



CVA6-VMRT: A Modular Approach Towards Time-Predictable Virtual Memory in a 64-bit Application Class RISC-V Processor

Christopher Reinwardt
Integrated Systems Laboratory
ETH Zürich
Zürich, Switzerland
creinwar@iis.ee.ethz.ch

Robert Balas*
ETH Zürich
Zürich, Switzerland
balasr@iis.ee.ethz.ch

Alessandro Ottaviano*
ETH Zürich
Zürich, Switzerland
aottaviano@iis.ee.ethz.ch

Angelo Garofalo
University of Bologna
Bologna, Italy
angelo.garofalo@unibo.it

Luca Benini
ETH Zürich
Zürich, Switzerland
University of Bologna
Bologna, Italy
lbenini@iis.ee.ethz.ch

Abstract

The increasing complexity of autonomous systems has driven a shift to integrated heterogeneous SoCs with real-time and safety demands. Ensuring deterministic WCETs and low-latency for critical tasks requires minimizing interference on shared resources like virtual memory. Existing techniques, such as software coloring and memory replication, introduce significant area and performance overhead, especially with virtualized memory where address translation adds latency uncertainty. To address these limitations, we propose *CVA6-VMRT*, an extension of the open-source RISC-V CVA6 core, adding hardware support for predictability in virtual memory access with minimal area overhead. *CVA6-VMRT* features dynamically partitioned Translation Look-aside Buffers (TLBs) and hybrid L1 cache/scratchpad memory (SPM) functionality. It allows fine-grained per-thread control of resources, enabling the operating system to manage TLB replacements, including static overwrites, to ensure single-cycle address translation for critical memory regions. Additionally, *CVA6-VMRT* enables runtime partitioning of data and instruction caches into cache and SPM sections, providing low and predictable access times for critical data without impacting other accesses. In a virtualized setting, *CVA6-VMRT* enhances execution time determinism for critical guests by 94% during interference from non-critical guests, with minimal impact on their average absolute execution time compared to isolated execution of the critical guests only. This interference-aware behaviour is achieved with just a 4% area overhead and no timing penalty compared to the baseline CVA6 core.

CCS Concepts

• **Computer systems organization** → **Real-time system architecture; Reconfigurable computing.**

*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CF '25, Cagliari, Italy
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1528-0/25/05
<https://doi.org/10.1145/3719276.3725172>

Keywords

RISC-V, CPU, Virtual-memory, Caches, Real-time, Mixed-criticality, Automotive, Predictability

ACM Reference Format:

Christopher Reinwardt, Robert Balas, Alessandro Ottaviano, Angelo Garofalo, and Luca Benini. 2025. CVA6-VMRT: A Modular Approach Towards Time-Predictable Virtual Memory in a 64-bit Application Class RISC-V Processor. In *22nd ACM International Conference on Computing Frontiers (CF '25)*, May 28–30, 2025, Cagliari, Italy. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3719276.3725172>

1 Introduction

With the growing adoption of advanced driver assistance systems (ADASs) and autonomous driving, automotive platforms face increased computational demand on electronic control units (ECUs). Addressing this demand through a federated system by adding separate ECUs for each task is impractical, as it significantly impacts the size, weight, power, and cost (SWaP-C) metrics. Therefore, there has been a transition from a federated, physically split approach to heterogeneous mixed criticality systems (MCSs), allowing multiple tasks to be executed concurrently on a single platform [13, 14].

When multiple timing-critical tasks are consolidated onto an integrated system-on-chip (SoC), maintaining execution time predictability becomes challenging due to the interference in shared resources, such as the processor, interconnects, and main memory. In this work, we consider inter-task interference within a single processor core. In particular, we focus on interference in the virtual-to-physical memory address translation process, which introduces uncertainty in memory access times and the variability caused by state-dependent cache latency.

Consider a scenario where a real-time task and a general-purpose task run in separate virtual machines (VMs) sharing a single processor core through time-division multiplexing. If the general-purpose task, such as rendering passenger information, is memory-intensive, the translation lookaside buffers (TLBs) and caches will likely store translations and data related to this VM. When a timing-critical interrupt for the real-time task occurs, it is probable that neither the required interrupt service routine (ISR) nor the corresponding page table entries (PTEs) are cached, necessitating page table

miss handling and potentially main memory accesses. This process is particularly costly in a virtualized environment, as each intermediate guest-level page table access requires a complete address translation at the hypervisor level. Consequently, the response time of the ISR is influenced by the processor's non-deterministic execution history, leading to wide worst-case execution time (WCET) estimates and inefficient hardware utilization.

Approaches to reduce this overhead include physical isolation on a processor core level by assigning tasks to separate processor cores depending on their criticality [9]. As the critical tasks are run in isolation from the other tasks, they do not suffer from interference in core-local resources. Other work reduces TLB interference by coloring the virtual addresses used in the task to limit the available TLB resources [11]. The approach in [2] uses a custom virtual memory architecture, employing fully software-managed TLBs. The decision of which virtual-to-physical translations need to be cached and which can be replaced is therefore taken by supervisor software, enabling arbitrarily complex replacement schemes. To reduce the non-determinism of memory access latency due to caches, the addition of separate software-managed core-local scratchpad memories (SPMs) provides the possibility to place critical code and data close to the processor core [1, 2, 5, 7].

However, these measures come with drawbacks. Using a custom virtual-memory architecture with software-managed TLBs or virtual address coloring adds complexity to the operating system (OS) and, in the former case, increases the number of page-fault interrupts and context switches. Adding additional processor cores or SPM memory macros causes significant area overhead for hardware resources that might be underused.

To address the interference challenges, in this work, we propose minimal hardware extensions and demonstrate them on an open-source application class RISC-V processor design. The extensions expose control over the conventionally hidden shared resources of TLBs and cache memory. By controlling the replaceable TLB entries, supervisor software can create isolated TLB states for different tasks. Furthermore, by dynamically changing the ratio of cache and SPM memory, supervisor software can choose the appropriate amount of SPM space without wasting area.

1.0.1 Contribution. We present the following contributions:

- (1) We design a software-configurable hardware mechanism to dynamically partition (section 2.1) and lock (section 2.2) TLB entries to prevent interference in the TLBs, thus ensuring constant and predictable memory address translation times in the presence of multiple competing virtual machines.
- (2) We design a runtime-configurable hardware unit that partitions the instruction and data caches into cache and SPM regions (section 2.4), ensuring fast and predictable memory access for critical VMs. This approach allows SPM resources to be tailored to application requirements with minimal area overhead.
- (3) We integrate the extended processor core into a mixed-criticality platform and evaluate it in a virtualized environment hosting guests of different criticality, focusing on interference mitigation for a critical guest. Our experiments (section 3) demonstrate a 94% reduction in execution time

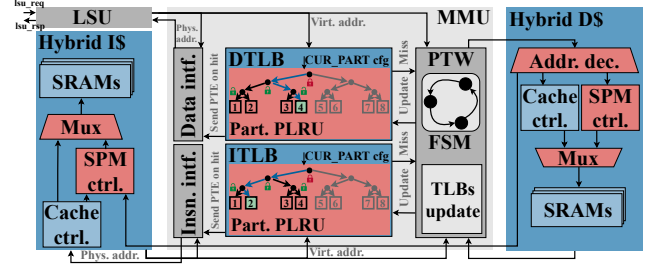


Figure 1: CVA6-VMRT MMU and hybrid cache/SPM subsystem. Modified architectural blocks are highlighted in red.

variability of the critical guest under interference from non-critical VMs, with minimal impact on the mean execution time, reclaiming most of the predictability of execution of the critical VM in isolation. To assess the cost of our changes, we synthesized the design in a 16 nm technology node, demonstrating no timing impact and an area overhead of only 4% with respect to the baseline core.

2 Architecture

Our approach to increasing the predictability of critical tasks focuses on the memory system of the processor core, namely the virtual-to-physical memory translation and the core-local cache architecture of the open-source processor core CVA6 (formerly Ariane) [16]. Figure 1 provides an overview of the memory management unit (MMU) and cache subsystem in CVA6-VMRT, where we highlight in red modified or new architectural blocks.

By dynamically controlling the available TLB resources (section 2.1) through the pseudo least-recently-used (PLRU) tree, we eliminate inter-task interference. Furthermore, by providing protected, software-defined overwrites of TLB entries (section 2.2), we enable software to specify cached PTEs directly. Additionally, by adding the option to trade cache for SPM space at runtime (section 2.4), CVA6-VMRT allows the OS to trade-off between average-case performance and predictability of the access time to critical code and data. The following sections detail the implementation of the proposed extensions.

2.1 PLRU TLB replacement

2.1.1 Vanilla CVA6 PLRU policy. A TLB caches virtual-to-physical translations to avoid the cost of a page table walk for every virtual memory access. CVA6 has two dedicated level-1 TLBs for instructions and data. The TLBs in CVA6 are implemented as fully associative register-based caches, storing the recently used PTEs. As the number of available TLB entries is limited, some entries are evicted from a fully occupied TLB by hardware once a new entry needs to be cached. CVA6 uses a PLRU approach for this decision, aiming to evict no longer needed entries while preserving recently used PTEs. To efficiently track the usage history of TLB entries, CVA6 uses a binary-tree-based PLRU implementation. When a new TLB entry is required in a full TLB, the eviction logic traverses the PLRU tree from the root to a leaf node, following the currently *chosen* branch at each internal node. The reached leaf node is then replaced by

the new entry, and all chosen edges on the used path are flipped away from the newly added node. Similarly, on every TLB hit, the edges on the path towards the entry containing the hitting PTE are pointed away from this entry to maximize its lifetime.

2.1.2 CVA6-VMRT PLRU partitioning mechanism. To gain control over the available TLB resources, we modify the PLRU replacement scheme by adding constraints on each node in the PLRU tree, dictating which edges (if any) are available for traversal. These constraints are specified in a custom CUR_PART control and status register (CSR) as a bitmap, where each bit corresponds to one partition. A partition refers to a power of two of TLB entries, given by the total number of TLB entries divided by the number of partitions, which can be parameterized. If a bit in the CUR_PART CSR is set, the corresponding TLB resources are modifiable and protected if their associated bit is clear. By modifying this bitmap, the OS or hypervisor can dynamically select the amount of TLB entries accessible to running tasks. If different tasks get assigned non-overlapping partitions, CVA6-VMRT ensures complete separation of TLB replacements and isolates the TLB states against interference from other tasks sharing the same processor core.

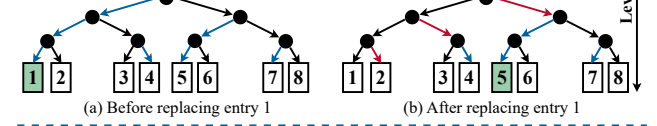
In addition to the CUR_PART CSR tracking the currently enabled partitions, we add two auxiliary control registers to aid in switching between different configurations quickly. The additional registers are LAST_PART and RESTORE_LAST_PART. Whenever the CUR_PART CSR is written, the LAST_PART register is updated with the value in CUR_PART before the write. To restore the LAST_PART state again into the CUR_PART register, a write to the RESTORE_LAST_PART register with the least significant bit (LSB) set suffices. The LAST_PART and RESTORE_LAST_PART registers are beneficial during context switching as they help to save and restore the TLB partition bitmap used by the interrupted task. This allows CUR_PART to be overwritten with the first instruction in an interrupt handler, minimizing the handler's impact on the previous task's TLB state. If the supervisor handler determines to return to a different task with a differing set of active partitions, overwriting the LAST_PART CSR with the new value as part of the task switch lets the trap handler restore the new state into the CUR_PART register.

2.2 CSR-based TLB locking

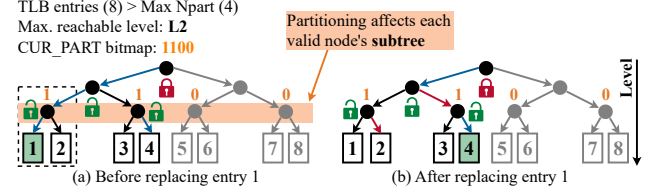
In conjunction with control over the replacement scheme, we also provide direct control over a parameterizable amount of TLB entries, using a combination of three CSRs per locked entry. This allows the OS or hypervisor to statically define TLB entries, preventing their replacement. The content of these software-provided TLB entries does not necessarily need to exist in the system's page tables, giving the management software full flexibility. The OS has to provide (i) the virtual page number (VPN) and flags of the mapping, (ii) the leaf PTE for the translation, and (iii) the ASID or VMID value used for this mapping using three CSRs.

Once all three registers have their valid bit set, the TLB entry associated with this locking slot is marked as unreachable in the PLRU tree, and its contents are provided by the CSRs. This way, we can ensure a constant latency for virtual-to-physical memory translations covered by the locked entry, which is helpful for hard real-time tasks with strict response time requirements, as the memory translation latency is made small and predictable.

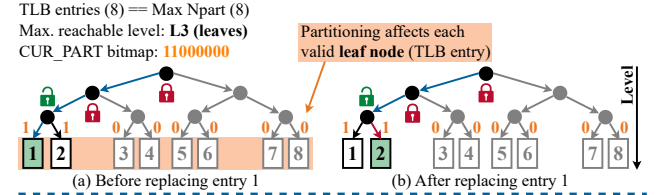
(A) Vanilla TLB PLRU



(B1) Partitioning-enabled TLB PLRU



(B2) Partitioning-enabled TLB PLRU



(C) Locking TLB PLRU (ex. without partitioning)

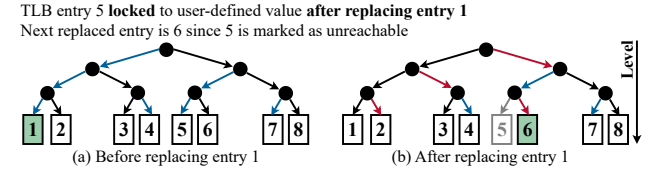


Figure 2: PLRU behavior example for an eight-entry TLB.

2.3 Exemplary PLRU behavior

To illustrate the partition-enabled PLRU policy, figure 2 provides an example of an eight-entry TLB with a visual representation of the corresponding PLRU tree that would be used in CVA6. In all cases, we show the state before (a) and after (b) replacing the entry at index 1. The blue edges show the path from the tree root to the next entry to be evicted. The red edges illustrate how PLRU updates the tree to select the next TLB entry for replacement by flipping the edges along the chosen path.

In figure 2 A, the default case is shown, which results in entry number 5 being marked as the next victim by the tree. In B1 and B2, we show our partition-enabled PLRU tree with control over the tree's branching possibilities. For B1, we show the case where the number of TLB entries is larger than the number of partitions, meaning a single bit in CUR_PART controls access to an entire subtree. In contrast, in B2 we have a one-to-one mapping of partitions to TLB entries, as the number of entries and partitions match. In both cases, we observe that TLB entries associated with cleared bits in the CUR_PART CSR are protected from replacement by being unreachable from the tree root. Case C highlights how the behavior of the non-partitioned PLRU tree changes if we lock TLB entry number 5. After replacing entry number 1, the tree would generally point to entry number 5. However, once a locking becomes valid, the associated leaf node is transparently marked as unreachable, independently of the state of the CUR_PART CSR. Therefore,

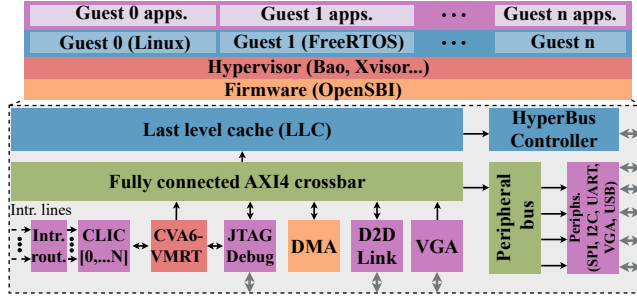


Figure 3: Cheshire microarchitecture and software stack.

the entry that will be replaced next in case C is entry number 6. As shown in figure 3, TLB locking and partitioning can cooperate to improve virtual-to-physical memory address translations of concurrent mixed-critical tasks.

2.4 SPM extended cache controllers

To further extend our system’s timing predictability, we enhance the L1 data and instruction caches with a hybrid cache/SPM mode. This extension trades a part of the cache memory for software-controlled SPM, providing core-local memory with constant access time that is not influenced by hardware.

We achieve this by selectively removing cache ways from the associativity available to the cache replacement logic and reusing these memory macros for the SPM (figure 1). Beyond modifying the way selection logic, we ensure that the associated tags and valid bits are cleared so that accidental cache hits on SPM data are impossible. To distinguish between SPM and other memory accesses, we extend the cache controllers with address decoding logic that selects the correct destination, given the physical address of the memory access. This way, the SPM regions can be mapped into the standard address space seen by the processor core.

The cache ways are mapped contiguously in the SPM region, simplifying the mapping from the physical address to the corresponding cache way in hardware. Before accessing the memory, the hardware ensures the target cache way is configured as SPM. If the way is not configured correctly, write requests are silently dropped, and read requests or instruction fetches respond with dummy data to avoid stalling the processor.

By utilizing this hybrid cache/SPM mode, the OS can ensure that essential data and instructions, for example, critical interrupt handlers and their associated interrupt stacks, are always available with minimal latency, minimizing the overall interrupt handler latency.

3 Evaluation

3.1 Evaluation Framework

We use the minimal Linux-capable host platform Cheshire [10] in a virtualized environment using Xvisor [12] as virtual machine monitor (VMM) to evaluate our hardware modifications. Figure 3 shows Cheshire’s block diagram and software stack with virtualization support. To collect a statistically significant set of measurements, we

use the Digilent Genesys II field programmable gate array (FPGA) implementation of Cheshire, extended with CVA6-VMRT.

Interference Setup. The scenario we consider is inspired by the trend of merging multiple different ECUs into a single zonal controller using virtualization in modern automotive systems. It consists of two virtual machines running atop Xvisor on a single CVA6-VMRT core. One of these VMs executes a timing-critical task. At the same time, the other virtual machine hosts a Busybox + Linux v6.1.64 environment running a synthetic memory-intensive process. This process uses as many TLB entries as possible by continuously accessing different virtual memory pages. We use the Linux VM as *noise source* of our system to evaluate the isolation capabilities of our hardware extensions.

Time-critical Benchmarks. We consider two types of time-critical tasks: representative benchmarks from state-of-the-art (SOTA) suites and synthetic benchmarks. The latter are crucial for assessing the extended hardware features in a controlled, streamlined environment. As synthetic benchmark, we use a minimal bare-metal application designed to access the same number of virtual memory pages as there are TLB entries. As a representative benchmark, we select *powerwindow* from the TACLeBench[4] suite, which models the control system of powered windows in modern vehicles. This benchmark represents control-intensive real-time tasks commonly found in automotive platforms, such as ADASs or engine control systems.

Xvisor Extensions. We extend the Xvisor virtual machine configuration to enable support for the TLB partitioning and locking mechanisms. This allows to specify a set of TLB partitions on a per VM basis as well as marking individual memory regions of each VM as TLB locked. TLB locking is only supported when the targeted memory region is naturally aligned in both the physical and virtual address spaces and when their size is a multiple of the base page size or one of the SV39 super page sizes. A new locking entry is used for each required page table entry. In hardware, it is possible to add locking support to all TLB entries; however, we chose to configure Xvisor to support at most eight locked entries as that is sufficient for our evaluation cases and minimizes the code repetition, which is necessary to access the different CSRs for each entry. The creation of locked memory regions is handled during the initial VM creation and uses a first-come, first-served basis, so it is up to the user to allocate locked entries to VMs.

To handle TLB partitioning, Xvisor requires minimal changes to its trap handler to save and restore the active partitions. With the first instruction in the interrupt handler, we overwrite the CUR_PART CSR with a compile-time constant partition reserved for the hypervisor to minimize TLB disturbance. The last instruction of the handler uses the RESTORE_LAST_PART register to atomically restore the value from LAST_PART into CUR_PART. If the trap did not cause a rescheduling of VMs, this restores the partition register to the state before the trap was taken. However, we also modify the VM switching code path to set the LAST_PART CSR to the TLB partition bitmap configured for the VM that is scheduled next. This way, the last instruction of the trap handler switches to the new set of allowed TLB partitions when the currently active VM is changed.

