Hardware-Assisted Context Preservation for Deterministic Task Switching in a RISC-V Real-Time Execution Model

Nicolai Iuga, Ionel Zagan, Nicoleta Cristina Gaitan and Vasile Gheorghita Gaitan

Abstract— **Temporal predictability in real-time preemptive embedded systems is strongly influenced by how execution contexts are interrupted and resumed. This paper investigates whether hardware-assisted context preservation can reduce the temporal jitter introduced by conventional task switching in a RISC-V real-time execution model. A controlled comparative evaluation is performed between a standard shared-pipeline execution configuration and a context-preserving multi-context execution configuration under identical scheduler-driven periodic workloads. The study focuses on determinism-oriented task switching indicators, including deadline misses, preemption counts, switch-related flush events, and accumulated switch overhead. Four control-oriented experimental scenarios are used to expose progressively different switching conditions, ranging from baseline periodic execution to frequent preemption, near-saturation behavior, and tight-deadline pressure. The results show that preserving execution context across preemptions consistently reduces switching overhead and improves deadline-related behavior under recurring real-time activations. The benefits of context-preserving execution become progressively more visible under elevated preemption pressure, reduced timing slack, and tighter deadline conditions. The study therefore provides a controlled comparative basis for analyzing deterministic task switching in RISC-V real-time execution models.**

I. INTRODUCTION

The growing use of RISC-V in embedded and real-time system design has increased the need for evaluation frameworks that can expose not only functional correctness, but also the timing effects introduced by execution organization and task-switching behavior. In control-oriented workloads, the practical challenge is not limited to instruction execution itself, but also to how execution state is preempted and restored when multiple periodic activities compete for processor time under preemptive scheduling conditions.

In conventional shared-pipeline execution, a task switch may introduce additional timing variability through pipeline timing degradation, refill delay, and short-term loss of useful progress after preemption. These effects are often less visible in functional instruction-level simulation, yet they become important when deadline compliance and repeatable switching behavior are treated as primary design objectives. Similar timing predictability requirements also appear in distributed embedded communication systems, where deterministic timing behavior and bounded execution latency directly influence real-time coordination and communication reliability [1]. Prior work on the nMPRA (multi pipeline register architecture, where n is the degree of multiplication) and nHSE (hardware scheduler engine for n threads) real-time architectural concepts has shown that hardware-assisted context management can reduce switching jitter by preserving execution state across preemptive scheduling events [2], [3], while related work has also extended this direction toward RISC-V-oriented real-time support [4]. The broader conceptual basis for predictable processor organization in small real-time applications was also outlined in earlier work on predictable processor architecture [5].

This paper investigates that idea through a comparative RISC-V simulation framework designed to evaluate deterministic task switching under periodic control-oriented workloads. Two execution organizations are studied within the same experimental environment: a conventional shared five-stage pipelined model and a context-preserving execution model aligned with nMPRA-inspired multi-context operation. The scheduler behavior, task structure, and simulation length are kept identical across both configurations, so the observed differences can be attributed to switching semantics rather than to differences in workload definition.

The evaluation uses four control-oriented task sets that emulate periodic monitoring, control computation, state update, and background activity under progressively tighter timing conditions. The comparison is centered on deadline misses, preemption counts, switch-related flush events, and accumulated switch overhead, with the objective of determining whether hardware-assisted context preservation improves switching determinism relative to a standard shared-pipeline organization.

The main contribution of this work is a controlled comparative demonstration that context-preserving execution reduces switching-induced overhead in a RISC-V real-time execution model, particularly under frequent preemption and tighter deadline pressure. The paper does not target a full RTL or FPGA-level hardware implementation. Instead, it focuses on a controlled simulation-based architectural comparison intended to isolate the timing effects of task-switch handling and execution-state preservation under identical scheduler-driven conditions. This abstraction level was intentionally selected in order to evaluate switching semantics and deterministic execution behavior independently of implementation-specific FPGA, memory-system, or synthesis-related effects. Hardware-level validation and extended embedded-system integration remain part of future work.

