ARCHIVIO ISTITUZIONALE DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

*Published Version:*
Sa, B., Valente, L., Martins, J., Rossi, D., Benini, L., Pinto, S. (2023). CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration. IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, 31(11), 1713-1726 [10.1109/TVLSI.2023.3302837].

*Availability:*
This version is available at: https://hdl.handle.net/11585/952940 since: 2024-01-12

*Published:*
DOI: http://doi.org/10.1109/TVLSI.2023.3302837

*Terms of use:*

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

15 September 2026

This is the final peer-reviewed accepted manuscript of:

**B. Sá, L. Valente, J. Martins, D. Rossi, L. Benini and S. Pinto, "CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 11, pp. 1713-1726, Nov. 2023**

The final published version is available online at:

https://doi.org/10.1109/TVLSI.2023.3302837

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

*This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)*

***When citing, please refer to the published version.***

# CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration

Bruno Sá *, Luca Valente †, José Martins *, Davide Rossi †, Luca Benini † ‡, Sandro Pinto *
* Centro ALGORTIMI/LASI, Universidade do Minho, Portugal
† DEI, University of Bologna, Italy ‡ IIS lab, ETH Zurich, Switzerland

arXiv:2302.02969v3 [cs.AR] 4 Aug 2023

***Abstract*—Virtualization is a key technology used in a wide range of applications, from cloud computing to embedded systems. Over the last few years, mainstream computer architectures were extended with hardware virtualization support, giving rise to a set of virtualization technologies (e.g., Intel VT, Arm VE) that are now proliferating in modern processors and SoCs. In this article, we describe our work on hardware virtualization support in the RISC-V CVA6 core. Our contribution is multifold and encompasses architecture, microarchitecture, and design space exploration. In particular, we highlight the design of a set of microarchitectural enhancements (i.e., G-Stage Translation Lookaside Buffer (GTLB), L2 TLB) to alleviate the virtualization performance overhead. We also perform a Design Space Exploration (DSE) and accompanying post-layout simulations (based on 22nm FDX technology) to assess Performance, Power ,and Area (PPA). Further, we map design variants on an FPGA platform (Genesys 2) to assess the functional performance-area trade-off. Based on the DSE, we select an optimal design point for the CVA6 with hardware virtualization support. For this optimal hardware configuration, we collected functional performance results by running the MiBench benchmark on Linux atop Bao hypervisor for a single-core configuration. We observed a performance speedup of up to 16% (approx. 12.5% on average) compared with virtualization-aware non-optimized design at the minimal cost of 0.78% in area and 0.33% in power. Finally, all work described in this article is publicly available and open-sourced for the community to further evaluate additional design configurations and software stacks.**

***Index Terms*—Virtualization, CVA6, Microarchitecture, TLB, MMU, Design Space Exploration, Hypervisor, RISC-V.**

## I. Introduction

Virtualization is a technological enabler used on a large spectrum of applications, ranging from cloud computing and servers to mobiles and embedded systems [1]. As a fundamental cornerstone of cloud computing, virtualization provides numerous advantages for workload management, data protection, and cost-/power-effectiveness [2]. On the other side of the spectrum, the embedded and safety-critical systems industry has resorted to virtualization as a fundamental approach to address the market pressure to minimize size, weight, power, and cost (SWaP-C), while guaranteeing temporal and spatial isolation for certification (e.g., ISO26262) [3]–[5]. Due to the proliferation of virtualization across multiple industries and use cases, prominent players in the silicon industry started to introduce hardware virtualization support in mainstream computing architectures (e.g., Intel Virtualization Technology, Arm Virtualization Extensions, respectively) [6], [7].

Recent advances in computing architectures have brought to light a novel instruction set architecture (ISA) named RISC-V [8]. RISC-V has recently reached the mark of 10+ billion shipped cores [9]. It distinguishes itself from the classical mainstream by providing a free and open standard ISA, featuring a modular and highly customizable extension scheme that allows it to scale from small microcontrollers up to supercomputers [10]–[16]. The RISC-V privileged architecture provides hardware support for virtualization by defining the Hypervisor extension [17], ratified in Q4 2021.

Despite the Hypervisor extension ratification, as of this writing, there is no RISC-V silicon with this extension on the market[1]. There are open-source hypervisors with upstream support for the Hypervisor extension, i.e., Bao [1], Xvisor [18], KVM [19], and seL4 [20] (and work in progress in Xen [21] and Jailhouse [22]). However, to the best of our knowledge, there are just a few hardware implementations deployed on FPGA, which include the Rocket chip [23] and NOEL-V [24] (and soon SHAKTI and Chromite [25]). Notwithstanding, no existing work has (i) focused on understanding and enhancing the microarchitecture for virtualization and (ii) performed a design space exploration (DSE) and accompanying power, performance, area (PPA) analysis.

