alternatively, the execution controller could be implemented as a reconfigurable hardware module on the FPGA within the SoC, potentially reducing the reaction latency to incoming trace data and thereby further enhancing the system performance.

VII. CONCLUSION

Targeting mixed-criticality systems implemented on SoCs, comprising both real-time and best-effort applications, we proposed a novel, dynamic approach to scheduling such systems efficiently by exploiting parallelization opportunities that cannot be identified by static analysis. An *execution controller* is proposed which monitors the critical task at run-time and controls the parallel execution of best-effort tasks on neighboring CPU cores based on the critical task's progress. Leveraging the trace infrastructure of modern SoCs as well as partial WCET information obtained at design-time, the execution controller ensures that the critical task deadline is always met while parallel execution potential is used early. In doing so, the critical task does not need to be augmented, avoiding recertification. An evaluation using tasks of varying interference intensity confirms this and shows that an efficiency gain of 37 % to 41 % can be achieved for low-interference tasks in dual-core operation, compared to a priority-based sequential scheduling approach. For high-interference tasks contending for resources, we identify that parallel execution can lead to major slowdown, resulting in sequential execution to be more efficient. Therefore, next steps include further evaluation of the proposed concept using real-world applications and further theoretical application scenarios. To ease integration with larger system designs, the integration of the execution monitor software with an RTOS can be investigated. Finally, the trade-off between a hypervisor-internal and external implementation of execution monitor functionality needs to be investigated, considering both performance and safety aspects.

ACKNOWLEDGMENT

This work was funded within the EMDRIVE project by the German Federal Ministry of Education and Research (BMBF) under grant number 16ME0454. The authors are responsible for the contents of this publication.

REFERENCES

[1] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008.

[2] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *First International Workshop on Microkernels for Embedded Systems*, vol. 50, 2007.

[3] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, "XtratuM: a Hypervisor for Safety Critical Embedded Systems," in *In: Proceedings of the 11th Real-Time Linux Workshop*, 2009.

[4] S. Wegener, "Towards Multicore WCET Analysis," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, ser. OpenAccess Series in Informatics (OASIcs), J. Reineke, Ed., vol. 57. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 7:1–7:12.

[5] A. Burns and R. I. Davis, *Mixed Criticality Systems - A Review (13th Edition, February 2022)*, Feb 2022. [Online]. Available: https://eprints.whiterose.ac.uk/183619/

[6] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems," *ACM Comput. Surv.*, vol. 52, no. 3, Jun 2019. [Online]. Available: https://doi.org/10.1145/3323212

[7] J. Nowotsch and M. Paulitsch, "Leveraging Multi-core Computing Architectures in Avionics," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[8] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 109–118.

[9] E. Lara, G. Debon, R. Goerl, P. Villa, D. Schramm, L. B. Poehls, and F. Vargas, "A New Approach to Guarantee Critical Task Schedulability in TDMA-Based Bus Access of Multicore Architecture," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019.

[10] J. Freitag, S. Uhrig, and T. Ungerer, "Virtual Timing Isolation for Mixed-Criticality Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 13:1–13:23.

[11] Xilinx, "Zynq UltraScale+ Device - Technical Reference Manual," Dec 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm