Work in Progress: Predictable Execution of Isolated Real-Time Tasks on Multicore Systems Using the LET Paradigm

Konstantin Dudzik, Maximilian Kirschner, Victor Pazmino Betancourt, Jürgen Becker
FZI Research Center for Information Technology
Karlsruhe, Germany
{dudzik, kirschner, pazmino, juergen.becker}@fzi.de

***Abstract*—An ongoing trend in the domain of embedded computing systems is the consolidation of functionality on few, high-performance platforms. This development also impacts real-time systems, where a shift to parallel architectures enables meeting the increased throughput demands. On these multicore platforms, memory contention is a central concern regarding time-predictability. Additionally, isolation between tasks is required to limit the impact of faults during run time.**

We propose a scratchpad memory-based approach to predictable execution that integrates runtime-based isolation mechanisms with an LET-based task model. In order to mitigate interference between cores, each core executes from a local memory, while the data transfers between the local memories and the shared main memory are incorporated into a global, static schedule.

Our task execution model is based on the Logical Execution Time (LET) paradigm, which we extend to include explicitly scheduled data transfers similar to the Predictable Execution Model (PREM). The implementation and evaluation of our approach is ongoing and will be evaluated on a custom RISC-V-based multicore platform. This novel approach allows for consolidating hard real-time tasks with high demands for functional safety onto a single platform.

***Index Terms*—Predictable Execution, Multicore Processing, Logical Execution Time, Real-Time Systems**

I. Introduction

An ongoing trend in the domain of embedded computing systems with real-time requirements, such as automotive or aerospace applications, is the ever-increasing demand for computational performance. One contributing factor is the tendency to consolidate functionality on a small number of high-performance platforms. A core issue that arises when functionality from formerly distributed systems is consolidated into a central system with shared resources is the loss of previously existing temporal isolation and fault isolation boundaries. The main factor impairing the time-predictability of multicore systems is the contention for shared resources, particularly shared memory. As a result, the worst-case execution time (WCET) analysis of the whole application needs to account for interference between cores.

Predictable execution efforts, like the predictable execution model (PREM), aim to circumvent such interference by limiting the memory accesses during execution to memories local to the respective core [1]. Due to the composable nature of the resulting system, the WCET can instead be computed for individual software components, thereby turning an often intractable timing analysis problem into a scheduling problem [2].

In summary, the main contribution of this work consists of a novel predictable execution approach based on the LET paradigm that enables the consolidation of real-time applications by upholding fault isolation boundaries. To this end, each core executes from a local scratchpad memory. The necessary transfers of instructions and data to and from the local memories are performed according to a global, static schedule, thereby avoiding contention by design. We achieve a modular system design by integrating the LET paradigm, which allows for an efficient analysis of task execution on multicore systems [3]. Task execution is facilitated by a runtime that provides isolation mechanisms that limit the impact of faults to a single task. The implementation of the aforementioned runtime is the subject of ongoing work and targets a custom RISC-V-based platform that provides the necessary hardware features like local memories and memory protection units (MPU).

The paper is structured as follows. Section II gives a brief overview of the related work. Our concept for predictable execution is described in Section III. Next, Section IV concerns the implementation of the runtime, while Section V discusses future work and concludes the paper.

II. Related work

Time-predictable multicore architectures are a key part of high-performance real-time systems and have received significant attention in recent years [4]. One example of a hardware-centric approach is the T-CREST platform [5]. There, contention is avoided by a time division multiplexing (TDM) memory controller. This ensures that each core gets a fixed fraction of the memory bandwidth.

A more flexible approach compatible with COTS hardware was introduced with the predictable execution model (PREM) [1]. The key idea is to derive predictability from the joint scheduling of tasks and the accompanying transfers of instructions and data to and from local memories. Consequently, the program execution is split into memory and computation phases. Several different scheduling strategies that build on this core characteristic of PREM have been proposed. This

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00038

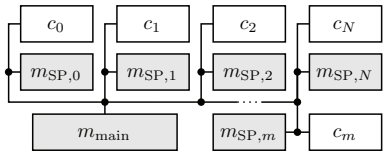

Fig. 1. The target hardware architecture consisting of application cores with local memories, a separate management core and a shared main memory

includes artificial deadline-based non-preemptive [6], global fixed-priority [7] and ILP scheduling [8].