II. RELATED WORK

The prior work most directly related to this study comes from the nMPRA and nHSE concept of real-time processor architectures, where hardware-assisted task management and

preserved execution state were introduced as mechanisms for reducing the timing overhead associated with preemptive scheduling [2], [3]. These studies show that context-aware hardware support can improve execution continuity and reduce the cost of task switching in real-time environments. This architectural direction was later extended to FPGA-based embedded systems with explicit RISC-V support, reinforcing the relevance of hardware-assisted real-time event and context management in an open instruction set architecture (ISA) setting [4]. In addition, earlier work on predictable processor organization for small real-time applications provides a broader conceptual basis for treating timing determinism and task-switch behavior as architectural concerns rather than purely software-level effects [5].

Beyond the nMPRA/nHSE lineage, recent RISC-V research has also addressed low-latency interrupt handling and reduced context-switch cost through architectural support mechanisms. Balas et al. introduced CV32RT, a RISC-V microcontroller architecture explicitly designed to accelerate interrupt entry and context switching, showing that microarchitectural support can substantially reduce software-visible switching overhead [6]. Related work has also explored core-local interrupt controller (CLIC)-based fast interrupt extensions for embedded RISC-V processors, emphasizing reduced interrupt response latency through dedicated architectural mechanisms rather than software-only handling [7]. More recently, studies combining RISC-V processor microarchitectures with FreeRTOS extensions have directly targeted lower context-switch latency, highlighting that task-switch overhead remains a critical design concern even when hardware and software are optimized together [8]. While these efforts focus on interrupt responsiveness and operating-system-level context-switch acceleration, they do not primarily examine deadline-oriented switching behavior through a controlled comparison of shared-pipeline and context-preserving execution semantics under identical periodic workloads, which is the focus of the present study.

A related line of work addresses predictability more broadly as an architectural and timing-analysis objective in real-time and mixed-criticality systems. FlexPRET is a representative example of a processor platform explicitly designed for mixed-criticality execution, with an emphasis on timing predictability and controlled interaction between time-sensitive and non-critical activities [9]. More recent work has continued this direction by treating efficient and predictable context switching as a first-class architectural concern, showing that reduced switching cost and stronger timing composability remain central design goals in modern real-time systems [10]. In parallel, predictability is also being pursued at the toolchain level, where recent Low Level Virtual Machine (LLVM) based worst-case execution time (WCET)-oriented compiler autotuning shows that execution-time determinism depends not only on processor structure but also on how software transformations influence worst-case timing behavior [11].

A further group of related efforts focuses on simulation, virtual prototyping, and verification environments for RISC-V systems, but typically with objectives different from deadline-oriented real-time execution analysis. For example, recent instruction set simulator (ISS)-driven verification approaches use functional simulators as golden references for validating increasingly complex RISC-V microarchitectures, including superscalar designs, with an emphasis on correctness rather than scheduler-visible timing behavior [12]. Other work has proposed integrated co-simulation frameworks that combine functional verification and early performance evaluation through Universal Verification Methodology (UVM) and transaction-level modeling (TLM) abstractions, prioritizing design-space exploration and validation efficiency over cycle-accurate task-switch analysis [13]. In parallel, web-based and educational pipeline simulators provide accessible cycle-by-cycle visualization of RISC-V pipeline execution [14], while system-level virtual prototypes have extended RISC-V evaluation toward vector-enabled architectural exploration [15]. These tools are valuable for instruction-level understanding, verification, or broader architectural experimentation, but they do not primarily target comparative analysis of preemption behavior, deadline misses, and switch-induced execution degradation under controlled periodic real-time workloads, which defines the scope of the present work.

III. Experimental Framework and Evaluation Methodology

This section summarizes the execution models, experimental scenarios, and timing metrics used in the comparative evaluation. The objective is to isolate the effect of task-switch handling and context preservation under identical control-oriented workload conditions.

A. Execution Models and Switching Semantics

The comparative evaluation considers two execution organizations implemented within the same RISC-V simulation framework, both based on an identical five-stage pipeline composed of instruction fetch, decode, execute, memory, and write-back stages. The framework models scheduler-driven execution at explicit pipeline-stage level, including task preemption, pipeline-state progression, and switch-related execution effects. The present study does not target a specific physical RISC-V hardware platform, since the objective is a controlled comparison of execution-state handling semantics under identical workload conditions. The comparison isolates the effect of task-switch handling under scheduler-driven execution while keeping the instruction-processing model, task set, and simulation length unchanged.

In the standard pipelined configuration, all runnable tasks share a single pipeline instance and a common in-flight execution path. When the scheduler selects a different task, execution is redirected to the newly selected context, while partially progressed instructions associated with the previously running task are discarded. As a result, preemption introduces visible switch-related overhead through lost in-flight work, refill latency, and temporary execution discontinuity. Under this model, task switching is therefore reflected not only at the scheduling level, but also at the microarchitectural level through interruption of the active pipeline flow.

In the nMPRA-aligned configuration, each execution context preserves its own architectural and pipeline-related execution state. This includes the task-specific program counter, register context, and the in-flight pipeline state associated with that execution path. When a preempted task is

selected again, execution resumes from the retained context rather than re-entering the pipeline through a full refill sequence. Within the adopted simulation model, this preserves execution continuity across task switches and reduces the timing disturbance normally associated with shared-pipeline preemption.

The difference between the two execution models is illustrated conceptually in Fig. 1. While both configurations are subject to scheduler-driven preemption, only the shared-pipeline model turns a switch event into flush-and-refill execution loss, whereas the context-preserving model resumes execution from retained state. This distinction is important for interpreting the results, since deadline-related behavior depends not only on how often preemptions occur, but also on whether they introduce non-productive recovery cycles at the microarchitectural level.

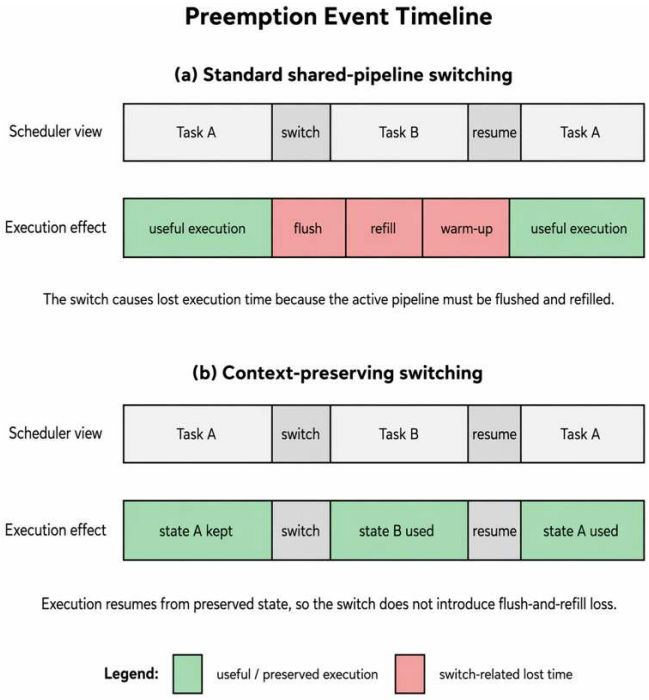

Figure 1. Conceptual comparison of scheduler-driven preemption behavior in the two evaluated execution models.

The scheduling behavior, priority ordering, task activation parameters, and simulation length remain identical in both configurations. Consequently, any observed differences in response-time variation, slack behavior, or deadline-related outcomes can be attributed to the switching semantics of the execution model rather than to differences in workload composition or scheduling behavior. In this formulation, the standard configuration serves as the shared-pipeline reference, while the nMPRA-aligned configuration represents a hardware-assisted context-preserving alternative intended to improve timing stability under preemptive real-time execution.

The comparison is not framed as a general throughput study. Instead, the two configurations are treated as alternative task-switching organizations, and the evaluation is centered on their effect on timing stability and deadline-oriented behavior under identical control-oriented workloads.

Unlike interrupt-oriented latency-reduction approaches or mixed-criticality execution platforms proposed in recent RISC-V real-time research, the present study focuses specifically on the effect of execution-state preservation on scheduler-visible task-switch behavior under identical periodic workloads. The objective is therefore not to optimize interrupt entry latency or operating-system services, but to isolate the timing effects introduced by shared-pipeline versus preserved-context execution semantics during repeated preemptive execution.

B. Experimental Method and Control-Oriented Workloads

The experimental study was designed as a controlled comparative evaluation in which the workload definition, priority assignment, release behavior, and simulation length were kept constant while only the execution organization was varied. This approach was adopted to isolate the timing effects of task switching and context preservation under identical scheduler-driven operating conditions.

All experiments use the same four periodic tasks, representing a compact control-oriented real-time workload composed of a high-priority sensing or monitoring activity, a control-computation task, a state-update task with memory interaction, and a lower-priority background or diagnostic activity. The programs assigned to these roles were kept unchanged across all evaluated scenarios in order to preserve comparability at the instruction-stream level. In this way, differences observed between the two execution models can be interpreted as consequences of switching behavior rather than as artifacts of altered program content.

To expose task-switching effects under progressively more demanding timing conditions, four experimental scenarios were defined. The first scenario (C1) represents a stable baseline under moderate processor utilization and is used to characterize nominal response-time behavior under limited preemption pressure. The second scenario (C2) increases the activation frequency of the highest-priority task in order to intensify preemption and make switch-related execution disturbance more visible. The third scenario (C3) operates close to the schedulability boundary without entering persistent overload, allowing near-limit timing dispersion to be observed under otherwise stable execution. The fourth scenario (C4) assigns a tighter relative deadline to the highest-priority task, emphasizing response sensitivity for the most critical control-oriented activity. The concrete task parameters associated with these scenarios are summarized in Table I.

Each scenario was executed in both the standard shared-pipeline configuration and the nMPRA-aligned preserved-state configuration, using the same task set, priorities, periods, relative deadlines, and release offsets. The simulation length was fixed at 10,000 cycles for all paired runs. No workload-specific tuning or scenario-dependent scheduler modifications were introduced between paired executions. This paired-experiment structure ensures that the comparison remains centered on the effect of execution-state handling during preemption and execution-state restoration, rather than on differences in scheduling behavior or workload composition.

The selected scenarios intentionally cover distinct operating regimes relevant to real-time control behavior:

nominal execution, elevated preemption pressure, operation close to the schedulability boundary, and tight-deadline responsiveness. Together, these scenarios provide a focused basis for assessing whether preserved execution state reduces timing variability and improves deadline-oriented behavior under preemptive pipeline execution.

TABLE I. CONTROL-ORIENTED EXPERIMENTAL SCENARIOS AND TASK PARAMETERS USED IN THE COMPARATIVE EVALUATION

Scenario	Task	Period (cycles)	Execution Budget (cycles)	Deadline (cycles)	Priority
C1	T0	40	6	40	3
	T1	60	10	60	2
	T2	80	12	80	1
	T3	120	8	120	0
C2	T0	20	4	20	3
	T1	50	12	50	2
	T2	70	14	70	1
	T3	100	10	100	0
C3	T0	25	5	25	3
	T1	40	10	40	2
	T2	60	15	60	1
	T3	120	14	120	0
C4	T0	30	6	20	3
	T1	60	12	60	2
	T2	90	16	90	1
	T3	120	10	120	0

C. Determinism-Oriented Timing Metrics

The comparative evaluation emphasizes timing predictability rather than aggregate execution throughput. Accordingly, the analysis focuses on metrics that capture how preemption and execution-state handling influence the temporal behavior of scheduler-driven real-time tasks under identical workload conditions. For each task, the simulator records the response time of every completed event, defined as the interval between task release and event completion. For task Ti and job k, the response time is defined as

$$R(i,k) = t_{\{finish\}}(i,k) - t_{\{release\}}(i,k), \quad (1)$$

where $t_{\{finish\}}(i,k)$ and $t_{\{release\}}(i,k)$ denote the completion and release times of job $k$for task T_i, respectively.

Based on these observations, the minimum, maximum, and average response times are computed for each task over the simulation length. Response-time jitter is defined as

$$J(i) = Rmax(i) - Rmin(i), \quad (2)$$

where $R_{max}(i)$and $R_{min}(i)$represent the maximum and minimum observed response times for task T_i, respectively. This measure provides a direct indication of timing variability across repeated activations of the same task.

Deadline-oriented behavior is evaluated through the total number of completed jobs, the number of deadline misses, and slack-based measures computed relative to the configured relative deadline. In this context, slack is defined as

$$S(i,k) = D(i) - R(i,k), \quad (3)$$

where $D(i)$ represents the relative deadline of task T_i. Positive slack indicates available timing margin, while negative slack directly reflects a deadline miss. Minimum and average slack values are therefore used to distinguish stable execution from scenarios approaching the schedulability limit.

To make the source of timing variation observable, the simulator also records the number of preemptions, the number of switch-related flush events in the shared-pipeline configuration, and recovery-related delays associated with the execution delay observed when a preempted task resumes execution. These measures are not treated as primary performance objectives, but as contextual indicators that help interpret differences in response-time variation and deadline behavior between the two execution models.

In the shared-pipeline configuration, switch overhead is defined as the cumulative number of non-productive cycles introduced by switch-related pipeline recovery behavior following preemption events. Within the adopted execution model, this overhead corresponds to the accumulated flush-and-refill delay associated with interrupted in-flight execution. The reported values in Table II represent absolute cycle counts measured over the full simulation interval rather than normalized ratios.

In this study, the reported preemption count reflects effective execution interruptions observed at the pipeline-execution level, rather than only scheduler-level task-selection decisions. Consequently, the preserved-state configuration may exhibit fewer counted preemptions when a scheduling change no longer produces the same effective execution interruption semantics observed in the shared-pipeline configuration. Conventional pipeline indicators such as retired instructions or aggregate instruction throughput may be retained as secondary context where useful, but they are not used as the primary basis for comparison. Accordingly, the analysis is intended to show whether preserved execution state reduces timing variability and improves deadline-oriented behavior under preemptive control-oriented execution.

IV. COMPARATIVE RESULTS AND DISCUSSION

The comparative results confirm that the preserved-state execution model consistently reduces switching-related timing jitter relative to the conventional shared-pipeline organization. Although the simulator records additional per-task timing metrics, the scenario-level comparison presented here emphasizes deadline misses and switch-related indicators because they most directly expose the effect of task-switch semantics across all four workloads. Across all evaluated control-oriented scenarios, the standard configuration exhibits visible deadline degradation under scheduler-driven preemption, while the context-preserving configuration maintains lower deadline-miss counts by eliminating repeated pipeline flush-and-refill behavior during task switches. The overall comparison is summarized in Fig. 2, which highlights the evolution of deadline misses and switch-related flush

events across the four scenarios, while the detailed numerical values are reported in Table II.