This work describes the architectural and microarchitectural support for virtualization in an open-source RISC-V CVA6-based [14] (64-bit) SoC. At the architectural level, the implementation is compliant with the Hypervisor extension (v1.0) [17] and includes the implementation of the RISC-V timer (Sstc) extension [26] as well. At the microarchitectural level, we modified the vanilla CVA6 microarchitecture to support the Hypervisor extension and proposed a set of additional extensions/enhancements to reduce the hardware virtualization overhead: (i) a dedicated second stage Translation Lookaside Buffer (TLB) coupled to the Page Table Walker (PTW) (i.e., G-Stage TLB (GTLB) in our lingo), and (ii) a second level TLB (L2 TLB). We also present and discuss a comprehensive design space exploration on the microarchitecture. We first evaluate 23 (out of 288) hardware designs deployed on FPGA (Genesys 2) and assess the impact on functional performance (execution cycles) and hardware. Then, we elect 7 designs and analyze them in depth with post-layout simulations of implementations in 22nm FDX technology.

We ran the MiBench (automotive subset) benchmarks for the DSE evaluation to assess functional performance. The virtualization-aware, non-optimized CVA6 implementa-

[1] SiFive, Ventana, and StarFive have announced RISC-V CPU designs with Hypervisor extension support, but we are not aware of any silicon available on the market yet.

tion served as our baseline configuration. The software setup encompassed a single Linux Virtual Machine (VM) running atop Bao hypervisor for a single-core design. We measured the performance speedup of the hosted Linux VM relative to the baseline configuration. Results from the DSE exploration demonstrated that the proposed microarchitectural extensions could achieve a functional performance speedup up to 19% (e.g., for the *susanc* small benchmark); however, in some cases at the cost of a non-negligible increase in area and power. Thus, results from the PPA analysis show that: (i) the Sstc extension has negligible impact on power and area; (iii) the GTLB increases the overall area in less than 1%; and (iv) the L2 TLB introduces a non-negligible 8% increase in area in some configurations. As a representative, well-balanced configuration, we selected the CVA6 design with Sstc support and a GTLB with 8 entries. For this specific hardware configuration, we observed a performance speedup of up to 16% (approx. 12.5% on average) at the cost of 0.78% in area and 0.33% in power. To the best of our knowledge, this paper reports the first public work on a complete DSE evaluation and PPA analysis for a virtualization-enhanced RISC-V core.

To summarize, with this work, we make the following contributions. Firstly, we provide hardware virtualization support for the CVA6 core, which was completely absent in the vanilla implementation. In particular, we implement the RISC-V Hypervisor extension (v1.0) and design a set of (virtualization-oriented) microarchitectural enhancements to the Nested Memory Management Unit (Section III). To the best of our knowledge, no work or study describes and discusses microarchitectural extensions to improve the hardware virtualization support in a RISC-V core. Second, we perform a DSE encompassing dozens of design configurations. This DSE includes trade-offs on parameters from three different microarchitectural components (L1 TLB, GTLB, L2 TLB) and respective impact on functional performance and hardware costs (Section IV). Finally, we conduct post-layout simulations on a few elected design configurations to assess a PPA analysis (Section V). All contributions described in this manuscript are open source [2] and available to the RISC-V community to foster collaboration and enable contributions with further extensions, optimizations, and testing/verification activities. The Hypervisor extension is now going towards formal upstream into the CVA6 main repository[3]. Our goal is to democratize virtualization for the next-generation CVA6-based SoCs.

## II. Background

This section covers the background related to RISC-V technology (ISA, extensions, and cores) and virtualization. To improve readability, we also provide a table of abbreviations and terminologies used throughout the article (see Table I).

### A. Virtualization Technology

Virtualization is the de facto technology that consolidates and isolates multiple non-related software stacks onto the same hardware platform by partitioning and multiplexing hardware resources (e.g., CPUs, memory) between multiple virtual machines. The virtual machine monitor (VMM) or hypervisor is the software layer that implements virtualization. The software executing in the VM is referred to as a guest, typically an operating system (OS), i.e., guest OS.

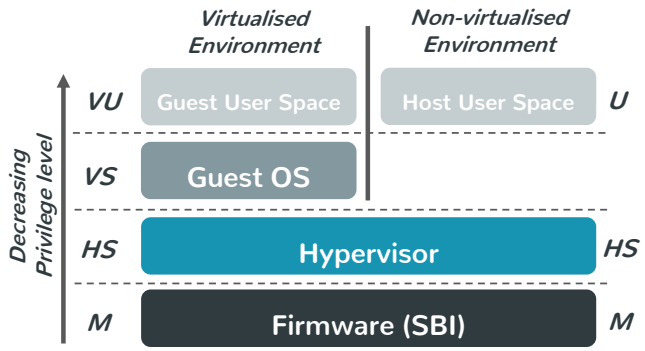


Fig. 1. RISC-V privilege levels: machine (M), hypervisor-extended supervisor (HS), virtual supervisor (VS), and virtual user (VU).