To transform an existing application to fit the PREM model, it needs to be split into sections that can then be loaded as part of a memory phase. Suitable sections are single entry single exit (SESE) regions in the control flow graph (CFG).

Previous efforts have extended compilers to identify SESE regions to be used as the basic scheduling components in the PREM model [8]. However, the scope of scheduling optimizations is typically limited to one compilation unit. To address this shortcoming, the authors of [2] propose a PREM-compiler based on a genetic algorithm that includes system-level scheduling information.

Another approach to enable predictable multicore execution is the LET paradigm [9]. Its main idea is to enable deterministic communication by defining fixed communication instances at the boundaries of a task's LET [9]. Because of this, it has garnered attention in the domain of automotive software development as a way to construct predictable multicore applications. The LET abstraction allows developers to specify the system's schedule on an abstracted, logical level, allowing for a more modular approach to predictable execution than deriving the schedule directly from the CFG.

There are several approaches that consider the interference resulting from LET communication [3], [10], [11]. In these approaches, however, the task instructions are either statically allocated to local memories [11], limiting the complexity of the application, or not considered as part of the interference analysis [3], [10]. Similar in principle to [10], our approach also features direct memory access (DMA) as well as statically computed scheduling windows for said transfers. The predictable task execution model presented in this work, however, simultaneously considers transfers of data and instructions between memories.

Another feature of our approach is the integration of predictable execution as a core component of the runtime environment. This is in concept similar to the work presented in [12]. In contrast to our work, their approach targets sporadic tasks and divides memory access into fixed time slots in which transfers to and from local memories are scheduled.

III. Concept for predictable execution of isolated real-time tasks

A. System architecture

The underlying system architecture of our proposed predictable execution concept is a multicore platform with N application cores $\{c_0, c_1, \ldots, c_{N-1}\}$ and a separate management core c_m as seen in Figure 1. In addition to a shared main memory m_{main}, there exists a local scratchpad memory m_{SP} for each core. We assume a time-predictable interconnect between the cores and memories.

A static schedule, computed at compile time, describes the system behavior in terms of time-triggered *schedule windows* $W = (t_s, \tau)$, that are defined by a start time t_s and duration τ. Program execution is fundamentally split into *memory phases* $P_{\mathrm{mem}} = (W_{\mathrm{mem}}, \rho_{\mathrm{src}}, \rho_{\mathrm{dst}})$ and *computation phases* $P_{\mathrm{comp}} = (W_{\mathrm{comp}}, \rho_{\mathrm{inst}})$, defined by their respective memory regions and scheduling windows. For memory phases, these include source and destination regions, while computation phases are defined by their instruction region. The data transfers performed during memory phases are delegated to a dedicated direct memory access (DMA) engine.

B. Scheduling model

The logical schedule is composed of strictly periodic tasks $T = (p_{\mathrm{input}}, p_{\mathrm{output}}, \tau_{\mathrm{LET}})$, whose period is equal to their logical execution time τ_{LET}. Tasks communicate through input ports p_{input} and output ports p_{output} assigned to each task. According to the LET abstraction, the reading of input ports and publishing of output ports takes place instantaneously at the boundaries of the task's LET.

The LET semantics [9] also include the concept of modes that correspond to sets of interconnected tasks. Mode switches allow for a more expressive system behavior that can adapt to external conditions. In the context of this work, we assume that execution is limited to a single mode, although our concept can easily be extended to multiple modes. We refer to the period of that mode as the *hyper-period*, which is given by the least common multiple of the LET of the set of tasks that make up the mode.

In this work we consider an application consisting of n LET tasks T_i; i=1, …, n, each composed of a set of *runnables* r_{ij}; j=1, …, m. Each execution of a runnable is defined by a memory and computation phase. The constraints for the scheduling of the runnables are given by a directed acyclic graph (DAG). The LET semantics guarantee deterministic dataflow between tasks on the system level, while the subdivision of tasks into runnables enables fine-grained scheduling. Furthermore, this hierarchical task model enables compatibility with automotive software architectures like AUTOSAR [3].

C. Task execution from local memories

All data transfers between different memories are incorporated into the schedule as non-overlapping memory phases, whereas computation phases may be scheduled concurrently since they do not require access to shared memory. Thus, contention for memory access can be eliminated by finding a schedule that executes the application without overlapping memory transfer windows. To this end, we are pursuing an approach based on an SMT-solver, which aims to derive a feasible schedule based on constraints that result from the task execution model described in this section.