In the C1 baseline scenario, the difference between the two execution models is already observable, although the workload remains relatively moderate. The standard shared-pipeline configuration records 4 deadline misses, 161 preemptions, and 609 flush events, whereas the context-preserving configuration reduces these values to 2 deadline misses, 5 preemptions, and 0 flush events. This result indicates that even under nominal control-oriented execution, the conventional pipeline incurs measurable timing loss due to switching-related overhead. By contrast, the preserved-state model avoids this penalty and maintains a more stable execution flow. The contrast becomes substantially stronger in C2, which was designed to increase preemption pressure through more frequent higher-priority activations. In this case, the standard configuration reaches 47 deadline misses, together with 325 preemptions and 1219 flush events, while the context-preserving configuration reduces the deadline-miss count to 2, despite still experiencing 171 preemptions. This result suggests that the improvement is not explained only by fewer scheduling interruptions, but by the fact that preemptive execution no longer produces repeated microarchitectural overhead. The elimination of flush events in the preserved-state model therefore represents the main mechanism behind the improved deadline behavior observed in this scenario.

TABLE II. COMPARATIVE DEADLINE-ORIENTED RESULTS FOR THE EVALUATED CONTROL-ORIENTED SCENARIOS

Scenario	Config	Deadline Misses	Preemptions	Flush Events	Switch Overhead (cycles)
C1	Standard	4	161	609	1827
	Context-preserving	2	5	0	0
C2	Standard	47	325	1219	3657
	Context-preserving	2	171	0	0
C3	Standard	38	358	1013	3039
	Context-preserving	18	266	0	0
C4	Standard	29	270	945	2835
	Context-preserving	2	64	0	0

The absence of flush events in the preserved-state configuration is structural and follows directly from the retained execution-state semantics of the adopted execution model. Consequently, flush-event counts are not treated as independent performance metrics, but as explanatory indicators of switch-related execution disruption within the shared-pipeline organization.

A similar pattern is visible in C3, which places the workload closer to the schedulability boundary and reduces the available timing slack. Under these conditions, the standard shared-pipeline model produces 38 deadline misses, 358 preemptions, and 1013 flush events, while the context-preserving configuration reduces the miss count to 18 and completely eliminates flush events. Although the number of preemptions remains relatively high at 266, the results still show a clear improvement in timing behavior. This suggests that, near the schedulability limit, preserving execution state across context switches remains beneficial even when scheduling activity itself remains high. The effect becomes even more visible in C4, which emphasizes tight-deadline behavior through reduced timing margins. The standard configuration records 29 deadline misses, 270 preemptions, and 945 flush events, whereas the context-preserving configuration reduces the deadline-miss count to only 2, with 64 preemptions and 0 flush events. In this scenario, the reduced tolerance to switching penalties makes the architectural difference particularly visible. The preserved-state organization avoids the repeated refill cost associated with shared-pipeline switching and therefore preserves a much larger fraction of useful execution time for deadline-relevant work.

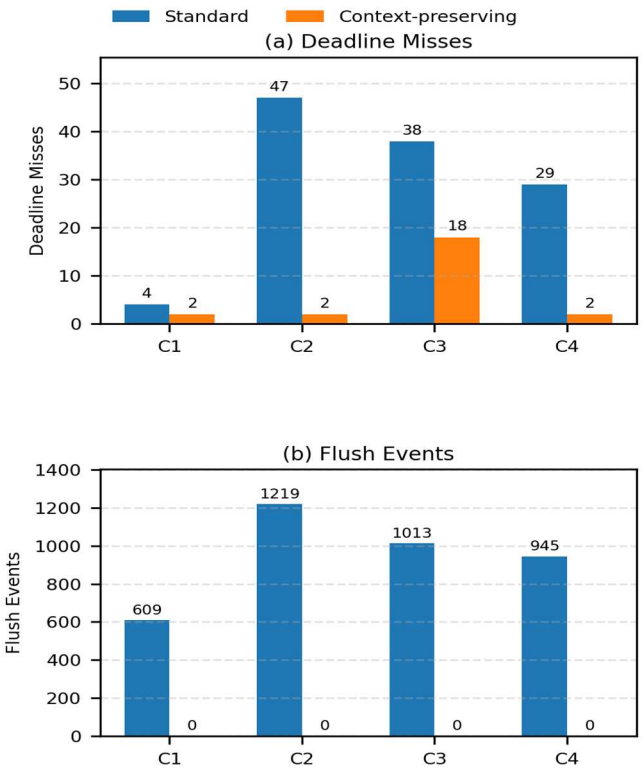

Figure 2. Comparative deadline-oriented behavior across the evaluated control-oriented scenarios: (a) deadline misses and (b) switch[IZ7.1]-related flush events for the standard shared-pipeline and context-preserving configurations.