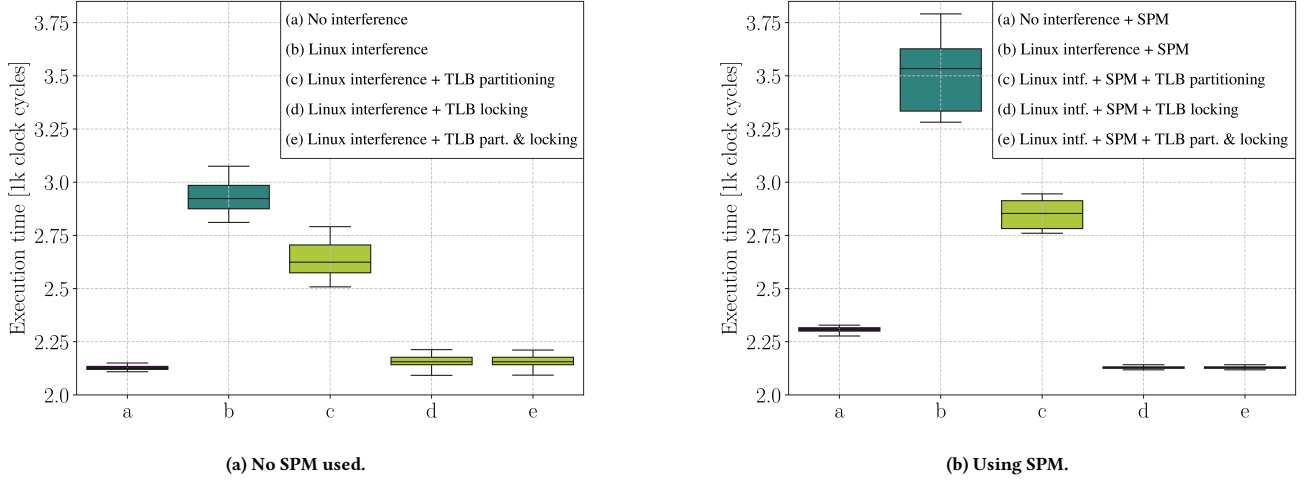


Figure 4: Box plots for the synthetic benchmark using different interference mitigation configurations. Without our SPM extensions, TLB locking achieves a standard deviation reduction of 60 % compared to the unmitigated case. Using SPM additionally to locking further decreases the standard deviation, bringing the total reduction to 91 %.

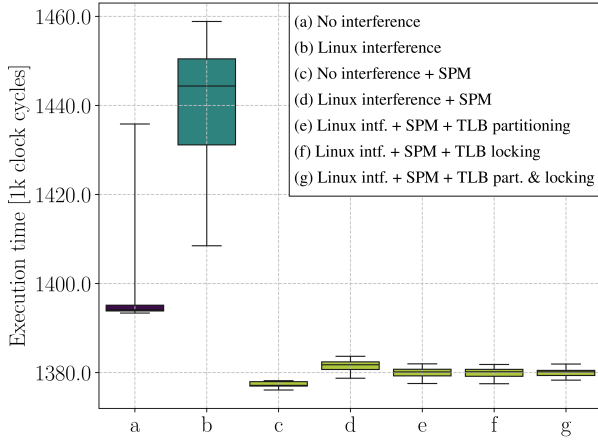


Figure 5: Results of the *powerwindow* benchmark for varying levels of enabled interference mitigations. Combining TLB partitioning and locking with the SPM functionality CVA6-VMRT achieves a 94 % reduction in execution time standard deviation over the unmitigated case.

Evaluated Configurations. In the test scenarios that utilize TLB partitioning, we reserve one-half of the PLRU tree for the critical VM, a single entry of the other half for Xvisor itself, and the remainder for the Linux VM. If TLB locking is used, we use TLB entries from the Linux VM to store the locked entries as it is the less critical VM.

For the tests using the hybrid SPM, we statically partition the available cache memory into 50% cache and 50% SPM during boot, exposing the SPM regions as regular memory to Xvisor, staying

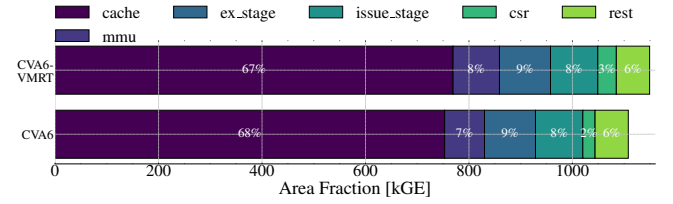


Figure 6: CVA6-VMRT area breakdown. The overall area increase compared to vanilla CVA6 is about 3.7 %.

entirely transparent to the VMs. We choose this ratio to balance the predictability improvement for the VM using SPM with the impact of the reduced cache size on the non-SPM VM. This trade-off should be carefully considered for each real-life application. We allocate as much data and code as possible to the available SPM sections. In the case of the *powerwindow* benchmark, we cannot fit the entire data set into the data scratchpad memory (DSPM) without changing the 50% split ratio, so we keep the ratio unchanged and prioritize large continuous data structures for allocation into the DSPM, filling the remaining space with smaller variables.

Performance Metrics. The primary performance metrics are the mean and standard deviation of the clock cycles required to complete a time-critical task under three scenarios: isolation, unregulated TLB interference, and regulated TLB interference with the proposed hardware extensions. The smaller the standard deviation, the higher the execution time predictability of the critical task. To evaluate inter-VM interference, the synthetic benchmark primes the TLBs without timing, then measures execution after de- and re-scheduling by Xvisor, accessing previously primed virtual memory pages in reverse order to avoid self-eviction of TLB entries.

Experiments are conducted over 10000 iterations for all scenarios, analyzing the impact on the mean and standard deviation of execution time (section 3.2). Hardware overhead is assessed by synthesizing Cheshire in a FinFET Intel 16 nm technology node (section 3.3).

3.2 Functional results

In figure 4a, we show the results for the synthetic benchmark without utilizing our hybrid cache/SPM functionality. The X-axis shows the TLB interference mitigations in use and whether the interference VM is running. The Y-axis reports the distribution of the benchmark's execution time as a multiple of 1000 clock cycles.

Scenario (a) marks the best-case scenario, with no other VM active in the system. In figure 4a-(b), the Linux interference VM is enabled, causing a 38 % increase in average execution time and a standard deviation increase of 652 %. In figure 4a-(c), we partition the data and instruction TLBs in the three non-overlapping sections mentioned above. This partitioning reduces the impact of the interference on the mean execution time by 36 %, but at the same time, further increases the standard deviation by 10 %.