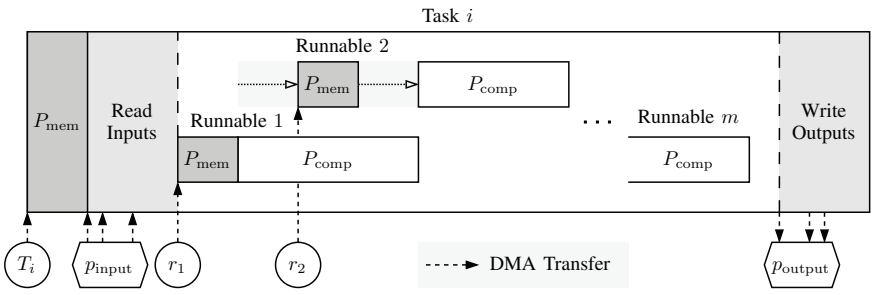

Fig. 2. Overview of memory and compute phases contributing to task execution

One crucial aspect of the isolation concept is that tasks may only access the address space of their assigned local memory. In order to facilitate task execution according to the LET-based model outlined above, the memory region of each scratchpad memory is divided into the following five sections. Overlapped execution and memory phases as seen in Figure 2 is enabled by reserving two sections for runnable code, $S_{\text{inst},A}$ and $S_{\text{inst},B}$. This assumes the use of dual-ported scratchpad memories. A third code section $S_{\text{inst},C}$ holds task-level code and functions that are shared between multiple runnables of a task. The data section S_{data} is common to all runnables of a task, which enables communication over shared variables. A separate data and instruction section is reserved for a task kernel, which is common to all tasks and loaded once during initialization. Both sections are protected from access by the application tasks.

At the start of each task, an initial memory phase is scheduled to load task-level code and data. During a second memory phase, the input port data is copied into the data section. Both phases may be scheduled adjacent and in arbitrary order. At this point, the code of the first runnable is loaded. This process is shown in Figure 2. Every runnable may freely access local copies of the task's input and output ports. At the end of the task execution, the content of the output ports is transferred to the respective buffers in main memory in a final memory phase.

Execution of one runnable and the memory phase of the next runnable are overlapped in order to maximize the core's utilization. Ideally, the memory phases that are overlapped with computation phases do not contribute to the task's WCET. However, memory phases may be delayed in the schedule beyond the end of the preceding computation phase. The ordered execution of runnables is accomplished by task-level code based on the constraints of the underlying DAG. Data transfers, on the other hand, are not performed by the task itself but by the management core in order to preserve the isolation between tasks.

Our concept also allows for task execution to be split into multiple scheduling windows. In the case of a task preemption, the task kernel saves and restores the state, both of which require a memory phase. Therefore, a task preemption may only be scheduled when no other memory phase is active. The overhead incurred by this process can be significant since large parts of the local memory need to be transferred to or from the main memory. However, the ability to interrupt task execution enables a broader range of scheduling strategies than strictly non-preemptable tasks.

D. Discussion

With this concept, we require the logical schedule as well as the partitioning of the application into tasks and runnables to be given. However, by requiring these design decisions to be made by the application developer, our approach offers the ability to include software components of different origins as tasks of a global schedule while all tasks are compiled separately. Compiler-based approaches, on the other hand, require a monolithic application in order to extract scheduling and partitioning information from the CFG [8].

Since the development of safety-relevant software based on a formal task model is common practice, it is advantageous to derive the partitioning of the application from the structure of the underlying task model. Furthermore, our approach separates task execution from runtime functionality, such as scheduling and data transfers, through the use of privilege levels and memory protection mechanisms. This limits the impact of faults resulting from implementation bugs in software components to the scope of the affected task, thereby addressing the functional safety concerns regarding the integration of software components with varying degrees of criticality and trust.

IV. Implementation

We are implementing our concept on a custom RISC-V multicore platform, which includes memories for each core and necessary peripherals, namely DMA and timer devices. Interrupts are raised for scheduling events according to the global, static schedule and propagated to the respective core. All scheduling events pertaining to data transfers between different memories within the system are serviced by the management core, which then programs the DMA engine to initiate the data transfer. The setup of the next DMA transfer can generally be performed while the current transfer is in progress and therefore does not cause additional overhead. Task preemption events, on the other hand, are propagated to the affected core and handled by the task kernel.

