ISA Support for Hardware Resource Partitioning in RISC-V

Nils Wistoff[1]*, Robert Balas[1], Alessandro Ottaviano[1], Gernot Heiser[2], Luca Benini[1,3]

[1]ETH Zürich, [2]UNSW Sydney, [3]University of Bologna

Abstract

In modern computing environments, applications concurrently executing on the same system often compete for shared hardware resources, such as caches and buffers. The ensuing contention can lead to timing interferences, posing significant threats such as deadline misses in real-time systems and the creation of timing channels in secure systems. This work proposes an ISA extension based on the RISC-V Capacity and Bandwith Controller QoS Register Interface (CBQRI). Our proposal enables dynamic, comprehensive temporal and spatial partitioning of shared hardware resources, ensuring the isolated execution times of concurrent applications.

Introduction

Modern computing systems are often time-shared between multiple applications. By doing so, these applications compete for shared hardware resources such as caches, buffers, and branch predictors, impacting each other's execution time. This becomes an issue in two types of systems.

Mixed Criticality Systems comprise applications with different safety and timing requirements, such as engine control and infotainment in a vehicle. Timing interference can cause deadline misses of real-time applications due to unpredictable execution time.

Mixed Confidentiality Systems run mutually untrusting applications, such as a mail client and a browser on a personal device or virtual machines on a computing server. As previously demonstrated, interference in execution time caused by contention for hardware resources can be leveraged as a *timing channel* to transfer information across isolation boundaries [1, 2].

Depending on the system's requirements in such safety- and security-critical environments, an OS must be able to partition shared hardware resources partially or completely. Generally, a resource can be partitioned *spatially* by sub-dividing it and allocating its discrete elements (e.g., ways or sets of a cache) or *temporally* by setting it to a predefined state. Mutually trusting or non-critical applications can be grouped into *domains* to reduce the overhead of partitioning.

Previous works have proposed ISA extensions for temporally partitioning on-core microarchitectural state [3, 4, 5]. A similar approach is currently being discussed in the RISC-V Microarchitecture Side Channels Special Interest Group (uSC SIG) [6]. However, hitherto, these proposals do not address off-core components that could leak data and violate real-time requirements.

One option to spatially partition shared caches is by *cache colouring*, which leverages the physical memory layout to map isolated applications to non-overlapping cache sets [7, 8]. This approach is applicable only to last-level caches in systems where the physical page bits coincide with the cache index bits, significantly constraining its applicability and adaptability.

A more general approach to system-level hardware resource management is being discussed in the RISC-V Capacity and Bandwidth Controller Quality-of-Service Register Interface Task Group (CBQRI TG) [9]. They propose tagging memory requests with resource control IDs (RCIDs), indicating the originator domain, and splitting hardware resources into *capacity units* that can be allocated to one such RCID, providing corresponding performance guarantees. However, the CBQRI lacks mechanisms for complete temporal and spatial partitioning, ensuring strict non-interference of execution times.

In this work-in-progress, we propose a general spatial and temporal hardware resource partitioning methodology based on the CBQRI semantics and discuss previous and ongoing implementation efforts.

Architecture

Similar to the CBQRI, each shared hardware resource can be spatially divided into one or more *capacity units*, as shown in Figure 1. These are the smallest fraction of the resource that can be allocated to a domain. As in the CBQRI, memory requests are tagged with the requesting domain's RCID to specify the target capacity units. However, this proposal extends the CBQRI with the following properties:

Flushable capacity units. There is a mechanism to flush a capacity unit (writing back dirty state and setting it to a predefined state) to temporally partition it. This is needed for deterministic execution time after a capacity unit was re-allocated, and to

*Corresponding author: `nwistoff@iis.ee.ethz.ch`

This work was supported in part through the TRISTAN (101095947) project that received funding from the HORIZON CHIPS-JU programme.

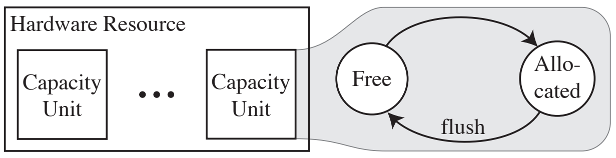

Figure 1: *Partitioning shared hardware resources.*

allow predictable access latency to data that is shared between domains.

