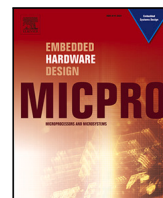




Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Bridging hardware and software: Dynamic runtime-reconfigurable RISC-V ISA extensions with hardware-managed software fallback and seamless state handover[☆]

Tobias Scheipel¹*, Maximilian Ogris¹, Marcel Baunach¹

Institute of Technical Informatics, Graz University of Technology, Inffeldgasse 16/1, Graz, 8010, Austria

ARTICLE INFO

Keywords:

Dynamic Partial Reconfiguration
RISC-V
Instruction Set Extensions
Runtime-reconfigurable Accelerators
Reconfigurable Computer Architectures
Processor Architecture
Hardware/Software Co-design
Seamless Hardware/Software Handover
Embedded Systems

ABSTRACT

Embedded and edge devices increasingly require lifetime adaptability: once deployed, they must accommodate evolving workloads, standards and protocols, and dependability and security requirements under strict energy and cost constraints. This paper presents a runtime-reconfigurable FPGA-based RISC-V microcontroller architecture that combines dynamic partial reconfiguration with standardized custom-instruction integration via SCAIE-V. Accelerators are implemented as reconfigurable modules hosted in reconfigurable partitions and invoked via a custom instruction. We showcase the approach with two soft cores – the non-pipelined PicoRV32 and the pipelined Orca – showing that the design is mainly independent of the underlying microarchitecture.

Beyond standardized accelerator integration, we contribute a system-level execution mechanism that preserves ISA-level semantics under runtime reconfiguration. When an accelerator is absent, dedicated hardware transparently triggers a controlled context switch to an equivalent software fallback function, then restores the architectural state so that execution proceeds as if the custom instruction had executed in hardware, including program-flow changes. To maintain correctness for stateful accelerators during reconfiguration, we introduce a memory handover manager that enables seamless migration across the hardware/software boundary. An automatic reconfiguration module loads accelerators on demand while the software fallback remains available.

We evaluate the concept on an AMD Artix-7 FPGA using a control-flow-intensive reduced-overhead loop and an ASCON-based lightweight cryptographic accelerator. For ASCON, hardware execution reduces runtime from 126,017 to 3515 clock cycles on PicoRV32 and from 66,191 to 1800 clock cycles on Orca, while the software fallback adds only 20.23% and 18.04% overhead over regular software, respectively. In contrast, the ROL case study exposes the fixed cost of full semantic transparency, reaching 1289.93% and 1089.45% fallback overhead. These results show that the approach is most beneficial for accelerators with sufficient per-invocation work and reuse to amortize fallback and reconfiguration costs, while maintaining correctness and software-level compatibility under dynamic accelerator availability.

1. Introduction and motivation

The slowdown of Moore's Law has pushed computer architectures further towards *specialization*. Instead of executing all workloads on homogeneous general-purpose cores, modern systems increasingly rely on domain-specific accelerators and custom instructions to meet tight performance, energy, and economic constraints. In Internet of Things (IoT) devices, embedded systems, and edge nodes, this pressure is amplified by long product lifetimes: once deployed, devices must adapt to evolving workloads, protocols, or security requirements without frequent hardware replacement.

At the same time, open Instruction Set Architectures (ISAs), most prominently RISC-V [1,2], have made it straightforward to introduce application-specific functionality at the ISA and microarchitectural level.

Reconfigurable logic provides a practical vehicle for extending this flexibility beyond design time. Full-fledged Field-Programmable Gate Arrays (FPGAs) and embedded FPGAs within System on Chips (SoCs) allow post-fabrication modification of implemented logic, while DPR refines this capability by time-sharing a fixed pool of logic resources among multiple hardware functions at runtime. In effect, rarely used

[☆] This article is part of a Special issue entitled: 'DSD 2025' published in Microprocessors and Microsystems.

* Corresponding author.

E-mail addresses: tobias.scheipel@tugraz.at (T. Scheipel), m.ogris1@gmx.at (M. Ogris), baunach@tugraz.at (M. Baunach).

URL: <https://www.scheipel.com> (T. Scheipel).

<https://doi.org/10.1016/j.micpro.2026.105281>

Received 8 January 2026; Received in revised form 25 March 2026; Accepted 12 May 2026

0141-9331/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

functionality can be moved out of the static design and loaded on demand, while frequently used units remain resident.

Previous work has explored several ways to integrate reconfigurable logic with processors, ranging from memory-mapped coprocessors to tightly coupled ISA Extensions (ISAXs). In parallel, emerging frameworks now provide standardized interfaces for attaching such ISAXs to existing RISC-V cores at the Register Transfer Level (RTL) level, easing accelerator integration into the pipeline.

What has received comparatively less attention is the *system-level behavior* of such instruction set extensions when combined with runtime reconfiguration:

- How can software remain functionally correct and compatible¹ if a custom instruction is executed while its corresponding hardware accelerator is absent because the reconfigurable region currently hosts a different accelerator?
- How can we avoid stalling or deadlocking the processor pipeline during long DPR operations?
- How can we preserve and transfer the internal state of an accelerator between its hardware implementation and a software fallback, such that computations can seamlessly migrate across the hardware/software boundary?

Naive solutions, such as simply trapping on an “illegal” custom instruction and aborting execution, are unacceptable in realistic systems. Instead, we need *bridging mechanisms* that make custom instructions behave as stable ISA-level abstractions even in the presence of dynamic hardware reconfiguration. This paper, being an extended version of our previous work [3], addresses precisely this gap by combining a standardized RISC-V ISAX interface with a runtime-reconfigurable FPGA-based MCU architecture and a hardware-backed software fallback. Using SCAIE-V [4] to integrate custom instructions into the core pipeline, we implement an end-to-end flow that dispatches each custom opcode either to a dynamically loaded hardware accelerator or – if the accelerator is absent – to a transparent, hardware-managed software fallback. Our architecture targets soft-core RISC-V MCUs on an off-the-shelf FPGA and is instantiated with two representative cores: the non-pipelined PicoRV32 [5] and the pipelined Orca [6].

A key contribution of the present work is a generic software fallback mechanism that preserves the illusion of a stable ISA despite changing hardware. When a custom instruction is executed, and its hardware accelerator is unavailable, dedicated hardware support captures the instruction context. It triggers a controlled context switch to a developer-friendly C/C++ level fallback function. This function emulates the accelerator’s behavior in software, including potential changes to the program flow (e.g., branches or loop constructs). Afterwards, it returns control to the hardware such that the architectural state appears as if the hardware accelerator had executed. From the programmer’s perspective, the custom instruction preserves its architectural behavior (register/PC effects and control flow), while non-functional properties such as timing, energy, electromagnetic, and side-channel characteristics may change depending on whether hardware acceleration is available.

To prevent loss of internal accelerator state when reconfigurable regions are reused, we introduce a *seamless memory handover* mechanism. Accelerators expose their internal state through a narrow interface, allowing it to be serialized into regular memory before reconfiguration and reloaded when the accelerator is instantiated again. The software fallback implementation accesses the same logical state via a shared memory-resident layout, enabling seamless migration of long-running computations between hardware and software. Together with

¹ Here, “compatibility” refers to the ability to run the same software across variants with different (or dynamically changing) accelerator availability. Note that this notion of compatibility refers to software behavior; partial bitstreams remain device- and partition-specific artifacts.

an automatic reconfiguration module that triggers DPR according to a configurable policy (miss-triggered in our prototype, and extensible to usage-based policies), this enables on-demand loading of accelerators without sacrificing correctness or requiring explicit coordination at the application level.

We evaluate our architecture on an AMD Artix-7 FPGA using two representative case studies: (i) a ROL instruction that implements a control-flow intensive loop skeleton, and (ii) a lightweight cryptographic accelerator for ASCON [7], which exhibits substantial arithmetic complexity and internal state. Both are implemented as RMs and as software fallback functions, and are invoked through custom instruction opcodes. Our results quantify the performance and overhead of hardware acceleration, software fallback, and automatic reconfiguration on both PicoRV32 and Orca, and analyze under which conditions the dynamic approach outperforms pure software execution.

In summary, this paper makes the following contributions:

- We present a generic MCU architecture including the SCAIE-V interface [4] that integrates dynamically reconfigurable RISC-V custom instructions with DPR, supporting both non-pipelined and pipelined soft cores.
- We design and implement a transparent software fallback mechanism for custom instructions that preserves ISA-level semantics,² including program-flow changes, while allowing hardware accelerators to be absent or swapped at runtime.
- We introduce a memory handover mechanism for accelerator state, enabling seamless migration of computations between hardware accelerators and their software fallbacks.
- We provide a proof-of-concept implementation and experimental evaluation on an AMD Artix-7 FPGA, demonstrating the behavior of our approach with a ROL instruction and an ASCON accelerator across two different microarchitectures.

The remainder of this paper is organized as follows: Section 2 introduces the necessary background and terminology on dynamic partial reconfiguration and RISC-V extension interfaces. Section 3 then reviews related work on reconfigurable accelerators and runtime-extensible processors. Section 4 presents the architecture of our concept, whereas Section 6 provides an evaluation and proof of concept. From the evaluation, we derive design guidelines presented in Section 7, we conclude this paper with Section 8, where we also outline directions for future work.

2. Background and terminology

This work combines three building blocks to enable *lifetime* hardware specialization: (i) runtime-adaptable hardware via DPR on FPGAs, (ii) RISC-V custom instruction support, and (iii) standardized and structured ISA extension interfaces. This section introduces the concepts and terms that recur throughout the paper and provides the “red thread” that ties DPR, ISAXs, and runtime management.

Dynamic Partial Reconfiguration. DPR enables the reconfiguration of selected FPGA regions at runtime while the remaining design continues to operate. A design is partitioned into a *static* region (e.g., processor, interconnects, memories, control) and one or more RPs whose contents can be replaced on demand. Each RP can host different RMs implementing alternative hardware accelerators or peripherals. During reconfiguration, only the active RM in the targeted RP is replaced; the static subsystem (including the processor core and memories) remains active and unaffected. Each RM is implemented as a partial bitstream

² We design our system so that every custom instruction invocation is functionally equivalent to its software fallback, including architecturally visible register effects and control-flow updates; only timing and other non-functional properties (e.g., energy consumption, electromagnetic characteristics) may differ.

for exactly that RP and device family. Vendor tooling (e.g., AMD Vivado [8]) is typically used to generate these partial bitstreams and to enforce consistent RP boundaries and interfaces across RMs. On AMD FPGA platforms, DPR is referred to as DFX and is typically driven via on-chip configuration ports, such as the Internal Configuration Access Port (ICAP), and interfaced by dedicated controllers, such as the DFX Controller [9]. The controller streams partial bitstreams from external or on-chip memory into the configuration fabric, asserts a decouple signal to isolate the RP during reconfiguration, and emits a reset once the new RM is in place. This work adopts this model and uses the DFX Controller as a standard component in the MCU system, rather than relying on an external host to drive reconfiguration. The achievable reconfiguration throughput, and thus the one-time cost of instantiating an RM, is ultimately bounded by the device’s configuration organization and programming mechanisms (e.g., on Artix-7 [10] class devices).

In a runtime-reconfigurable system, reconfiguration cost³ matters because it is paid *before* an RM can execute and therefore must be amortized over subsequent invocations. The effective cost depends on (i) the partial bitstream size, (ii) the bandwidth of the configuration port and the data path feeding it, and (iii) software overhead in triggering and streaming the bitstream. Consequently, many designs treat reconfiguration as a first-class operation, scheduled and amortized across repeated accelerator invocations, rather than as an infrequent, offline activity.

Surveys provide a broader context on DPR architectures, design flows, and application domains (e.g., Vipin and Fahmy [11]).

RISC-V Custom Instructions and Extension Interfaces. The RISC-V ISA [1] explicitly reserves opcode space⁴ for non-standard, custom instructions. In particular, the `custom-*` opcodes allow designers to introduce application-specific operations without breaking compatibility with the base ISA. A custom instruction behaves like any other instruction from the programmer’s perspective, but it can be decoded into an interface that drives tightly or loosely coupled accelerators. In this paper, accelerators are exposed as ISAXs: they are invoked like regular instructions, with operands provided via architectural registers and results written back to registers. Compared to memory-mapped peripherals, this style avoids explicit load/store sequences for short operand/result exchanges and aligns naturally with the compiler and register-based computation conventions. A key implication of using ISAXs in a DPR system is that instruction availability becomes dynamic: software may execute an instruction while its corresponding RM is currently not instantiated. Existing systems address this either by (i) ignoring or redirecting the instruction to a software sequence, or (ii) triggering an exception/handler that performs a fallback routine. Our design adopts this general idea and extends it with explicit support for preserving architectural state and data continuity across the hardware/software boundary (detailed later in Section 4), aiming for stable ISA-level behavior even as accelerators appear and disappear at runtime.

Several ISAX integration interfaces exist in practice, differing in coupling style, timing, and memory access capabilities. Examples include Rocket’s RoCC coprocessor interface [12], coprocessor ports in PULP-based cores [13], and the eXtension Interface (XIF) [14] in modern open RISC-V cores. These interfaces decouple the in-order pipeline from the accelerator while supporting handshakes, backpressure, and

optional side effects. They differ in timing guarantees, visibility into the pipeline, and how they expose memory access.

SCAIE-V [4], a Scalable Interface for ISA Extensions for RISC-V, occupies a complementary position as a Hardware Description Language (HDL) generator and interface standard for RISC-V custom instructions. It generates a SCAIE-V Abstraction Layer (SCAL) and a set of well-defined signals per ISAXs, including register operands, program counter access, pipeline stall/flush control, and instruction validity. Importantly, it abstracts away core-specific details and supports multiple back-ends such as PicoRV32 [5] and Orca [6]. In this paper, SCAIE-V provides the glue between the processor pipeline, the runtime-reconfigurable accelerators in the RPs, and the control modules that manage fallback and state handover.

Custom Instructions and their Software Fallback. Custom instructions executed by a hardware accelerator must interact cleanly with the architectural state: registers, program counter, and potentially an accelerator-local state that spans multiple invocations. A common strategy is to treat accelerators as optional functional units and use a software fallback whenever a custom instruction cannot be executed in hardware. This can happen because the corresponding RM is absent, being reconfigured, or simply not supported on a particular MCU variant.

Our prior work (e.g., *SmartOS* [15] and *moreMCU* [16]) has shown that such fallbacks can be implemented by raising an illegal-instruction exception and letting the operating system emulate the instruction in software. However, this approach exposes the presence or absence of the accelerator at the software level and does not directly address the seamless transfer of accelerator-local state between software and hardware.

In contrast, the architecture presented in this paper pushes these responsibilities into hardware and treats the presence of an accelerator as an implementation detail rather than a software-visible feature. We use SCAIE-V to provide two ISAXs: one (`custom-2`) that invokes the runtime-reconfigurable accelerators and another (`custom-3`) that provides a command interface to configure and monitor the reconfiguration subsystem. Around these ISAXs, we design (i) a hardware-managed software fallback path that preserves architectural transparency and (ii) a *Memory Handover Manager* that externalizes accelerator-local states such that even *stateful* accelerators can be evicted between invocations, emulated in software, and later resumed in hardware after DPR. The goal is that each custom instruction preserves the same architectural effects regardless of whether it is executed by hardware or by a software fallback.

The platform enforces architectural transparency of the transition by capturing/restoring the architectural state and control-flow effects; functional equivalence between an accelerator and its fallback is a design-time requirement that can be validated with the same test cases employed for the accelerator and its software implementation.

With these concepts in place, Section 3 reviews representative systems along the axes of (i) fabric placement (full FPGA vs. eFPGA), (ii) accelerator exposure (memory-mapped vs. ISAXs), and (iii) runtime management (controller- vs. Operating System (OS)-driven), and positions our design choices in that landscape.

3. Related work

Runtime-reconfigurable acceleration sits at the intersection of three recurring design questions, mirroring the axes introduced in Section 2: (i) *where* reconfigurable fabric is placed (embedded in an Application-specific Integrated Circuit (ASIC) as an eFPGA vs. a full FPGA hosting the complete SoC), (ii) *how* accelerators are exposed to software (memory-mapped peripherals vs. custom instructions), and (iii) *who* decides about reconfiguration (a dedicated controller, the OS, or a

³ We focus on time and latency effects in this paper; we acknowledge the importance of energy and endurance aspects and leave them for future work.

⁴ Custom opcode space is limited and must be managed at system-integration time. In our concept, we reserve `custom-2` for accelerator invocations and `custom-3` for management/monitoring. To expose multiple *logical* operations within a single opcode, we further encode the requested accelerator in `funct3` and use `funct7` to select sub-operations/variants within that accelerator.

mix of both). The following overview discusses representative designs across this space and motivates the choices made in this work.

Embedded FPGAs for tight coupling. A first line of work integrates eFPGAs into an ASIC to retain post-fabrication programmability while targeting microcontroller-class power and area. FlexBex [17] modifies the IBEX RISC-V core [18] by adding tightly coupled eFPGAs that serve as a configurable execution unit for custom instructions. As a case study, it demonstrates how the open-source FABulous [19] framework can generate eFPGAs fabrics from a parameterization table, enabling rapid exploration of fabric organization and tool flow. Arnold [20] follows a complementary design point: an eFPGA-augmented RISC-V MCU aimed at flexible, energy-efficient IoT end nodes, showing how an eFPGA can be used not only for compute offload but also for custom peripheral interfacing and streaming-style pre-processing close to I/O. Greyhound [21] represents a recent and *fully open-source* ASIC implementation of a tightly coupled eFPGA-based RISC-V SoC, emphasizing reproducibility across the entire stack (RTL, toolchain, and physical design). It combines a CV32E40X-based [22] SoC with on-chip SRAM and QSPI-based external memory access, and integrates a FABulous-derived eFPGA that can serve as either a custom-instruction extension or a custom peripheral. Notably, Greyhound also documents multiple in-system configuration paths (e.g., SPI-based and CPU-driven) that enable runtime retargeting of the embedded fabric without requiring a full-system re-synthesis. Neumann et al. [23] describe an even more direct coupling style by connecting an eFPGA to the processor register file, reusing general-purpose registers as operand and result channels (two 32-bit inputs and two 32-bit outputs), which avoids a separate peripheral bus path at the cost of a more constrained interface.

Full-FPGA SoCs and memory-mapped reconfigurable regions. A second, widely used category implements the complete system on a full FPGA and dedicates one or more RPs to the runtime exchange of RMs via DPR. In these systems, a static subsystem (processor, memory, interconnect, and control) remains active while accelerators are swapped in and out on demand. Many designs follow a memory-mapped access model, in which accelerators appear as peripherals and software controls both invocations and (re) configurations. Lesniak et al. [24] extend a LEON3 core [25] with RPs and high-bandwidth data movement (two 128-bit load-store units interconnected via a 128-bit data bus). To support larger accelerators than a single RM can handle, they partition complex functionality across cooperating RMs and orchestrate them using a microprogram initiated by a custom instruction and control data flow while stalling the processor pipeline. Monopoli et al. [26] focus on adaptivity of a GPU-like overlay architecture in a RISC-V SoC: the number of compute units is changed via DPR, and the work studies the trade-off between reconfiguration cost and compute speedup for a representative kernel. To characterize the reconfiguration side, they compare several controller/port combinations, including AMD's HWICAP [27], PCAP [28], the vendor DFX Controller [9], and the open-source ZyCap [29] with respect to reconfiguration time, reconfiguration throughput, and partial bitstream size. In addition to ZyCap, earlier work introduced a high-speed, open-source partial reconfiguration controller [30] and an automated design flow for adaptive systems on Zynq using CoPR [31]. Finally, El-Araby et al. [32] discuss how partial reconfiguration overhead affects accelerator task profiles, and Liu et al. [33] analyze the energy-efficiency trade-offs of systems that employ runtime partial reconfiguration. Another RISC-V-based SoC with memory-mapped RPs populated by RMs featuring different hardware accelerators is used to evaluate RV-CAP [34], a DPR controller. In contrast to vendor-specific solutions, this line of work emphasizes a software-managed programming flow with a wide Advanced eXtensible Interface (AXI) data path and streaming-based partial-bitstream delivery.

Predictability and runtime management. Besides average-case speedups, DPR impacts system predictability because reconfiguration

introduces a non-negligible, data-movement-heavy phase that can interfere with other traffic. In addition, mode changes between hardware acceleration and software fallback introduce execution-time variability that must be accounted for. Pezzarossa et al. [35] study this in a real-time context by integrating reconfigurable accelerators into a Patmos [36,37] system and analyzing the impact on Worst Case Execution Time (WCET) when accelerators are time-multiplexed using DPR. Their results highlight that DPR can be compatible with real-time analysis when the reconfiguration cost is explicitly modeled, and the platform is designed for analyzable memory and I/O behavior. A complementary approach is provided by Adetomi et al. [38], who introduce R3TOS and propose *slotless* reconfiguration: rather than defining fixed RPs sized for the largest RM, the placement, size, and boundary are chosen at runtime to reduce internal fragmentation. Accelerators are still memory-mapped, but hardware-supported semaphores coordinate allocation, activation, and safe replacement.

Custom instructions, fallbacks, and OS-level integration. To reduce invocation overhead and to integrate accelerators into the programming model, several systems expose reconfigurable functionality as custom instructions (ISAXs). In RISC-V, this naturally leverages the custom opcode space defined by the ISA specification [1]. With eMIPS, Pittman et al. [39] introduce a modified RISC processor that inserts custom instructions in front of frequently executed instruction sequences. If the corresponding RM is present, the custom instruction skips the original code block; if it is absent, the custom instruction is ignored, and the original software sequence runs unchanged. The OS uses execution profile information to decide which RMs to install and triggers DPR at program start. More recent RISC-V-based concepts build similar mechanisms around open cores, e.g., a RI5CY-based [40] MCU [41] that registers custom instructions at the OS layer [15] along with a software fallback routine. If the accelerator is present in the pipeline, the instruction executes like a regular ISA operation; otherwise, an illegal-instruction exception transfers control to the fallback function. In that work, core-side instruction logic (instruction integration, decoder hooks, and the illegal-instruction path) is updated via partial reconfiguration under control of an external host, i.e., reconfiguration is not driven purely in-system. Building on this direction, the *moreMCU* [16] combines a CV32E40P [42] core with *SmartOS* [15] and leverages DPR to time-share both custom-instruction accelerators and custom memory-mapped peripherals. *SmartOS* decides when to reconfigure based on configurable metrics (e.g., execution counts) and delegates partial bitstream programming to a controller using the ICAP mechanism.

Taken together, the works above demonstrate multiple points in the design space of reconfigurable acceleration; however, comparatively few explicitly address the *system-level semantics* of custom instructions under runtime reconfiguration, in particular, transparent execution when accelerators are absent and seamless transfer of accelerator-local state across hardware and software.

Positioning and Distinction. In contrast to the works discussed above, this work proposes a full-FPGA implementation that (i) targets different RISC-V soft processors, (ii) uses DPR to time-share scarce fabric resources among multiple accelerators, and (iii) exposes accelerators as custom instructions using the custom-2 opcode. Building on our prior work on OS-registered ISAXs with software fallback [15,41], the DPR-enabled *moreMCU* runtime [16], and the Greyhound ASIC/eFPGA SoC [21], and extending our paper [3], this work implements a software fallback path that preserves ISA-level semantics, assuming functional equivalence between accelerator and fallback, even when accelerators are absent: when a custom instruction is executed while its accelerator is absent, a dedicated hardware module triggers a context switch, buffers the input operands, invokes the corresponding software fallback routine, and finally writes back the results while restoring the architectural state, so that the execution appears as if the hardware accelerator had been present. During this transition, the pipeline is

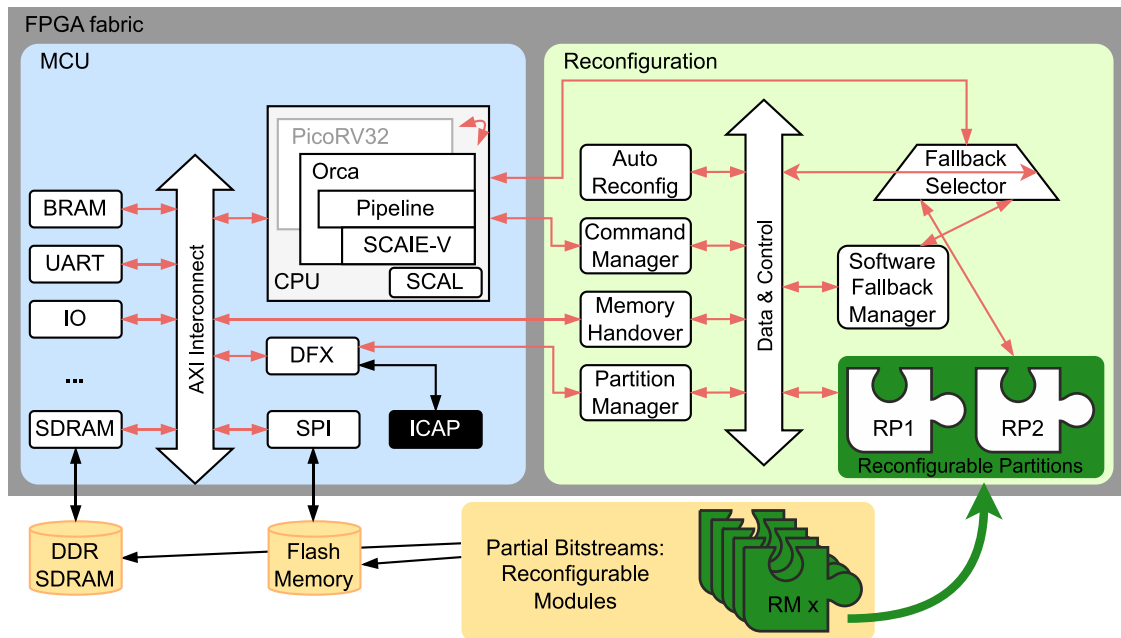


Fig. 1. System overview of the runtime-reconfigurable MCU. The MCU subsystem (core, memories/interconnect, peripherals, DFX controller) interfaces with the reconfiguration subsystem (Partition Manager, fallback path, Memory Handover Manager, RPs hosting RMs). Arrows indicate control/data interactions.

controlled via SCAIE-V stall/flush signals so that younger in-flight instructions do not observe partial effects and the custom instruction can be safely replayed for writeback. The entire functionality is realized without OS support – contrary to our prior work.

Compared with our earlier paper [3], the present article substantially extends the technical depth and scope: it makes the ISA-level execution semantics explicit, formalizes the miss path and instruction replay behavior, introduces and details the memory handover mechanism for stateful accelerators, broadens the evaluation from a single proof-of-concept style demonstration to two complementary case studies across two processor microarchitectures, and adds a clearer discussion of design trade-offs and applicability.

4. System architecture and runtime reconfiguration

This section presents the hardware/software realization of an MCU platform that (i) invokes hardware accelerators as ISAXs and (ii) time-shares hardware resources via DPR. A key requirement is correct execution even when an accelerator is temporarily unavailable (e.g., during reconfiguration): in that case, the corresponding instruction is transparently emulated by a software fallback while preserving architectural semantics. For stateful accelerators, the platform also supports state migration, enabling execution to continue seamlessly across changes in accelerator availability.

Fig. 1 depicts the implementation: the design is split into an *MCU* subsystem (including the RISC-V processor core, memories, interconnect, peripherals, and the DFX controller) and a *Reconfiguration* subsystem (including the RPs and the management and control logic). This separation also mirrors the implementation: the MCU subsystem is assembled as a Vivado block design using mainly vendor-provided AMD IP blocks (e.g., AXI interconnect, DFX controller), whereas the reconfiguration subsystem and the cores are implemented as custom (System)Verilog modules.

Execution Model at a Glance. Each custom instruction call is either a *hardware hit* that executes within an RM or a *hardware miss* that is transparently emulated in software, while preserving ISA-level semantics, *assuming functional equivalence between accelerator and fallback*. For stateful accelerators, continuity of accelerator-local state is maintained by stalling invocations during state handover and by transferring state

between the RM and a memory-resident state image. An RM that is being (re)configured is treated as *absent*.

- (1) **Hit (RM present):** the request is routed to the RP hosting the selected RM; the RM may stall and/or redirect the control flow via SCAIE-V, and returns the result accordingly.
- (2) **Miss (RM absent):** the request is routed to the hardware-managed fallback path. The fallback mechanism captures the instruction context, performs a controlled context switch into a C/C++ fallback function, and restores the architectural state so that the execution continues as if the instruction had executed in hardware.

Upon a miss, a runtime policy may trigger DPR to instantiate the requested RM for subsequent hits; correctness is still provided immediately by the miss path while DPR prepares later hits, so the core does not block on reconfiguration completion to make forward progress. To avoid reconfiguration-induced deadlocks, misses always execute via the fallback path while DPR proceeds asynchronously; at worst, shared-memory contention causes slowdown, not loss of forward progress. For stateful accelerators, (de)instantiation is accompanied by a state handover procedure that transfers accelerator-local state between RM-internal registers and a memory-resident state image shared with the fallback implementation. While such a handover is in progress, invocations targeting the affected RM are stalled to maintain state consistency.

4.1. Microcontroller unit

A core requirement of our platform is support for ISAXs that may stall the core (i.e., delay the retirement of younger instructions) and, if needed, redirect the control flow (e.g., loop-like constructs). We therefore build on SCAIE-V, which generates a core-specific integration layer (SCAL) and exposes a structured, customizable interface for custom instructions.

SCAIE-V supports multiple processors, including PicoRV32 [5], a non-pipelined RISC-V processor, and Orca [6], a pipelined RISC-V processor with 4 or 5 stages (depending on its configuration). PicoRV32 uses a single AXI 4 Lite [43] interface to fetch instructions and access

Table 1

SCAIE-V timing configuration used for `custom-2` (accelerator) and `custom-3` (command) on PicoRV32 and Orca. Numbers denote pipeline stages (Orca) or cycle-aligned phases (PicoRV32); “–” indicates signals not used for `custom-3`.

Operation	SCAIE-V interface pipeline stages			
	custom-2		custom-3	
	PicoRV32	Orca	PicoRV32	Orca
RdRS1, RdRS2	2	3	2	3
WrRD	3	4	3	4
RdPC	2	3	2	3
WrPC	3	4	3	4
RdInstr	2, 3	3, 4	2, 3	3, 4
RdIVValid	2, 3	3, 4	2, 3	3, 4
RdStall	2, 3	3, 4	2, 3	3, 4
WrStall	2	3	2	3
RdFlush	2	3	–	–
WrFlush	2	3	–	–

the data memory, while Orca has been configured to use two separate interfaces for instruction and data memory. We configure Orca with five pipeline stages, no caches for both the instruction and data memories, and a branch target buffer with 16 entries. We limit Orca to at most one pending instruction fetch.

To integrate our concept, both processors are extended by two new R-type instructions. The first instruction with the `custom-2` opcode executes all hardware accelerators. In contrast, the second one with the `custom-3` opcode serves as an access point to the *Command Manager*, an interface for the modules in the reconfiguration part. The SCAIE-V parameters defining the interface for the two ISAXs are shown in [Table 1](#). Both interfaces are forwarded and exposed to the RPs and the custom modules that support them.

The processor’s instruction/data memory interfaces connect to an AXI interconnect, which serves as the central hub of the MCU memory system (star topology). The bus connects to two AXI BRAM modules: the first (32 kB) serves as a combined program and data memory, allowing the linker to apply the defined memory layout. The second (8 kB) stores the states of absent hardware accelerators (cf. [Section 4.2.6](#)); the size is chosen to comfortably hold the state images of our prototype accelerators and can be scaled with the number and size of supported states. The AXI UART and General-Purpose Input/Outputs (GPIOs) modules provide inputs and outputs for the test cases. The AXI Double Data Rate (DDR) Synchronous Dynamic Random-Access Memory (SDRAM) and the Serial Peripheral Interface (SPI) memory modules provide access to the two external memory chips on the Nexys 4 DDR Development Board [44] that are used to store the partial bitstreams of the RMs.

The second master on this bus is an instance of the AMD DFX Controller [9], which is responsible for the system’s DPR. Once a DPR has been requested via one of the hardware triggers, the DFX controller sets the RP decouple signal to high. Subsequently, it fetches the corresponding partial bitstream from an address within the AXI-connected memory and forwards it to the ICAP. Once the DPR is complete, the decouple signal is set to low, and a reset signal is emitted to reset the logic within the newly reconfigured RP. The *Partition Manager* tracks the states of all RPs using the decouple and reset signals of the DFX controller.

4.2. Runtime-reconfigurable hardware accelerators

The *Reconfiguration* part (cf. [Fig. 1](#)) contains the interconnected modules that support runtime-reconfigurable hardware accelerators. To keep the narrative consistent with the execution model, we describe the modules in the order they participate in handling a custom instruction invocation: (i) hardware accelerator selection and dispatch, (ii) DPR control and partition state tracking, (iii) miss-path software fallback, and (iv) state handover and on-demand instantiation.

Table 2

Partial bitstream size and memory map used by the DFX Controller for each RM in RP1/RP2 (bytes, base addresses).

RM	Partial bitstream size	RP 1	RP 2
		360 011 bytes	359 203 bytes
	Partial bitstream address		
RM 1	0x8000_0000	0x8030_0000	
RM 2	0x8006_0000	0x8036_0000	
RM 3	0x800C_0000	0x803C_0000	
RM 4	0x8012_0000	0x8042_0000	
RM 5	0x8018_0000	0x8048_0000	
RM 6	0x801E_0000	0x804E_0000	
RM 7	0x8024_0000	0x8054_0000	
RM 8	0x802A_0000	0x805A_0000	

Table 3

Resource budget per RP on the XC7A100T. All RMs must fit within this RP envelope (LUTs, Registers, BRAMs, DSPs).

RP	LUT	Register	BRAM	DSP
RP 1	1600	3200	10	20
RP 2	1600	3200	10	20

4.2.1. Hardware accelerators, reconfigurable modules, and partitions

To demonstrate the capabilities of our generic concept, we created a system comprising two RPs and eight RMs. The approach itself is not restricted to this number of partitions, since the corresponding management logic is largely generic. However, partition sizing is a design-time use-case decision and is therefore fixed in the present implementation rather than treated as an evaluation parameter. A fixed-size RP may nevertheless be larger than required by some compatible RMs, leading to internal underutilization of partition resources, even though each RP hosts exactly one RM at a time and no runtime packing or splitting of RMs is performed. In this work, scalability is therefore reflected mainly in the relation between the active accelerator working set and the number of available RPs: if more accelerators are needed than can be resident simultaneously, absent ones are handled by the fallback path until reconfiguration makes later invocations hit again.

The position and size of each partial bitstream are configured in the DFX controller. [Table 2](#) documents the partial bitstream sizes and the memory addresses used in our prototype system configuration.

The two RPs are designed for a generic use case and contain the numbers of LUTs, registers, BRAMs, and DSPs, as listed in [Table 3](#). Each RP can be populated with one supported hardware accelerator at a time. Each accelerator is implemented as an RM that is compatible with the fixed RP interface (including decouple/reset behavior and the state-handover signals described in [Section 4.2.6](#)). The RMs are accessed using an R-type instruction with the `custom-2` opcode, with two 32-bit input registers and one 32-bit output register. The three bits of the R-type instruction’s `funct3` field identify which RM is requested, while the seven `funct7` bits are forwarded to the RM as additional information. If necessary, an RM can stall the processor and change the program flow by modifying the Program Counter (PC) and flushing the pipeline. Furthermore, the RMs contain an interface used by the *Memory Handover Manager*.

The RPs and RMs are realized using a block design container for DFX. They are implemented as a three-layered system, with the outer layer being the block design container. The middle layer consists of a Verilog module, as SystemVerilog modules are not supported by the design container for DFX. The innermost layer is a SystemVerilog module that is dynamically exchanged depending on the (test) application being executed.

4.2.2. Partition manager

The *Partition Manager* coordinates DPR requests, tracks RP occupancy, and provides the presence/availability information required for correct dispatch. It interacts with the DFX Controller, triggers the DPR, and keeps track of the following per-RP states:

- (i) **Present:** the RP hosts an active, usable RM.
- (ii) **Cleanup:** the *Memory Handover Manager* transfers the RM state to memory prior to reconfiguring the RP.
- (iii) **Empty:** the RP is empty and ready for reconfiguration.
- (iv) **Trigger:** the *Partition Manager* issues a reconfiguration request to the DFX controller.
- (v) **Reconfiguration:** DPR is in progress for this RP.
- (vi) **Waiting:** the DFX controller is busy (only one RP can be reconfigured at a time).
- (vii) **Prepare:** the *Memory Handover Manager* restores the RM state from memory after reconfiguration.

The *Partition Manager* also ensures that any distinct RM populates only one RP at a time. It also exposes its status via the *Command Manager* (Section 4.2.5) for debugging⁵ and software-controlled experiments.

4.2.3. Fallback selector

Based on the RP's occupancy information provided by the *Partition Manager*, the *Fallback Selector* routes each custom-2 invocation either to the RP currently hosting the requested RM (hit path) or to the *Software Fallback Manager* if that RM is not present (miss path). For stateful accelerators, it also stalls the dispatch while a handover operation is in progress, ensuring a consistent state between the RM and its software fallback implementation.

4.2.4. Software fallback manager

DPR enables time-sharing of a fixed pool of logic resources among multiple RMs; however, each RP can only host one RM at a time. The *Software Fallback Manager* implements the miss path: if the requested RM is absent, it orchestrates a controlled transition into a corresponding software implementation of the hardware accelerator, computing the same results (possibly with different non-functional behavior). This assumes functional equivalence at the architectural interface: for a given custom instruction, the RM and its software fallback must produce the same architecturally visible effects for the same input state, including register writeback and any PC/control-flow effect. This does not require identical microarchitectural behavior, latency, energy consumption, or side-channel behavior; it only requires architectural correctness. In practice, both implementations can be validated against the same functional test vectors and reference outputs. The fallback process is function-based, with one C/C++ function (the *Software Fallback Function*) per RM providing an architecturally equivalent implementation of the corresponding custom instruction.

Terminology. We refer to the per-accelerator C/C++ code as the *fallback function/implementation*, the overall mechanism as the *fallback path*, and the hardware module that orchestrates it as the *Software Fallback Manager*.

The software fallback process can be divided into three parts: **Enter**, **Execute**, and **Exit**, while modifying the processor state so that the execution continues *as if* the missing instruction had executed in hardware. Fig. 2 depicts the sequence of the subsequently explained steps:

- (a) The *Software Fallback Manager* buffers the execution parameters of the RM, including the input registers, the PC, the `funct3` field identifying the absent RM, and the `funct7` field from the R-type instruction triggering the fallback.
- (b) The *Software Fallback Manager* triggers a PC update via the SCAIE-V interface, allowing the processor to enter a small assembly stub located at a fixed memory address.⁶

⁵ Furthermore, as an optional debug aid, the current RMs and the state are shown on the FPGA board's 7-segment display.

⁶ In our prototype we place this entry stub at `0x18` to keep the linker layout simple in a small on-chip memory configuration.

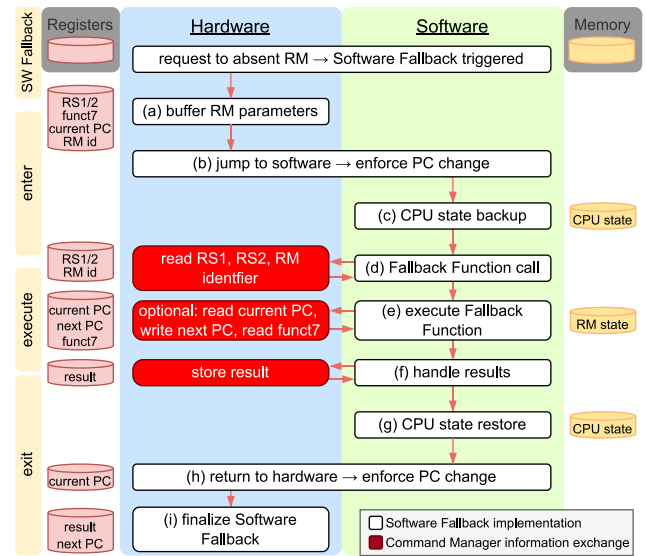


Fig. 2. Hardware-managed software fallback sequence for a custom instruction miss. Steps (a)–(i) correspond to Section 4.2.4: context capture, controlled jump to the fallback stub, C/C++ fallback execution, state restore, and instruction replay with optional PC override via the *Command Manager*.

- (c) The compiler expects the software fallback to behave as if executing an arbitrary R-type instruction. As it also manages register utilization within a function (cf. RISC-V specification [1]), registers the *Software Fallback Function* may clobber must be backed up (via a context switch). Following the calling convention, we back up caller-saved registers as well as registers without a stable role (e.g., global pointer, thread pointer). The stack pointer is also backed up and repositioned, creating a fresh stack for the fallback execution.
- (d) The identifier of the absent RM and its buffered input registers are fetched using the *Command Manager* and used to call the corresponding *Software Fallback Function* following the function calling convention. The return address points to an assembly epilogue handling the exit.
- (e) The *Software Fallback Function* computes the instruction result in C/C++ with two 32-bit inputs, `rs1` and `rs2`, and one 32-bit output, `rd`. The software includes helper functions to access `funct7`, the original PC, and to request a PC override (to emulate accelerator-induced control-flow changes) via the *Command Manager*.
- (f) The function returns the computed result according to the calling convention; the value is forwarded to the *Software Fallback Manager* via the *Command Manager*.
- (g) The processor state is restored using the backed-up values.
- (h) Now that the result is buffered in a hardware register and the processor state is updated, the *Command Manager* enforces a PC change via the SCAIE-V interface, returning from the software part of the fallback path.
- (i) To write the result into the architecturally correct destination register (as encoded in the original custom instruction), we resume at the faulting instruction and use an *instruction replay* scheme: the pipeline re-observes the same instruction fields, while the fallback path injects the computed result and optional next PC override into the corresponding SCAIE-V writeback/control signals.

From a cost perspective, the software-resident steps contribute an instruction-count overhead, while the hardware-controlled steps contribute to miss latency in cycles (e.g., due to pipeline control and

Table 4

Dynamic instruction overhead of the software-resident fallback prologue/epilogue (steps (c)–(h) in Fig. 2). Hardware-controlled steps (a), (b), and (i) add cycle latency but no executed software instructions.

Step		# Ins.
(c)	CPU state backup	25
(d)	Software Fallback Function call	10
(f)	handle the Software Fallback Function result	1
(g)	CPU state restore	22
(h)	return to hardware → enforce a PC change	1
Σ		59

handshakes). Table 4 reports the instruction-count overhead of the *software-resident* parts (steps (c)–(h)). Steps (a), (b), and (i) are orchestrated by hardware control and do not execute additional *software* instructions. However, they still contribute to the overall miss latency measured in cycles in Section 6.

Pipeline behavior during a miss is intentionally localized to the faulting custom instruction. The processor does not globally stall for the DPR duration. Instead, when an absent RM is detected, the fallback path captures the instruction context and uses the SCAIE-V control signals to prevent younger in-flight instructions from observing partial effects. The custom instruction is then completed via software fallback and replayed at the architectural interface for correct write-back and optional PC update. DPR proceeds asynchronously in the background; subsequent custom instruction invocations may still miss and use the fallback path until the requested RM becomes present. Thus, fallback introduces per-miss latency, whereas DPR does not block forward progress of independent instructions beyond shared-memory interference.

4.2.5. Command manager

The *Command Manager* exposes a small set of configuration/status registers using an R-type instruction with the `custom-3` opcode.⁷ The `funct7` field acts as an identifier for the configuration/status to be accessed, and the instruction's `rs1`, `rs2`, and `rd` fields are used as inputs and outputs, respectively. Besides general observability and control, the *Command Manager* is a key building block for semantic transparency: it provides the fallback stubs with access to buffered parameters (`funct3`, `funct7`, PC, operands), enables the fallback to request a PC override (to emulate accelerator-induced control-flow changes), and forwards the computed result back to hardware control. Additionally, it provides access to the *Partition Manager* (to read RP states and active RMs), supports DPR requests, and enables controlled experiments by temporarily overriding whether dispatch proceeds to an RP or to the fallback path, independent of the RM's presence.

4.2.6. Memory handover manager

Stateful accelerators require more than architectural register consistency: they also maintain *accelerator-local state* across invocations (e.g., buffered intermediate values), or the internal state of a cryptographic primitive. This state is not part of the processor register file or Control and Status Register (CSR) state, yet it influences future results of the corresponding custom instruction. If such an accelerator is evicted, replaced, or temporarily served by software, that *accelerator-local state* must remain consistent across the hardware/software boundary. The purpose of the *Memory Handover Manager* is therefore not general processor-state migration, but the preservation of *accelerator-local execution context* between a hardware RM and its software fallback. This module ensures consistent state transfer

⁷ The effort required to create an easy-to-use interface using SCAIE-V seemed smaller than creating a separate memory-mapped interface, thanks to the simplicity of the generator framework.

between hardware accelerators and their software fallback implementations. For stateful accelerators, the instruction-level consistency must be preserved when execution migrates across the hardware/software boundary. To this end, the accelerator-local state is transitioned to a memory-resident state image whenever an accelerator is removed, and restored when it is reinstated. For evolving hardware, where the internal logic changes between two configurations, transformation of the state representation might be needed. Hardware RMs use internal registers or other memory blocks to store their current state, which can be updated on each execution and influence subsequent results. If the RM is absent, the *Software Fallback Function* executes instead and accesses the state image in memory. When an RP is reconfigured, the *Memory Handover Manager* transfers the state between RM-local registers and memory via its AXI connection. The state is moved using a customizable number of 32-bit words and a configurable starting memory address, incremented in 4-byte steps, as shown in the flowchart in Fig. 3. For the *Software Fallback Functions*, the memory layout is defined as a 4-byte-aligned structure so that each state word maps directly to the addresses used by the handover procedure.

On the RM side, the handover interface contains the relevant signals for state transfer. To minimize static-to-RP wiring, we use SCAIE-V `rs1` and `rd` signals to send/receive 32-bit state words. The `memory_cnt` signal indicates which word is being transferred, and `memory_read/memory_write` indicate the direction.

Valid state rule. The accelerator-local state has two physical representations (RM-local storage vs. a memory-resident state image), but exactly one representation is *valid* at any time. When the RM is present, the RM-local state is *valid* and software must not modify the memory-resident image except during an explicit handover. When the RM is absent, the memory-resident image is *valid* and is accessed only by the corresponding *Software Fallback Function*. The *Fallback Selector* stalls the processor *only* if the `custom-2` request occurs that targets the RM *while* the state is being transferred, so that no invocation can operate on a partially transferred state. Furthermore, the *Memory Handover Manager* waits to transfer the state until the *Software Fallback Function* has finished and does not initiate a handover while the RM is executing.

4.2.7. Automatic reconfiguration module

The *Automatic Reconfiguration Module* triggers the *Partition Manager* to perform DPR whenever an absent RM is requested. In our prototype, it populates the least recently used RP with the requested RM. The *Command Manager* provides configuration hooks for this module (enable/disable and overriding the next RP to be reconfigured). Importantly, the policy does not replace the miss-path: correctness is always provided immediately by the fallback path, while DPR is used to make *subsequent* invocations hit again.

5. Experimental setup and methodology

This section describes (i) the experimental platform and tool flow, and (ii) the measurement methodology. All benchmark- and accelerator-specific interface details are provided alongside the corresponding results in Section 6 to keep the evaluation narrative local.

5.1. Platform and tool flow

Our experiments use a Nexys 4 DDR development board [44] featuring an AMD XC7A100T [10] FPGA. All (partial) bitstreams are generated using the AMD Vivado 2023.2 tool flow [8]. The corresponding software is compiled using GCC (`-O0` optimization) and linked into the on-chip BRAM used as program/data memory of the MCU. Partial bitstreams for the different RMs are flashed onto the external SPI flash memory. The full bitstream, including the software binary, is then flashed onto the FPGA.

Once the flash is completed, the FPGA resets and the processor copies partial bitstreams from the SPI flash into DDR SDRAM, from

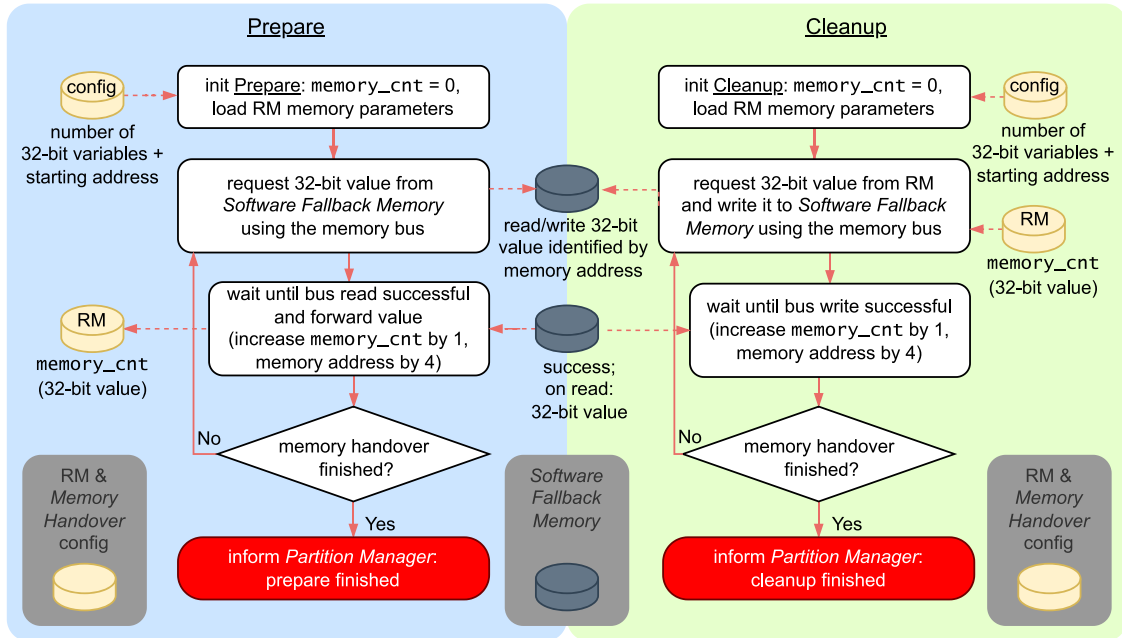


Fig. 3. Memory handover procedure for stateful accelerators. The *Memory Handover Manager* transfers an RM's internal state to/from a memory-resident state image (word-granular, 32-bit steps) during Cleanup/Prepare while dispatch is stalled to enforce the valid state rule.

which the DFX controller streams them via the AXI-connected ICAP module. This implies that DPR traffic and fallback execution can contend for shared memory/interconnect bandwidth in the on-demand experiments (Section 6.2). Table 2 documents the sizes and addresses used by the controller.

Unless stated otherwise, the system runs at a fixed clock frequency across all configurations (a single clock domain for the core, interconnect, and reconfiguration logic). Since we report cycle counts, the results are independent of the absolute frequency as long as it is held constant.

5.2. Measurement methodology

Execution time is measured in clock cycles using RDCYCLE. Unless stated otherwise, we report cycle counts for three execution modes: (i) regular software baseline, (ii) hit-path hardware execution with the RM present, and (iii) miss-path software execution via the fallback path when the RM is absent. For (iv) on-demand instantiation experiments, we additionally report a mixed mode in which fallback provides correctness while DPR runs in the background, and the RM takes over once it becomes available.

All measurements are taken on the same platform configuration per core as described in Section 4.1. In particular, Orca is evaluated without caches, a 16-entry branch target buffer, and at most one outstanding fetch. For stability, each benchmark run is executed under identical initial RPs occupancy conditions, and the results are collected from the same software binary linked into BRAM. The measured timings also reflect the chosen memory-system organization. PicoRV32 uses a single AXI4-Lite path for instruction fetch and data access, whereas Orca uses separate instruction and data interfaces without caches and with at most one pending fetch. Thus, the reported cycle counts capture not only accelerator and fallback costs, but also timing effects of the chosen platform integration. In the on-demand experiments, partial-bitstream transfers and fallback execution may share resources in principle, although the test cases are arranged to expose the relevant overheads directly.

6. Proof of concept and evaluation

Using the platform and methodology described in Section 5, we evaluate our concept using two complementary proof-of-concept accelerators that deliberately stress different aspects of *stable ISA semantics under changing hardware availability*:

- **ROL (control-flow, low compute intensity):** stresses program-flow changes (WrPC/WrFlush) with minimal arithmetic work. This benchmark is intentionally unfavorable for fallback overhead and therefore exposes the fixed cost of full ISA-level transparency.
- **ASCON (stateful, compute-intensive):** a realistic, lightweight cryptographic primitive with substantial computation and a non-trivial internal state, making it a natural beneficiary of hardware acceleration and a representative case for state handover.

We report cycle counts according to our measurement methodology presented in Section 5.2. In this context, two performance criteria were defined to analyze the effectiveness of embedding a given hardware accelerator in an RM:

The *Fallback Overhead*, defined in Eq. (1), indicates the relative overhead of a software fallback implementation compared to a regular software implementation, i.e., the relative increase in execution time when using the fallback instead of the software. Let t_{SW} denote the execution time of the regular software implementation and t_{FB} the execution time of the software fallback implementation.

$$Fallback\ Overhead = \frac{t_{FB}}{t_{SW}} - 1 \quad (1)$$

The *Hit Rate*_{1:1} indicates the percentage of executions where the RM has to be present and executed, in addition to the executions of the software fallback implementation, in order to provide, on average, an execution time similar to that of the software implementation (1:1). In our evaluation script, we approximate this threshold by considering a batch of $N = 10^6$ executions and forming the combined total time

$$T(i) = (N - i) t_{RM} + i \cdot t_{FB}, \quad (2)$$

with i fallback executions and $N - i$ RM hits. We then search for the largest fallback count i^* such that the combined total remains below the software baseline $N t_{SW}$,

$$i^* = \max \left\{ i \in \{1, \dots, N - 1\} \mid (N - i)t_{RM} + it_{FB} < N t_{SW} \right\}, \quad (3)$$

and report the corresponding $Hit Rate_{1:1}$ as the required fraction of RM executions:

$$Hit Rate_{1:1} = 1 - \frac{i^*}{N}. \quad (4)$$

The following Sections 6.1 and 6.2 detail the proof-of-concept designs and observations.

Listing 1: Example invocation of the ROL custom instruction pair via inline assembly. The two custom-2 instructions (distinguished by `funct7`) implement loop init and loop step/branch counting steps of 2 and accumulating into result.

```

1 register uint32_t rs1 asm ("s2") = 2;
  //define rs1 as s2 (x18); counter
  start value/increment
2 register uint32_t rs2 asm ("s3") = 10;
  //define rs2 as s3 (x19);
  condition < 10
3 register uint32_t rd asm ("s4") = 0;
  //define rd as s4 (x20); current
  counter value
4 uint32_t result = 5;
5 //initialize loop: rs1 = counter start
  value, rs2 = condition, R-type
  instruction with
6 //funct7=1, rs1 = s2, rs2 = s3, funct3
  = 4, rd = s4, opcode = custom-2
7 __asm__ __volatile__ (".word 0
  x03394A5B\n" ::: "x18", "x19", "x20
  "); //loop start
8 {
9     result = result + rd;
10 }
11 //update counter, compare it with end
  condition and jump to loop begin
  if necessary
12 //rs1 = counter increment, R-type
  instruction with
13 //funct7=2, rs1 = s2, rs2 = s3, funct3
  = 4, rd = s4, opcode = custom-2
14 __asm__ __volatile__ (".word 0
  x05394A5B\n" ::: "x18", "x20");

```

6.1. ROL case study: Control-flow extension

The ROL accelerator implements the structure of an incrementing for-loop using two custom instructions per loop (distinguished via `funct7`). The first instruction (placed before the loop body) initializes the loop state, counter value, and records the loop-body entry PC. The second instruction (placed after the loop body) updates the counter (using `rs1` as increment), evaluates the termination condition (`counter < rs2`), and redirects control flow back to the loop-body entry if another iteration is required. Overall, the ROL reduces the loop-control overhead to one instruction for initialization and one instruction per iteration.

The ROL benchmark counts from zero to $0x40$ while accumulating the counter's values, and the execution time (in clock cycles) is obtained using `RDCYCLE`. Listing 1 illustrates an ROL example that counts from 2 to 10 in increments of 2 while accumulating the counter's values in the variable `result`. The variables are mapped to registers via inline assembly, and the RISC-V instructions that invoke ROL are encoded as raw hexadecimal instruction words, simplifying debugging when analyzing internal logic signals.

Table 5

End-to-end execution time (in cycles) for ROL and ASCON on PicoRV32 and Orca, comparing Software baseline, RM hit (Hardware), and RM miss (Software Fallback). ASCON encrypts 48 bytes (12×32 -bit words) including tag. The derived metrics *Fallback Overhead* and *Hit Rate_{1:1}* are computed from these timings.

		PicoRV32	Orca
ROL: Execution time/# Clock cycles			
1	Software (FOR Loop)	7286	3897
2	Hardware (ROL)	4836	2665
3	Software Fallback (ROL)	101 271	46 353
ASCON: Execution time/# Clock cycles			
1	Software	126 017	66 191
2	Hardware	3515	1800
3	Software Fallback	151 521	78 133
ROL: Performance criteria/# Percentage			
1	<i>Fallback Overhead</i>	1289.93%	1089.45%
2	<i>Hit Rate_{1:1}</i>	97.46%	97.18%
ASCON: Performance criteria/# Percentage			
1	<i>Fallback Overhead</i>	20.23%	18.04%
2	<i>Hit Rate_{1:1}</i>	17.23%	15.64%

Table 5 and Fig. 4 present the execution times for ROL's hardware implementation and software fallback implementation, which are compared against those of a for loop performing the same task.

Why is ROL fallback much slower than regular software? ROL is intentionally a *low-compute*, instruction-style accelerator: when present, it replaces a short loop-control sequence with a single instruction for initialization and one per iteration. On a miss, however, each custom instruction invocation pays a largely fixed transparency cost: context save/restore, entering/exiting the fallback function, and coordinating architectural effects (register writeback and potential PC redirection) so that the instruction remains ISA-equivalent. In our prototype, the software-resident part of this fixed cost is 59 instructions per miss (Table 4), plus additional control-plane interaction via the *Command Manager* and pipeline-control latency, as seen in cycle measurements. Because the benchmark executes the ROL instructions frequently, this fixed cost repeats many times while the useful work per invocation is small, and thus dominates the total miss-path execution time. Concretely, each loop iteration triggers one ROL update instruction; on a miss, this adds ≈ 59 software instructions (Table 4) plus hardware control latency, which quickly outweighs the few-instruction loop-control baseline and explains the overhead in Table 5. This exposes the general rule of thumb for the proposed architecture: the fully transparent fallback path is best amortized when misses are rare (high hit rate) and/or when the accelerator performs enough work per invocation (high compute intensity), as in ASCON. In other words, ROL exposes the *price of full transparency* in the worst case: if the useful work per accelerated operation is small, the fallback overhead cannot be amortized. ASCON, in contrast, performs enough computation per operation that the fallback overhead is comparatively small and the hardware speedup dominates.

6.2. ASCON case study: Stateful cryptographic accelerator

While ROL is a custom-made example, ASCON – the international standard for lightweight cryptography – is a defined family of algorithms supporting Authenticated Encryption with Associated Data (AEAD). It operates internally on 64-bit data types, while the open-source AEAD-128 reference implementation exposes an interface based on byte arrays with length parameters. These characteristics cannot be directly supported by a 32-bit custom instruction interface that provides only two 32-bit inputs and one 32-bit output per instruction. Consequently, the open-source C implementation has been adapted to

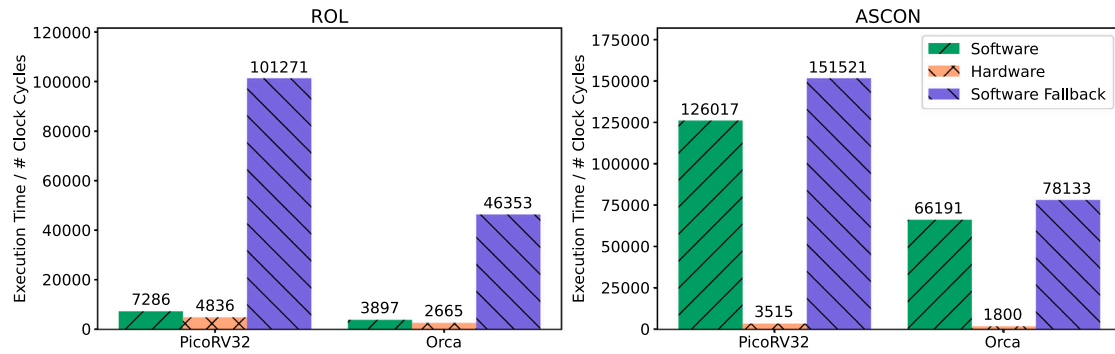


Fig. 4. Execution time comparison for ROL and ASCON on PicoRV32 and Orca (in cycles). ASCON encrypts 48 bytes (12×32 -bit words) including tag generation; bars correspond to Software baseline, RM hit (Hardware), and RM miss (Software Fallback).

an interface aligned with our generic concept and invoked via an R-type custom instruction. Based on ASCON's round-based algorithm (64-bit blocks, internal sponge state of five 64-bit words), the implementation has been restructured to store the key and sponge state as 32-bit unsigned integers and to expose the algorithm's stages through multiple function calls, each triggering a distinct step of the encryption process.

In the initialization phase, four distinct calls initialize the sponge with key and nonce, each transferring two 32-bit words, yielding 128-bit representations for both inputs. During associated data processing and plaintext encryption, 64-bit blocks are transferred to and from the accelerator across an arbitrary number of iterations. The final phase generates the authentication tag, producing 128 bits of output data.

Based on the adapted C implementation, the ASCON hardware accelerator has been realized as an RM. Similar to its software counterpart, it is invoked multiple times per encryption, and the `funct7` field of the custom instruction identifies the operation to be executed. The plaintext-to-ciphertext encryption function has been decomposed into two instructions due to the limitation that RISC-V R-type instructions support only two 32-bit input operands and one 32-bit output operand. The first instruction processes a 64-bit input block, updates the internal state, and buffers the second 32-bit output internally while returning the first. The second instruction indicates whether the current block is the final block and retrieves the buffered value. Similarly, the 128-bit authentication tag is obtained through four instruction executions.

Afterwards, the software implementation has been integrated into the corresponding *Software Fallback Function*. ASCON's internal state is represented as a struct, and the *Memory Handover Manager* is configured such that the state can be moved between the software fallback implementation and the hardware accelerator's internal registers whenever the accelerator is (de)instantiated (cf. the *Cleanup* and *Prepare* phases in Section 4.2.6), enabling seamless continuation across the hardware/-software boundary. Table 5 and Fig. 4 present the execution times for ASCON's hardware implementation and software fallback implementation, compared with those of a baseline software implementation performing the same task. We benchmark the encryption of 48 bytes of plaintext data, organized into 12 words, including authentication-tag generation.

Compared with the results in Section 6.1, this suggests a simple selection rule: instruction-style accelerators are attractive when the work per invocation is large enough to amortize the fallback's fixed transparency cost, or when misses are rare (high hit rate).

The data collected show that applications with high computation effort, such as ASCON, perform better. When the *Software Fallback Function* supports an application with low computation effort, such as ROL, the corresponding software fallback is more costly due to the relatively high effort of entering and leaving the *Software Fallback Function*. Emulating a program flow change originating in an absent RM is more costly than a regular program flow induced by emulating a PC change, i.e., interacting with the *Software Fallback Manager*. Furthermore, the faster RM (i.e., an algorithm with lower computational effort) must be more prevalent in the $Hit Rate_{1:1}$ to recover the clock cycles spent on the software fallback overhead.

Table 6

Per-step execution time (in cycles) for ASCON with automatic reconfiguration enabled (RM initially absent). The benchmark encrypts 124 bytes (31×32 -bit words, padded).

Step		Exec. Time/# Clock cycles	
		PicoRV32	Orca
1	Key	^a 3 032	^a 1 503
2	Nonce	29 787	15 570
3	Encrypt	16 410	8 545
4	Encrypt	16 432	8 497
5	Encrypt	16 442	8 495
6	Encrypt	16 442	8 503
7	Encrypt	16 412	8 506
8	Encrypt	^b 306	8 494
9	Encrypt	306	8 497
10	Encrypt	306	8 500
11	Encrypt	306	8 494
12	Encrypt	306	8 503
13	Encrypt	306	8 497
14	Encrypt	306	^b 291
15	Encrypt	306	156
16	Encrypt	306	156
17	Encrypt	306	156
18	Encrypt	305	162
19	Tag	233	129
Σ		118 555	111 654

^a Triggers DPRs.

^b Marks the first step executed using the ASCON RM after DPR completes.

Table 7

ASCON per-step baseline timings (in cycles) for Software, RM hit (Hardware), and RM miss (Software Fallback) on PicoRV32 and Orca.

Step		Exec. Time/# Cycles	
		PicoRV32	Orca
Regular	Software		
	Key	441	250
	Nonce	27 134	14 256
	Encrypt	13 848	7 268
	Encrypt (last) ^a	568	352
	Tag	27 609	14 536
Hardware	(Accelerator)		
	Key	203	114
	Nonce	219	122
	Encrypt	306	156
	Encrypt (last) ^a	305	162
	Tag	233	129
Software	Fallback		
	Key	3 025	1 495
	Nonce	29 718	15 473
	Encrypt	16 390	8 447
	Encrypt (last) ^a	3 110	1 502
	Tag	32 693	16 896

^a Only the last encrypt omits the permutation.

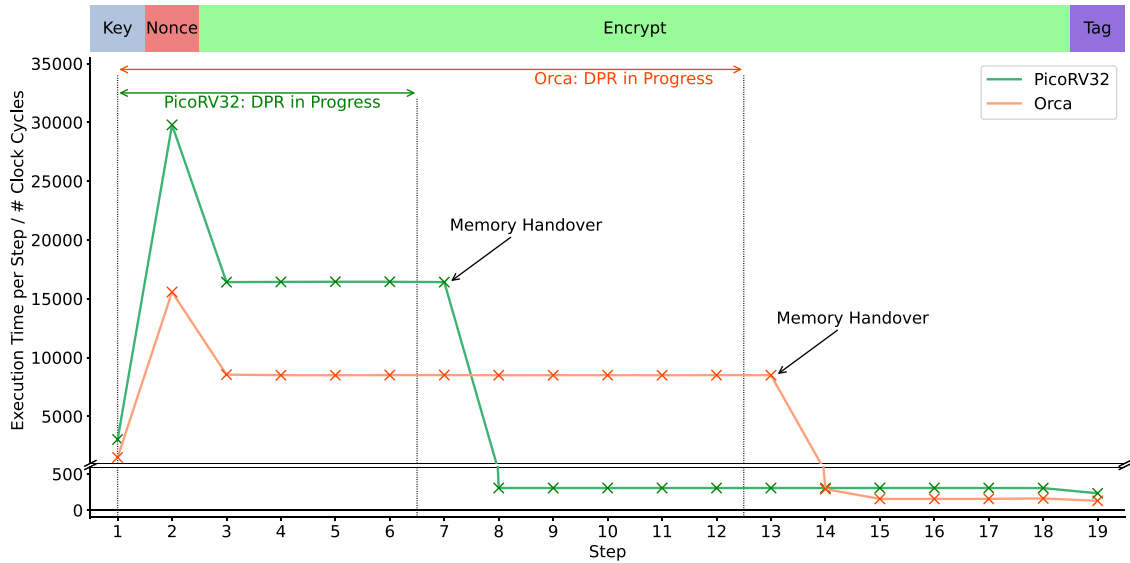


Fig. 5. Per-step execution time during on-demand ASCON instantiation (RM initially absent). The fallback path provides correctness while DPR runs in the background; the benchmark encrypts 124 bytes (31×32 -bit words, padded to 32). The first instruction triggers DPR, and the markers indicate the first step executed in hardware after the RM becomes present.

6.2.1. On-demand instantiation for ASCON (automatic reconfiguration)

This evaluation additionally demonstrates on-demand instantiation: the requested ASCON RM is initially absent, the fallback path provides immediate correctness, and the *Automatic Reconfiguration Module* triggers DPR so that later invocations hit in hardware. To ensure an initial miss, both RPs are configured with RMs that do not contain the ASCON accelerator, and the *Automatic Reconfiguration Module* is activated. This test case encrypts 124 bytes in 31 words.⁸ It is split into steps, each consisting of either forwarding key/nonce, encrypting 64-bit blocks, or generating the tag.

The first instruction requesting the absent ASCON RM triggers a DPR to populate one of the two RPs with the ASCON RM. The fallback path is used while the DFX controller reconfigures the RP. Once DPR finishes, the ASCON state is transferred (if configured), and the RM takes over. This transition can be identified by comparing step timings in Table 6 and Fig. 5 with the baseline times for regular software, hardware, and software fallback shown in Table 7 and Fig. 6.

As discussed in Section 5.1, fallback execution can slow down during ongoing DPR due to contention with shared memory/interconnect resources during partial-bitstream transfers. While fallback execution overlaps with DPR, both may contend for shared memory/interconnect bandwidth; thus, the overlap provides functional progress and hides some latency, but does not eliminate interference. From the measurements in Tables 6 and 7 and the step-by-step timings in Figs. 5 and 6, we observe that initial invocations incur a one-time DPR overhead. Once reconfiguration is complete, the hardware acceleration significantly reduces encryption time compared to software-only execution. We also observe measurable interference on the memory buses, leading to slightly higher execution time for regular software during partial bitstream fetches. This underscores that reconfiguration frequency and scheduling policy are first-class design parameters in such systems. Nevertheless, the accelerated steps still outperform purely software-based encryption, confirming that on-demand reconfiguration can be a viable strategy when accelerators are reused multiple times. From a predictability perspective, the key property of the proposed execution

model is that DPR proceeds asynchronously in the background, while equivalent behavior and forward progress must be ensured by the software fallback path. Thus, at the instruction level, the worst-case execution time is in principle bounded by the corresponding fallback rather than by reconfiguration completion. A full worst-case execution time analysis would nevertheless still need to account for the fallback cost itself, possible state-handover delays for stateful accelerators, and any remaining shared-resource interference of the concrete platform.

7. Design guidelines and applicability

The evaluation suggests three practical rules of thumb for applying the proposed architecture. First, the approach is most attractive for accelerators with sufficiently high *per-invocation computation*, because the miss path incurs a largely fixed transparency cost that is difficult to amortize for very small instruction-style operations such as ROL. Second, accelerators should exhibit sufficient *reuse*: if a requested RM is likely to be used repeatedly after instantiation, the one-time DPR cost can be amortized over subsequent hardware hits; otherwise, repeated misses and reconfigurations reduce the benefit. Third, stateful accelerators benefit particularly when their state is reused across many operations, since the handover mechanism preserves correctness but also adds management overhead that should be amortized by the accelerated workload.

In our measurements, this difference is clearly visible: ASCON remains well suited because its computation per invocation is high and the accelerator is reused across multiple steps, whereas ROL represents an intentionally unfavorable corner case with little work per invocation and therefore a very high fallback overhead. Practically, the architecture is best suited to medium- to high-compute custom instructions or multi-step stateful accelerators with temporal locality, and less suited to very small, frequently invoked instruction-style extensions unless misses are expected to be rare.

8. Conclusion and future work

This work showed that *runtime-reconfigurable* instruction set extensions can be integrated into RISC-V-based MCUs in a way that preserves the ISA-level programming model, even when accelerators appear and

⁸ The data is padded by another 4 bytes (1 word) to obtain 32 total words for ASCON to work with.

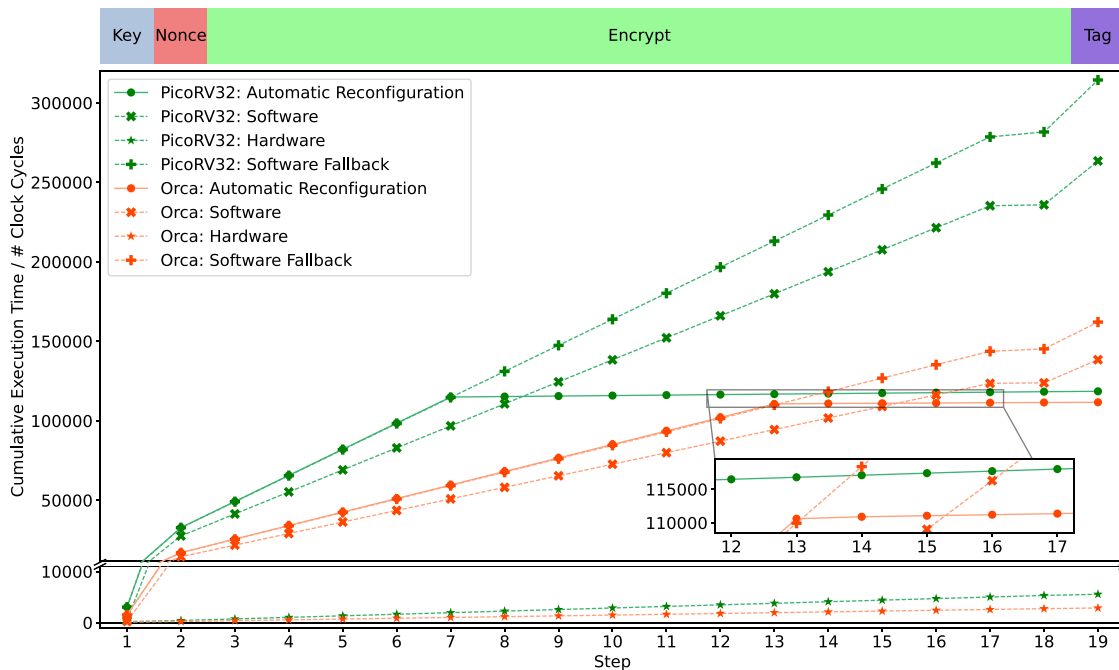


Fig. 6. Accumulated ASCON runtime across steps: Software vs. Hardware vs. Software Fallback vs. Automatic Reconfiguration. Same workload as Fig. 5 (encrypt 124 bytes); the automatic-reconfiguration curve transitions from fallback-dominated steps to hardware-hit steps once DPR completes.

disappear at runtime. Building on DPR, we implemented a full-FPGA MCU architecture in which accelerators are instantiated as RMs inside RPs and invoked via a custom instruction interface. Using SCAIE-V as the integration layer, the same concept was realized for two different cores (PicoRV32 and Orca), demonstrating that the approach is not tied to a specific microarchitecture style (non-pipelined vs. pipelined). Importantly, this integration keeps the *static* infrastructure (core, memories, interconnect, and reconfiguration control) generic while moving application specificity into dynamically loaded RMs.

Beyond simply mapping accelerators into the instruction stream, the main contribution of this extended work is the explicit treatment of *system-level semantics* under runtime reconfiguration. Rather than exposing accelerator availability as a software-visible feature, we implemented a hardware-backed software fallback path that turns a missing accelerator into an implementation detail: when a custom instruction targets an absent RM, dedicated hardware captures the instruction context, triggers a controlled context switch, executes a developer-friendly C/C++ fallback function, and finally seamlessly restores architectural state such that execution continues *as if the instruction had executed in hardware*. Importantly, this mechanism also supports program-flow changes initiated by the custom instruction, which is essential for instruction-style accelerators that behave like structured control-flow building blocks.

To extend this transparency from *stateless* to *stateful* accelerators, we further introduced the *Memory Handover Manager*. It externalizes accelerator-local state to memory-resident state with a well-defined layout, shared between the hardware RMs and their software fallback implementations. As a result, computations can migrate across the hardware/software boundary without loss of correctness: an accelerator can be evicted (e.g., due to DPR), emulated in software, and later resumed in hardware after it is re-instantiated. Together with the *Automatic Reconfiguration Module*, the platform provides a concrete form of accelerator “virtualization”: scarce fabric resources can be time-shared while software continues to observe stable instruction semantics.

Our experimental evaluation on a Nexys 4 DDR board (Artix-7 class FPGA) quantifies the benefits and costs of this approach. The first

case study on ROLs illustrates that instruction-style acceleration can reduce loop overhead when the accelerator is present, but also that a fully transparent fallback path introduces a non-trivial fixed overhead that dominates for low-compute operations. In contrast, the second study on ASCON highlights the intended sweet spot: for compute-intensive, stateful workloads, hardware acceleration can deliver significant speedups, while the fallback remains acceptable as an occasional bridge during reconfiguration. The automatic reconfiguration experiments further show the expected one-time DPR cost on first use, followed by fast steady-state execution once the RM is resident. We also observed measurable interference during partial bitstream fetches, underscoring that reconfiguration frequency and scheduling policy are first-class design parameters in such systems.

Although this paper focuses on cycle-level behavior, the same trade-offs also have an energy and endurance dimension. Reusing a limited pool of fabric resources via DPR can reduce the need to permanently reserve logic for infrequently used accelerators, which may improve overall resource efficiency over a product lifetime. Conversely, repeated fallback execution and frequent reconfiguration incur extra compute, memory traffic, and bitstream-transfer activity, and therefore should be used sensibly. From this perspective, the architecture is especially attractive for long-lived embedded systems in which accelerator sets may need to evolve after deployment, while the actual runtime policy should still minimize unnecessary reconfigurations and exploit accelerator reuse.

Looking forward, several directions can strengthen both the *efficiency* and the *generality* of runtime-reconfigurable ISAXs:

- **Reduce fallback overhead and improve transparency.** The current software fallback path intentionally prioritizes correctness and generality. Future versions could reduce overhead by (i) minimizing context-save/restore traffic, (ii) providing optional hardware assistance for fast context switching, and (iii) integrating compiler support so that fallback becomes cheaper and more toolchain-friendly without sacrificing semantic equivalence.

- **Richer accelerator interfaces while keeping the instruction model.** Many practical accelerators require more than two 32-bit operands and one 32-bit result, or need higher-bandwidth memory interaction. Extending the ISAX interface with optional multi-word operand/result protocols, streaming modes, or tighter memory access support (while preserving the same ISA-level abstraction) would broaden applicability.
- **Move towards tightly coupled eFPGA/SoC solutions.** While this work prototypes the entire platform on a discrete FPGA, a natural next step is to harden the *static* subsystem (core, memories, interconnect, and management/control) into an ASIC while retaining lifetime adaptability via an integrated eFPGA region. This would enable tighter coupling, lower power, and a more product-grade path while preserving the same ISA-level abstraction of optional, swappable instruction extensions. In such a setting, our fallback and state-handover mechanisms remain directly applicable, but the design space shifts towards eFPGA fabric organization, configuration bandwidth, and the runtime trade-offs between on-chip retargeting cost and achieved specialization.
- **Smarter runtime policies and predictability-aware scheduling.** The current reconfiguration policy is a reasonable baseline, but richer policies could exploit phase behavior, temporal locality, and workload hints (e.g., compiler- or OS-provided) to reduce reconfiguration costs. For real-time systems, future work should also incorporate predictability constraints: explicitly modeling DPR cost, memory interference, and worst-case behavior, and providing reconfiguration schedules that remain analyzable.
- **Automated design-space exploration for what to accelerate and when to swap.** Beyond evaluating fixed case studies, the next step is to automatically identify accelerator candidates, determine suitable RPs/RMs mappings, and synthesize reconfiguration policies from workload characteristics and platform constraints.
- **Hide or amortize DPR latency.** Architectural techniques such as partial-bitstream prefetching, decoupling bitstream loading from RP activation, compression/decompression, and overlapping reconfiguration-related memory traffic with computation can increase the average performance by reducing the overhead perceived by the software (at possibly higher hardware costs). Exploring multi-buffered state storage and double-layered activation schemes is particularly promising for frequent switching scenarios.
- **Scaling out: more RPs, more RMs, and multi-tenant software.** Extending the prototype from two RPs to a larger pool raises questions about arbitration, contention, and system software integration. A natural next step is to study how multiple applications (or protection domains) can share a reconfigurable accelerator pool while maintaining correctness and controlled interference.
- **Security and robustness.** Runtime reconfiguration, custom instructions, and state migration also introduce new attack surfaces. Future work should investigate integrity protection for partial bitstreams and the system state, controlled access to management functions, and side-channel-aware designs for accelerators that handle secrets.

Beyond performance and flexibility, the presented approach contributes to sustainability: instead of permanently dedicating fabric resources to infrequently used functionality, accelerators can be loaded on demand and time-shared via DPR, improving long-term utilization of a fixed hardware budget. Overall, this paper provides a concrete, end-to-end blueprint for making runtime-reconfigurable ISAXs behave like stable ISA-level abstractions – and thus a robust foundation for future adaptive, long-lived, and reusable embedded computing platforms.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

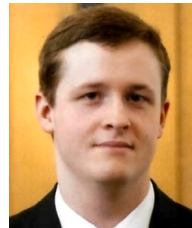
References

- [1] A. Waterman, K. Asanović, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, Technical Report, RISC-V Foundation, 2019, available online: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>.
- [2] A. Waterman, K. Asanović, J. Hauser, The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203, Technical Report, RISC-V International, 2021, available online: <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>.
- [3] T. Scheipel, M. Ogris, M. Baunach, You shall not stall: Achieving RISC-V on-demand runtime-reconfiguration using SCAIE-V, in: Proceedings of the 28th Euromicro Conference on Digital System Design, DSD, 2025, pp. 292–299, <http://dx.doi.org/10.1109/DSD67783.2025.00049>.
- [4] M. Damian, J. Oppermann, C. Spang, A. Koch, SCAIE-V: an open-source SCALable interface for ISA extensions for RISC-V processors, in: Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC, Association for Computing Machinery, New York, NY, USA, 2022, pp. 169–174, <http://dx.doi.org/10.1145/3489517.3530432>.
- [5] YosysHQ, YosysHQ/PICORV32: PicoRV32 - a size-optimized RISC-V CPU, 2025, URL: <https://github.com/YosysHQ/picorv32>.
- [6] VectorBlox, Cahz/orca: Vectorblox ORCA, 2025, URL: <https://github.com/cahz/orca>.
- [7] C. Dobraunig, M. Eichlseder, F. Mendel, M. Schläffer, Ascon v1.2: Lightweight authenticated encryption and hashing, *J. Cryptology* 34 (3) (2021) 33.
- [8] AMD, AMD Vivado™ design suite, 2025, URL: <https://www.xilinx.com/products/design-tools/vivado.html>.
- [9] Advanced Micro Devices, Inc., Dynamic function eXchange controller v1.0 product guide (PG374), 2024, URL: <https://docs.amd.com/v/u/en-US/pg374-dfx-controller>.
- [10] Advanced Micro Devices, Inc., 7 Series FPGAs: Overview DS180 (v2.6.1), Xilinx, Inc., 2020, [online] https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview.
- [11] K. Vipin, S.A. Fahmy, FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications, *ACM Comput. Surv.* 51 (4) (2018).
- [12] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P.-F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, K. Asanovic, An agile approach to building RISC-V microprocessors, *IEEE Micro* 36 (2) (2016) 8–20.
- [13] PULP Team, PULP platform (parallel ultra-low-power), 2025, <https://pulp-platform.org/>, (Accessed 16 December 2025).
- [14] OpenHW Group, CORE-V-XIF: eXtension Interface (XIF) for CORE-V cores, 2025, <https://github.com/openhwgroup/core-v-xif>, (Accessed 16 December 2025).
- [15] T. Scheipel, L. Batista Ribeiro, T. Sagaster, M. Baunach, SmartOS: An OS architecture for sustainable embedded systems, in: Proceedings of the FGOS Spring Meeting, 2022, pp. 1–10.
- [16] T. Scheipel, F. Angermair, M. Baunach, moreMCU: A runtime-reconfigurable RISC-V platform for sustainable embedded systems, in: Proceedings of the 25th Euromicro Conference on Digital System Design, DSD, 2022, pp. 24–31, <http://dx.doi.org/10.1109/DSD57027.2022.00013>.
- [17] N. Dao, A. Attwood, B. Healy, D. Koch, FlexBex: A RISC-V with a reconfigurable instruction extension, in: Proceedings of the International Conference on Field-Programmable Technology, ICFPT, 2020, pp. 190–195, <http://dx.doi.org/10.1109/ICFPT51103.2020.00034>.
- [18] lowRISC, lowRISC/ibex: Ibex RISC-V core, 2025, URL: <https://github.com/lowRISC/ibex>.
- [19] FPGA-Research-Manchester, FPGA-Research-Manchester/FABulous: Fabric generator and CAD tools, 2025, URL: <https://github.com/FPGA-Research-Manchester/FABulous>.
- [20] P.D. Schiavone, D. Rossi, A. Di Mauro, F.K. Gürkaynak, T. Saxe, M. Wang, K.C. Yap, L. Benini, Arnold: An eFPGA-augmented RISC-V SoC for flexible and low-power IoT end nodes, *IEEE Trans. VLSI Syst.* 29 (4) (2021) 677–690.
- [21] L. Moser, M. Kissich, T. Scheipel, M. Baunach, Greyhound: A reconfigurable and extensible RISC-V SoC and eFPGA on IHP SG13G2, in: Proceedings of the 33rd Austrian Workshop on Microelectronics (Austrochip), 2025, pp. 5–8, <http://dx.doi.org/10.1109/Austrochip67945.2025.11183709>.
- [22] OpenHW Group, CV32E40X, 2025, available online: <https://github.com/openhwgroup/cv32e40x>.

- [23] B. Neumann, T. von Sydow, H. Blume, T.G. Noll, Application domain specific embedded FPGAs for flexible ISA-extension of ASIPs, *J. Signal Process. Syst.* 53 (2008) 129–143.
- [24] F. Lesniak, T. Harbaum, J. Becker, Approximate accelerators: A case study using runtime reconfigurable processors, in: Proceedings of the IEEE 36th International System-on-Chip Conference, SOCC, 2023, pp. 1–6, <http://dx.doi.org/10.1109/SOCC58585.2023.10257090>.
- [25] FrontgradeGaisler, LEON3(SPARC), 2025, URL: <https://www.gaisler.com/products/leon3>.
- [26] M. Monopoli, L. Zulberti, G. Todaro, P. Nannipieri, L. Fanucci, Exploiting FPGA dynamic partial reconfiguration for a soft GPU-based system-on-chip, in: Proceedings of the 18th Conference on Ph.D. Research in Microelectronics and Electronics, PRIME, 2023, pp. 181–184, <http://dx.doi.org/10.1109/PRIME58259.2023.10161859>.
- [27] AMD, AXI hardware ICAP, 2017, URL: https://www.xilinx.com/products/intellectual-property/axi_hwicap.html.
- [28] AMD, Zynq 7000 SoC technical reference manual (UG585), 2024, URL: <https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM>.
- [29] K. Vipin, S.A. Fahmy, ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq, *IEEE Embed. Syst. Lett.* 6 (3) (2014) 41–44.
- [30] K. Vipin, S.A. Fahmy, A high speed open source controller for FPGA partial reconfiguration, in: Proceedings of the 11th International Conference on Field-Programmable Technology, FPT, 2012, pp. 61–66, <http://dx.doi.org/10.1109/FPT.2012.6412113>.
- [31] K. Vipin, S.A. Fahmy, Automated partial reconfiguration design for adaptive systems with CoPR for Zynq, in: Proceedings of the IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2014, pp. 202–205, <http://dx.doi.org/10.1109/FCCM.2014.63>.
- [32] E. El-Araby, I. Gonzalez, T. El-Ghazawi, Exploiting partial runtime reconfiguration for high-performance reconfigurable computing, *ACM Trans. Reconfigurable Technol. Syst.* 1 (4) (2009).
- [33] S. Liu, R.N. Pittman, A. Forin, J.-L. Gaudiot, On energy efficiency of reconfigurable systems with run-time partial reconfiguration, in: Proceedings of the 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP, 2010, pp. 265–272, <http://dx.doi.org/10.1109/ASAP.2010.5540985>.
- [34] N. Charaf, A. Kamaleldin, M. Thümmel, D. Göhringer, RV-CAP: Enabling dynamic partial reconfiguration for FPGA-based RISC-V system-on-chip, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2021, pp. 172–179, <http://dx.doi.org/10.1109/IPDPSW52791.2021.00033>.
- [35] L. Pezzarossa, A.T. Kristensen, M. Schoeberl, J. Sparso, Can real-time systems benefit from dynamic partial reconfiguration? in: Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2017, pp. 1–6, <http://dx.doi.org/10.1109/NORCHIP.2017.8124984>.
- [36] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. Probst, Towards a time-predictable dual-issue microprocessor: The patmos approach, in: Proceedings of the 1st Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems, PPES 2011, 2011, pp. 11–21, <http://dx.doi.org/10.4230/OASICS.PPES.2011.11>.
- [37] T-CREST, T-crest/patmos: Patmos is a time-predictable VLIW processor, and the processor for the T-CREST project, 2025, URL: <https://github.com/t-crest/patmos>.
- [38] A. Adetomi, G. Enemali, X. Iturbe, T. Arslan, D. Keymeulen, R3TOS-based integrated modular space avionics for on-board real-time data processing, in: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, AHS, 2018, pp. 1–8, <http://dx.doi.org/10.1109/AHS.2018.8541369>.
- [39] R.N. Pittman, N.L. Lynch, A. Forin, R. Pittman, N. Lynch, R. Forin, eMIPS, A Dynamically Extensible Processor, Tech. Rep. MSR-TR-2006-143, Microsoft Research, 2006.
- [40] A. Traber, M. Gautschi, P.D. Schiavone, RI5CY: User manual, 2019, URL: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.
- [41] T. Scheipel, P. Brungs, M. Baunach, A hardware/software concept for partial logic updates of embedded soft processors at runtime, in: Proceedings of the 24th Euromicro Conference on Digital System Design, DSD, 2021, pp. 199–207, <http://dx.doi.org/10.1109/DSD53832.2021.00040>.
- [42] OpenHWGroup, OpenHW Group CORE-V CV32E40P RISC-V IP, 2025, URL: <https://github.com/openhwgroup/cv32e40p>.
- [43] ARM Ltd., AMBA AXI Protocol v1.0, ARM Ltd., 2004, URL: http://mazzola.iit.uni-miskolc.hu/~drdani/docs_arm/AMBAaxi.pdf.
- [44] Diligent, Nexys 4 DDR (Legacy), 2016, URL: <https://diligent.com/reference/programmable-logic/nexys-4-ddr/start>.



Dr. Tobias Scheipel is a tenure-track Assistant Professor of Reconfigurable Computer Architectures at the Embedded Architectures & Systems Group at the Institute of Technical Informatics, Graz University of Technology and a RISC-V Advocate. He conducted his doctoral/Ph.D. studies under the supervision of Prof. Marcel Baunach, where he got promoted to a Doctor of Engineering Sciences (Dr.techn.) under the auspices of the Federal President of Austria. The title of his dissertation is Advances in Dynamic and Reconfigurable Embedded Systems Design. Before his doctoral studies, he received both his Bachelor's and Master's degrees (Dipl.-Ing.) in Information and Computer Engineering at Graz University of Technology in Austria. His research focuses on flexible and runtime-reconfigurable FPGA-based microcontroller architectures for embedded systems based on RISC-V. This involves hardware/software codesign strategies for both processor logic and embedded operating systems. Within this research area, he is the author of several peer-reviewed scientific publications in international journals and conference proceedings.



Maximilian Ogris started his technical education at the HTL Bulme Graz, Austria. He subsequently studied Information and Computer Engineering at the Graz University of Technology and earned his Bachelor and Master Degree with distinction. Furthermore, he was listed on the Dean's List, recognizing the top 5% of students in his specialization, for three years during his Bachelor studies.

During an exchange semester at the Norwegian University of Science and Technology in Trondheim in Autumn 2022, he developed a strong interest in microprocessor development. Building on this focus, he completed his master's thesis at Graz University of Technology entitled Runtime-Reconfigurable Hardware Accelerators Extending the ISA of RISC-V-based Microcontrollers, under the supervision of Dr. Tobias Scheipel. The thesis resulted in a scientific paper and a more comprehensive version in the form of this publication. After graduating, he joined the Anton Paar GmbH in Graz, Austria, as an embedded software and systems engineer. His interests include FPGA and ASIC design, embedded software development, general software engineering, hardware–software co-design, and runtime-reconfigurable FPGAs.



Prof. Marcel Baunach is head of the Embedded Architectures & Systems Group at the Institute of Technical Informatics at Graz University of Technology. He completed his Diploma and PhD with distinction at the University of Würzburg, Germany. He then went to industry as head of hardware development in the field of automotive diagnostics. Back at academia, his research at TU Graz is on hardware and software for highly dependable and sustainable embedded systems, in particular on real-time operating systems and processor architectures. The focus is on concepts for the design, verification and portability of system software, the dynamic composition of modular systems, as well as on application-specific and reconfigurable processors. Application areas include future vehicles, the IoT, or cyber–physical systems.