This issue is addressed by scenarios (d) and (e), which add locked TLB entries, covering the entire memory space of the benchmark VM. This reduces the mean execution time overhead over the isolated case to 1.4 %. The remaining difference is due to other unmitigated interference sources like the caches or the memory controller. By utilizing TLB locking, we achieve a decrease in the standard deviation of around 60 % compared to the unmitigated case.

We also plot the combined use of TLB partitioning and locking for completeness. However, the result remains within the margin of error compared to the locking-only configuration, as the locking mechanism already handles all necessary memory translations. As this might not be possible in all cases, for example in scenarios where the hypervisor is dynamically creating and tearing down virtual machines, potentially using fragmented physical memory to back a continuous region of memory in a VM, we still see the case for TLB partitioning, as it protects against external interference without assumptions on the virtual-to-physical memory mapping.

The same sequence of mitigations is applied to figure 4b, with the only difference being the use of the hybrid SPM for the main benchmark routine, where most execution time is concentrated. The first three configurations show between 8 % and 19 % increased mean execution times compared to their counterparts that do not use the SPM in figure 4a. This happens because the benchmark routine is placed in a separate code section, limiting applicable compiler optimizations and reducing code locality.

Additionally, the SPM-enabled benchmark requires an extra PTE for the SPM region. However, once the virtual-to-physical translation is provided statically from a locked TLB entry, the average execution time falls to the level of the isolated case without SPM use (figure 4a, scenario 1). When the complete set of CVA6-VMRT extensions is utilized (i.e., TLB locking in combination with SPM usage), we see a decrease in execution time standard deviation of around 91 % compared to the unmitigated case that is not using SPM.

For the *powerwindow* benchmark, we combine both the results without and using the hybrid SPM functionality in figure 5 to highlight the cumulative isolation effect of our extensions. The box plot

(a) shows the isolated case in which only the benchmark VM is present in the system without mitigations activated.

Configuration (b) introduces the Linux interference VM, which causes a 3 % increase in the mean and a 23 % increase in the standard deviation of the execution time. In scenario (c), we add our hybrid L1 cache/SPM functionality to scenario (a) and move the application code and most of its data into the private SPM partitions. This marks the best-case scenario for the benchmark, as most data resides in non-evictable memory, and the remaining data can only be evicted by hypervisor interference.

In scenario (d), we re-introduce the interference from the Linux VM, which now causes a mean execution time increase of 0.3 % and a standard deviation increase of 89 %. Compared to the isolated case without SPM (a), the use of SPM reduced the standard deviation by 92 %. Nonetheless, the following three scenarios show that interference through the TLBs still harms the standard deviation of the execution time, as our TLB partitioning (e), TLB locking (f), and the combination of both (g) manage to reduce it further.

In combination, our extensions achieve a reduction of execution time standard deviation of around 94 % for the *powerwindow* benchmark.

Our findings show that CVA6-VMRT can significantly reduce the standard deviation of the execution time experienced by virtualized critical tasks in the presence of interference by other VMs. This increases the timing predictability of the protected critical tasks. In our benchmarks, we have the case that the SPM-backed code and data are always part of the working set.

Therefore, the average execution time is reduced, as this statically allocates a part of the working set to fast memory, which cannot be evicted. If this is not the case, the mean execution time may increase due to the reduced cache resources available.

This trade-off should be carefully considered when deciding what data and instructions should be placed in SPM and how much cache space can be sacrificed.

3.3 Physical implementation

To evaluate the area overhead of the changes introduced by CVA6-VMRT, we synthesize identically configured CVA6s, both with and without our extensions, using an Intel 16 flow at 600 MHz. We use the RCSS (slow) technology corner at 0.72 V and 125 °C. In both cases, the instruction cache is configured to be 16 KiB large and the data cache 32 KiB. The instruction and data TLBs are configured to hold 16 entries each.

Figure 6 provides an area breakdown in kilo gate equivalents (kGE) of the two configurations. The two main hardware units that grew in size are the MMU, which contains both TLBs, and the caches. In both cases, the area increase is only caused by the control logic and CSRs we added, as we reuse all memory macros. The MMU incurs an overhead of 16.4 %, and the caches of 2.2 %. Our extensions did not noticeably affect the timing. Compared to the whole CVA6 processor core, all changes require a mere 3.7 % of additional area, which we consider negligible compared to the gained feature set.

4 Related Work

We focus on techniques or existing processors that try to achieve deterministic virtual-to-physical address translation and deterministic instruction and data execution for critical tasks. For this, previous works explored multiple approaches to increase TLB and cache content predictability. Table 1 summarizes other works and compares them on their control over TLB resources and use of SPM.

Panchamukhi *et al.* extend the coloring technique from caches to the TLBs in [11]. By controlling the virtual addresses returned from *malloc*, it is possible to ensure that the corresponding PTE entries do not collide in the set associative L2 TLB, enabling TLB partitioning without hardware overhead. This, however, relies on OS support and does not map well to our virtual machine host case. With *DTLB* [15] Varma *et al.* propose a modified TLB architecture that allows to create and restore backups of the TLB state. To increase the memory translation time predictability of tasks preempted by an OS, the TLB entries are saved as part of the task's state and restored when the task is rescheduled. This partitions the TLBs in time instead of space, retaining the number of available TLB entries while still providing isolation across processes. However, without hardware support for TLB entry locking, processes can still suffer from the timing variability caused by intra-process TLB interference.