These results clearly highlight the behavioral distinction between the two execution models. The critical factor is not only how often preemptions occur, but how much useful execution is lost when they occur. In the standard shared-pipeline configuration, each switch introduces repeated flush-and-refill effects that consume cycles and increase deadline pressure. In the context-preserving configuration, these penalties are removed, so task interruptions no longer produce the same level of execution loss. This is why the preserved-state model reduces deadline misses across all evaluated scenarios, with the strongest effect in the more demanding cases. Under these conditions, the advantage of hardware-assisted context preservation becomes directly visible. The

baseline scenario already shows a measurable improvement, but the difference becomes much clearer under heavier preemption pressure, reduced slack, and tighter deadlines. Preserving execution state across task switches keeps more cycles available for useful progress and leads to more stable deadline-oriented behavior than the conventional shared-pipeline alternative.

V. Conclusion

This paper presents a controlled comparative evaluation of task-switching behavior in a RISC-V real-time execution framework using preemptive control-oriented workloads. Under identical task and scheduling conditions, the study compares a conventional shared-pipeline execution model with a context-preserving execution model aligned with hardware-assisted multi-context execution principles.

The results show that the primary benefit of the context-preserving model is the removal of switch-induced microarchitectural overhead. Across all evaluated scenarios, the preserved-state configuration eliminated switch-related flush events and switch-overhead accumulation, while consistently reducing deadline misses relative to the standard shared-pipeline configuration. The benefit was visible in the baseline case and became substantially stronger under elevated preemption pressure, near-boundary timing stress, and tight-deadline conditions. These results indicate that preserving execution state across preemptive task switches improves timing robustness for control-oriented real-time workloads, although the study remains intentionally limited to a controlled simulation-level comparison of execution models and switching semantics. Overall, the results show that the key benefit of context preservation lies not in reducing scheduling activity itself, but in reducing the execution loss associated with each preemptive interruption. This suggests that deterministic real-time behavior depends not only on scheduling behavior, but also on how the underlying execution model handles task suspension and resumption at the microarchitectural level.

Future work will extend the analysis to more diverse real-time workloads, richer timing metrics, and deeper scheduler-aware architectural evaluation, including broader response-time and slack-based analysis under varying preemption conditions. It will also incorporate cache-related effects and memory-hierarchy interactions to evaluate how preserved execution state influences timing predictability in more realistic embedded RISC-V real-time systems.

Acknowledgment

This paper was supported by the North-East Regional Program 2021–2027, under Investment Priority PRNE_P1—A More Competitive, More Innovative Region, within the call for proposals Support for the Development of Innovation Capacity of SMEs through RDI Projects and Investments in SMEs, Aimed at Developing Innovative Products and Processes. The project is entitled “Arhitectura multi-core cu sistem de operare in timp real implementat prin hardware – mMCA_nHWRTOS”, Grant No. 724/02.07.2025, SMIS Code: 337817.

References

[1] N. C. Găitan, I. Zăgan, and V. G. Găitan, “Proposed modbus extension protocol and real-time communication timing requirements for distributed embedded systems,” Technologies, vol. 12, no. 10, Art. no. 187, Oct. 2024, doi: 10.3390/technologies12100187.

[2] V. G. Gaitan and I. Zagan, “An Overview of the nMPRA and nHSE Microarchitectures for Real-Time Applications,” Sensors, vol. 21, no. 13, Art. no. 4500, 2021, doi: 10.3390/s21134500.