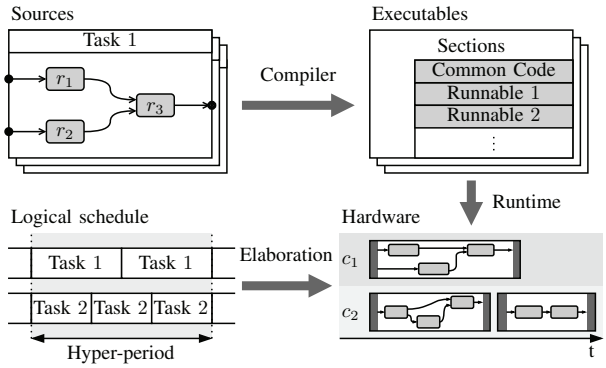

Fig. 3. Envisioned deployment process

Figure 3 shows the involved steps to deploy tasks on our platform. Each task is compiled into an individual executable, where each runnable is mapped to a separate section. The logical schedule specifies the high-level behavior of the application, including the mapping of runnables to tasks. An elaboration step is necessary to arrive at the hardware-level schedule that contains the scheduling windows as described in Section III. For the implementation of this process, we use the Z3 theorem prover[1] in order to derive a schedule consistent with our task execution model.

Each task corresponds to a separate image in the shared memory, where each runnable is linked to be executed from one of the two scratchpad sections. Since runnables may be mapped to different sections over the course of one hyper-period, it needs to be included in the task binary twice if is mapped to both sections.

All cores boot an initialization image from shared memory in the privileged machine mode. For the application cores, this involves setting up each core's MPU and pointing the machine mode exception vector to the protected region of the SPM, which is subsequently loaded with the task kernel image. Similarly, the management core's image, whose only function is to perform the DMA transfers, is loaded into its local memory. Since it runs in the privileged machine mode for the entire duration of program execution, it may access the complete address space, including peripherals.

The implementation is based on the Keystone TEE framework[2], which provides the mechanisms to isolate software components through the core's MPU registers and to save and restore their context. The extension of Keystone with our scheduling concept and adaptation to our local memory scheme is ongoing.

V. Conclusion and Outlook

Contention for memory access poses a significant challenge to the timing analysis of multicore real-time systems. In this work, we propose a predictable execution concept that leverages runtime functionality to avoid contention by design. We achieve a modular system design through the integration of the LET paradigm and a hierarchic task model, which closely resembles that of automotive real-time applications.

As of now, the basic scheduling and memory management mechanisms of the runtime are implemented, while the implementation of the task execution and LET communication are the subject of ongoing work. We plan to continue our work on the approach presented in this paper, which constitutes a first contribution towards a holistic predictable execution framework as outlined in Figure 3.

[1]https://github.com/Z3Prover/z3

[2]https://keystone-enclave.org/

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) with funding number 16ME0818.

References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A Predictable Execution Model for COTS-Based Embedded Systems," *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 269–279, Apr. 2011.

[2] B. Forsberg, M. Mattheeuws, A. Kurth, A. Marongiu, and L. Benini, "A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores," *2020 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 108–118, Jun. 2020.

[3] A. Biondi and M. Di Natale, "Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm," *2018 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 240–250, Apr. 2018.

[4] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, Feb. 2022.

[5] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, Oct. 2015.

[6] I. Senoussaoui, M. K. Benhaoua, H.-E. Zahaf, and G. Lipari, "Toward memory-centric scheduling for PREM task on multicore platforms, when processor assignments are specified," *2022 3rd International Conference on Embedded & Distributed Systems (EDiS)*, pp. 11–15, Nov. 2022.

[7] T. Thilakasiri and M. Becker, "An Exact Schedulability Analysis for Global Fixed-Priority Scheduling of the AER Task Model," in *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2023, pp. 326–332.

[8] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu, "Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution," *2018 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, pp. 11–20, Feb. 2018.

[9] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[10] P. Pazzaglia, D. Casini, A. Biondi, and M. D. Natale, "Optimizing Inter-Core Communications Under the LET Paradigm using DMA Engines," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 127–139, Jan. 2023.

[11] S. Igarashi, T. Ishigooka, T. Horiguchi, R. Koike, and T. Azumi, "Heuristic Contention-Free Scheduling Algorithm for Multi-core Processor using LET Model," *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–10, Sep. 2020.

[12] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems," in *2016 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2016, pp. 1–11.