A different approach to timing predictable virtual memory translation is taken by the Infineon TriCore [2], which does not support virtual memory using page tables. Instead, it exposes software-managed memory map segments, which the OS configures with the currently active memory address translations. This means address translations have a constant latency or cause a page fault exception. While this achieves the same as our TLB locking, it severely limits the system's usability, as the OS has to manage the range of currently accessible virtual memory on a very fine-grained basis. Additionally, the TriCore features separate SPMs that coexist with the caches to provide predictable and low latency instruction and data access.

To increase the timing predictability of critical tasks, software-managed memories have been employed in ARM's Cortex-R82AE [8], which adds separate memory macros for SPM. These SPMs are meant for critical code sections and data, so fast and predictable access latencies can be guaranteed. However, they can only be accessed from a protected memory system architecture (PMSA) context and are not usable when ARM's virtual memory system architecture (VMSA) is active, forbidding memory address translation when accessing SPMs. The intention is to provide a high-performance, 64-bit real-time processor that seamlessly integrates into a heterogeneous SoC while supporting rich OSs. With multiple R82AE cores in a system, the trade-off between rich OS and real-time compute power can then be managed dynamically by re-assigning cores to different OSs. The limitation is the physical-only SPM addressing, making it harder to use in virtualized contexts. Our implementation allows for full virtual or physical addressing without incurring the considerable area overhead of completely separate memory macros.

Similarly, to reduce the memory access time variability during instruction fetches, Cilku *et al.* rely on single-path code, prefetching,

and cache locking in their work on time-predictable instruction-cache architecture [3]. By exploiting the regular structure and simpler control flow in single-path code, the proposed prefetcher is guided on what instructions should be fetched sequentially and where loop prefetching is necessary. This way, given the absence of external interrupts, it is possible to provide deterministic instruction access times. Unfortunately, modern applications and execution environments do not fulfill these requirements, and fully software-managed memories like our hybrid L1 cache/SPM are more manageable.

The Gaisler LEON 5 [1] core supports tightly coupled memories, which can be added additionally to the default cache memories. Regardless of the page table contents, they are mapped to a specific virtual address. To provide isolation, the accessibility can be restricted to a single MMU context, or access can be granted to all contexts simultaneously. While this allows the use of virtual memory on SPM, it is a limitation compared to the arbitrary mappings supported by our implementation and still incurs the area overhead of separate memory macros. Additionally, the core provides a mechanism to freeze cache contents upon reception of an asynchronous interrupt. In this mode, the cache contents are kept coherent with the main memory, but no cache lines are evicted or replaced, preserving the set of cache lines present in the cache before the interrupt was taken. This is an effective way to retain predictability for a single protected thread but is not beneficial in a multi-process environment with multiple critical processes, as only one thread can be protected from interference by other threads. Additionally, the granularity of protection is the entire cache. In contrast, our SPM can be made available to arbitrary numbers of threads at a granularity of a single cache way.

The heterogeneous PolarFire SoC from Microchip [5] consists of one physical addressing only monitor core and four application class cores that support the RISC-V SV39 virtual memory specification. The monitor core features a software-managed DSPM instead of a data cache and supports reconfiguration of up to 50% of its 16 KiB instruction cache into instruction scratchpad memory (ISPM). The application class cores do not include the DSPM and instead rely on a standard L1 data cache. Their instruction cache retains the hybrid cache/ISPM functionality and allows for up to 28 KiB of ISPM. This memory can hold arbitrary data and instructions, but data load and store operations are less efficient than the monitor core's dedicated DSPM. Compared to our unified approach, one limitation of this approach is that the predictability and feature set are not uniform across all available cores. The monitor core can provide the highest predictability but no virtual memory. In contrast, the application cores can support a virtualized environment but suffer from less efficient data accesses to the ISPM and worse overall memory access time predictability due to the unmitigated memory address translation. This reduces the system's flexibility and increases the complexity of the programming model.

The Arm Cortex-A8 [6] core offers both TLB lockdown and L2 cache lockdown. Using TLB lockdown, it is possible to retain the contents of certain TLB entries to ensure constant address translation latency. In secure privileged modes, it is furthermore possible to manually specify the contents of arbitrary L1 TLB entries. The processor's L2 cache lockdown mechanism provides the user control over cache replacements at the granularity of cache ways.

Reference	Virtual memory support	Predictable core-local memory	TLB control	Control approach	Memory macro reuse
Arm Cortex-A8 [6]	Armv7 VMSA	L2 cache locking & preloading	✓	HW	✓
Arm Cortex-R82AE [8]	Armv8 VMSA	SPMs (physical addressing only)	✗	-	✗
Gaisler LEON 5 [1]	SPARC V8	Cache freezing & optional SPMs (physical addressing only)	✗	-	✓ ✗
Infineon TriCore [2]	Custom (TLB only)	SPM	✓	HW	✗
Microchip PolarFire SoC [5]	RISC-V SV39	Partial hybrid cache/SPM	✗	-	✓
Panchamukhi et al. [11]	Required	n/a	✓	SW	-
Varma et al. [15]	Required	n/a	✓	HW	-
Cilku et al. [3]	n/a	SW-managed IS w/ prefetching & locking	-	-	✓
This work	RISC-V SV39x4	Hybrid L1 cache/SPM	✓	HW	✓

Table 1: Comparison of approaches to increased system predictability focusing on the virtual memory system and core-local memories.