[3] I. Zagan and V. G. Gaitan, “Implementation of nMPRA CPU Architecture Based on Preemptive Hardware Scheduler Engine and Different Scheduling Algorithms,” IET Comput. Digit. Tech., vol. 11, no. 6, pp. 221–230, Nov. 2017, doi: 10.1049/iet-cdt.2017.0163.

[4] I. Zagan, C. A. Tanase, and V. G. Gaitan, “Hardware Real-time Event Management with Support of RISC-V Architecture for FPGA-Based Reconfigurable Embedded Systems,” Adv. Electr. Comput. Eng., vol. 20, no. 1, pp. 63–70, 2020, doi: 10.4316/AECE.2020.01009.

[5] N. C. Gaitan, I. Zagan, and V. G. Gaitan, “Predictable CPU architecture designed for small real-time application—concept and theory of operation,” Int. J. Adv. Comput. Sci. Appl. (IJACSA), vol. 6, no. 4, pp. 47–52, 2015, doi: 10.14569/IJACSA.2015.060406.

[6] R. Balas, A. Ottaviano, and L. Benini, “CV32RT: Enabling Fast Interrupt and Context Switching for RISC-V Microcontrollers,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 32, no. 6, pp. 1032–1044, Jun. 2024, doi: 10.1109/TVLSI.2024.3377130.

[7] B. Mao, N. Tan, T. Chong, and L. Li, “A CLIC Extension Based Fast Interrupt System for Embedded RISC-V Processors,” in 2021 6th International Conference on Integrated Circuits and Microsystems (ICICM), 2021, pp. 109–113, doi: 10.1109/ICICM54364.2021.9660345.

[8] M. Scheck, T. Mürmann, and A. Koch, “Co-Exploration of RISC-V Processor Microarchitectures and FreeRTOS Extensions for Lower Context-Switch Latency,” in Proc. 30th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems, Vol. 2 (ASPLOS), Mar. 2026, pp. 394–408, doi: 10.1145/3779212.3790141.

[9] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, “FlexPRET: A Processor Platform for Mixed-Criticality Systems,” in Proc. 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr. 2014, pp. 101–110, doi: 10.1109/RTAS.2014.6925994.

[10] A. Nurmi, A. Kalache, H. Lunnikivi, P. Lindgren, and T. D. Hämäläinen, “Efficient and Predictable Context Switching for Mixed-Criticality and Real-Time Systems,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 33, no. 11, Nov. 2025.

[11] G. Magnani, D. Baroffio, F. Reghenzani, G. Agosta, and W. Fornaciari, “Modern LLVM-Based Compiler Autotuning for WCET Optimization,” in Proc. 2025 IEEE Real-Time Systems Symposium (RTSS), 2025, pp. 448–460, doi: 10.1109/RTSS66672.2025.00043.

[12] A. Galimberti, M. Vitali, S. Vittoria, and D. Zoni, “Functional ISS-Driven Verification of Superscalar RISC-V Processors,” in Proc. 2024 31st IEEE Int. Conf. Electronics, Circuits and Systems (ICECS), Nov. 2024, pp. 1–4, doi: 10.1109/ICECS61496.2024.10848613.

[13] R. Qiu and Y. Liu, “An Integrated UVM-TLM Co-Simulation Framework for RISC-V Functional Verification and Performance Evaluation,” arXiv preprint arXiv:2505.10145, 2025, doi: 10.48550/arXiv.2505.10145.

[14] R. Giorgi and G. Mariotti, “WebRISC-V: A 64-bit RISC-V Pipeline Simulator for Computer Architecture Classes,” arXiv preprint arXiv:2504.03722, 2025, doi: 10.48550/arXiv.2504.03722.

[15] M. Schlägl, M. Stockinger, and D. Große, “A RISC-V ‘V’ VP: Unlocking Vector Processing for Evaluation at the System Level,” in Proc. 2024 Design, Automation & Test in Europe Conf. & Exhibition (DATE), 2024, pp. 1–6, doi: 10.23919/DATE58400.2024.10546838.