Lifecycle. Capacity units have a lifecycle, as shown in Figure 1. Software must flush the capacity unit when de-allocating it to ensure deterministic execution time after re-allocating it.

No inter-block accesses. The state of a capacity unit allocated to one domain must not impact the execution time of another domain. In particular, this means that a cache hit in a capacity unit allocated to an RCID cannot be returned if the request is tagged with a different RCID. This property ensures strict spatial partitioning.

System-level ubiquity. All hardware resources in a system with variable timing, including the CPU, must consist of one or more capacity units that can be temporally partitioned. Otherwise, non-partitionable hardware resources can create a timing channel.

Time padding. There needs to be a (set of) mechanism(s), e.g., a `pad` instruction, that an OS can use to enforce a constant context-switch latency, including flushing of capacities to prevent timing channels through context-switch latency [5].

These changes are intended to be flexible by maximising the observability and controllability of the underlying hardware so that an OS can implement a partitioning policy depending on the system's safety, security, and performance requirements.

Examples

Temporally partitioning on-core microarchitectural state. The previously proposed `fence.t` instruction [5] matches the proposed architecture by defining the processor core's microarchitecture as a hardware resource with a single capacity unit. When re-allocating the microarchitectural state on a context switch from one application to another, an OS can execute `fence.t` to temporally partition this capacity unit at a minimal overhead.

Spatially partitioning caches. A single capacity unit in a cache can, for instance, comprise the entire cache, one way, or one set. To flush a capacity unit, the cache controller needs to write back any dirty cache lines and invalidate all lines. Notably, the cache controller itself may contain state with a timing impact and thus need to be partitioned. A last-level cache with support for such flexible way-partitioning is currently under development.

Temporally partitioning the system bus. Time slicing regulates shared bus traffic by allocating specific bandwidth and time periods to each system master. A hardware helper module manages traffic at the bus ingress and egress, ensuring deterministic bounds, as shown in [10]. This method aligns with CBQRI, with helper modules acting as programmable capacity units by the OS or hypervisor. Note that an extension of this scheme is necessary for interference-free secure domains, as literature mostly focused on deterministic guarantees for the real-time application domain.

Discussion and Future Work

While some hardware resources can, by design, only contain a single capacity unit and need to be temporally partitioned, such as arbiters on the system's control path, for other resources, the number of capacity units provided by hardware and concurrently allocated by software is generally a performance trade-off. We emphasise that we expect the addition of the proposed mechanisms to come at low hardware overhead. At the same time, depending on the system's requirements, an OS is free to use only a subset or none of them.

Ongoing and future work can explore these trade-offs in resources such as on-core microarchitecture. Furthermore, proofs of concept and empirical evaluations of the proposed mechanism in last-level caches, memory controllers, system interconnect, and their integration into an OS are ongoing.

References

[1] C. Percival. "Cache missing for fun and profit". In: *BSDCan.* 2005.

[2] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *IEEE S&P'19.* 2019.

[3] N. Wistoff et al. "Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core". In: *DATE'21.* 2021.

[4] M. Escouteloup et al. "Under the dome: preventing hardware timing information leakage". In: *CARDIS'21'.* 2021.

[5] N. Wistoff et al. "Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning". In: *IEEE Trans. Comput.* 72.5 (2023), pp. 1420–1430.

[6] RISC-V uSC Special Interest Group. *Fence.t, a RISC-V extension proposal.* 2024.

[7] R. E. Kessler and M. D. Hill. "Page Placement Algorithms for Large Real-Indexed Caches". In: *TOCS* 10.4 (1992), pp. 338–359.

[8] Q. Ge, Y. Yarom, and G. Heiser. "No Security Without Time Protection: We Need a New Hardware-Software Contract". In: *APSys'18.* ACM, 2018, 1:1–1:9.

[9] RISC-V CBQRI Task Group. *RISC-V Capacity and Bandwidth QoS Register Interface.* 2024.

[10] T. Benz et al. "AXI-REALM: A Lightweight and Modular Interconnect Extension for Traffic Regulation and Monitoring of Heterogeneous Real-Time SoCs". In: *arXiv preprint arXiv:2311.09662* (2023).