With this approach, L2 cache ways can effectively be used as SPM by locking all other cache ways and pre-loading the desired data by accessing each corresponding cache line, finally inverting the locking to disable evictions of the pre-loaded data. This ensures critical data and instructions are kept in the L2 cache, achieving the same predictability as an L2 SPM without needing a separate memory macro. While this feature set is comparable to ours, it differs in key points. The cache lockdown feature is only available in the second cache level, meaning the user has to compromise on performance or predictability by disabling or enabling the L1 caches, respectively. Another key difference is the usability aspect, as the locked cache ways are not directly writeable but are pre-loaded using cache misses. This also requires care to ensure no conflict misses in the critical data or code. Our implementation avoids these complications by providing direct load and store access to the hybrid L1 SPM.

To the best of our knowledge, no state-of-the-art work combines a predictable virtual memory system with runtime configurable hybrid L1 cache/SPM partitions to reduce variability in critical virtual memory accesses on RISC-V.

5 Conclusion

This paper presents *CVA6-VMRT*, an enhanced version of the open-source RISC-V processor CVA6, which facilitates dynamic, software-controlled partitioning of TLB resources and enables the runtime configurable trade-off of cache space for deterministic SPMs to improve predictable execution of time-critical tasks in non-federated, virtualized MCSs. In experiments with two virtual machines competing for TLB and cache resources, *CVA6-VMRT* successfully reduced execution time variability of the protected virtual machine by 91 % in synthetic benchmarks and 94 % in application-class benchmarks. When implemented using a commercial 16 nm technology node, *CVA6-VMRT* introduced only a 3.7 % area overhead compared to the baseline CVA6 processor, making it a cost-effective extension for MCSs based on CVA6.

Acknowledgments

This work has received funding from the Swiss State Secretariat for Education, Research, and Innovation (SERI) under the SwissChips

initiative and was partly supported through the ISOLDE project that has received funding from Chips Joint Undertaking (CHIPS-JU) under grant agreement nr. 101112274. CHIPS JU receives support from the European Union's Horizon Europe's research and innovation programme and Austria, Czechia, France, Germany, Italy, Romania, Spain, Sweden, Switzerland.

References

- [1] Frontgrade Gaisler AB. 2024. GRLIB IP Core User's Manual. <https://download.gaisler.com/products/GRLIB/doc/grip.pdf>. Revision: Dec 2024, Version 2024.4, Retrieved December 31, 2024.
- [2] Infineon Technologies AG. 2002. TriCore™ 1.3 32-bit Unified Processor Core, Architecture Overview Handbook. https://www.infineon.com/dgdl/TC1_3_ArchOverview_1.pdf?fileId=db3a304312bae05f0112be86204c0111. Retrieved March 24, 2024.
- [3] Bekim Cilku, Daniel Prokesh, and Peter Puschner. 2015. A Time-Predictable Instruction-Cache Architecture that Uses Prefetching and Cache Locking. In *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 74–79. doi:10.1109/ISORCW.2015.58
- [4] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIS), Vol. 55)*, Martin Schoeberl (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.
- [5] Microchip Technology Inc. 2024. PolarFire SoC MSS Technical Reference Manual. https://www.microchip.com/content/dam/mchp/documents/FPGA/ProductDocuments/ReferenceManuals/PolarFire_SoC_FPGA_MSS_Technical_Reference_Manual_VC.pdf. Retrieved December 31, 2024.
- [6] ARM Limited. 2010. Cortex-A8 Technical Reference Manual. <https://developer.arm.com/documentation/ddi0344/k>. Revision: r3p2, Retrieved December 31, 2024.
- [7] ARM Limited. 2024. Arm® Cortex®-R82 Processor Technical Reference Manual. <https://developer.arm.com/documentation/102670/0300/?lang=en>. Revision: r3p0, Retrieved September 20, 2024.
- [8] ARM Limited. 2024. Arm® Cortex®-R82AE Processor Technical Reference Manual. <https://developer.arm.com/documentation/101550/0000/?lang=en>. Revision: r0p0, Retrieved September 20, 2024.
- [9] Arm Limited N. Werdmuller. 2020. Arm Cortex-R82: Combining high-performance 64-bit real-time and applications processing for the next generation of storage devices. <https://community.arm.com/arm-community-blogs/b/internet-of-things-blog/posts/arm-cortex-r82-high-performance-64bit-realtime-applications-processing>. Retrieved September 20, 2024.
- [10] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. 2023. Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3777–3781. doi:10.1109/TCSII.2023.3289186
- [11] Shrinivas Anand Panchamukhi and Frank Mueller. 2015. Providing task isolation via TLB coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 3–13. doi:10.1109/RTAS.2015.7108391
- [12] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. 2015. Embedded Hypervisor Xvisor: A Comparative Analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 682–691. doi:10.1109/PDP.2015.108
- [13] N. Santhanam R. Fletcher, A. Mahindroo and McKinsey A. Tschiesner. 2020. The case for an end-to-end automotive software platform. <https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/the-case-for-an-end-to-end-automotive-software-platform>. Retrieved September 20, 2024.
- [14] Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. 2021. The Road towards Predictable Automotive High - Performance Platforms. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1915–1924. doi:10.23919/DATE51398.2021.9473996
- [15] Kajal Varma, Geeta Patil, and Biju Raveendran. 2017. DTLB: Deterministic TLB for Tightly Bound Hard Real-Time Systems. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. 207–212. doi:10.1109/VLSID.2017.50
- [16] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640. doi:10.1109/TVLSI.2019.2926114