### B. RISC-V Privileged Specification

The RISC-V privileged instruction set architecture (ISA) [17] divides its execution model into 3 privilege levels: (i) *machine mode* (M-mode) is the most privileged level, hosting the firmware which implements the supervisor binary interface (SBI) (e.g., OpenSBI); (ii) *supervisor mode* (S-Mode) runs Unix type operating systems (OSes) that require virtual memory management; (iii) *user mode* (U-Mode) executes userland applications. The modularity offered by the RISC-V ISA seamlessly allows for implementations at distinct design points, ranging from small embedded platforms with just M-mode support to fully blown server class systems with M/S/U.

### C. RISC-V Virtualization

Unlike other mainstream ISAs, the RISC-V privileged architecture was designed from the initial conception to be classically virtualizable [27]. So although the ISA, per se, allows the straightforward implementation of hypervisors resorting, for example, to classic virtualization techniques (e.g., trap-and-emulation and shadow page tables), it is well understood that such techniques incur a prohibitive performance penalty and cannot cope with current embedded real-time virtualization requirements (e.g., interrupt latency) [23]. Thus, to increase virtualization efficiency, the RISC-V privileged architecture specification introduced hardware support for virtualization through the (optional) *Hypervisor* extension [17]. The following paragraphs provide a high-level overview of the RISC-V Hypervisor extension specification.

**Privilege Levels.** As depicted in Figure 1, the RISC-V Hypervisor extension execution model follows an orthogonal design where the *supervisor mode* (S-mode) is modified to an *hypervisor-extended supervisor mode* (HS-mode) well-suited to host both type-1 or type-2 hypervisors[4]. Additionally, two new privileged modes are added and can be leveraged to run

[2]https://github.com/minho-pulp/cva6/

[3]https://github.com/minho-pulp/cva6/tree/feat/hyp-upstream

[4]The main difference between type-1 (or baremetal) and type-2 (or hosted) hypervisors is that a type-1 hypervisor runs directly on the hardware (e.g., Xen) while a type-2 hypervisor runs atop an operating system (e.g., VMware).

TABLE I
TABLE OF ABBREVIATIONS, RISC-V TERMINOLOGIES AND RISC-V CONTROL STATUS REGISTERS AND INSTRUCTIONS.

| Name | Definition |
|---|---|
| **Abbreviations** | |
| PTW | Page Table Walker. The PTW is responsible for transversing the page table and converting a virtual address into a physical address. |
| DSE | Design Space Exploration. Functional evaluation carried out for multiple FPGA design configurations. |
| PPA | Power, Performance, Area. Post-layout silicon power, performance, and area analysis. |
| VM | Virtual Machine. The execution environment (e.g., OS or application) running atop a hypervisor. |
| Nested-MMU | Nested Memory Management Unit. Using two translation stages, convert a guest's virtual address into a host's physical address. The first is controlled by the guest OS and the second by the hypervisor. |
| TLB | Translation Lookaside Buffer. Microarchitectural element used to cache translations from virtual addresses to physical addresses. |
| VS-Mode | Virtual Supervisor Mode. Privilege mode in RISC-V, where a guest OS runs when virtualized. |
| HS-Mode | Hypervisor-extended Supervisor Mode. Privilege mode in RISC-V, where the hypervisor runs. |
| GTLB | G-stage Translation Lookaside Buffer. TLB that stores G-stage only translation, i.e., guest physical addresses into host physical addresses. |
| **RISC-V Terminologies** | |
| CSR | Control Status Registers. RISC-V CPU control registers. |
| VS-Stage | RISC-V terminology for a first translation stage. Converts guest virtual addresses into guest physical addresses. |
| G-Stage | RISC-V terminology for a second translation stage. Converts guest physical addresses into host physical addresses. |
| ASID | Address Space Identifier. Processes have a unique identifier used to tag translations on the MMU (e.g., on TLB entries). |
| VMID | Virtual Machine Space Identifier. Guest virtual machines have a unique identifier used to tag translations on the MMU. |
| PTE | Page Table Entry. Specific entry of a page table used to convert virtual addresses into physical addresses. |
| **RISC-V Control Status Registers and Instructions** | |
| *stimecmp* | Supervisor Time Comparator Register. Generates timer interrupts at HS/S-mode. Only available if the RISC-V Sstc extension is implemented. |
| *vstap* | Virtual Supervisor Guest Address Translation and Protection Register. Controls the VS-stage address translation and protection. |
| *vstimecmp* | Virtual Supervisor Time Comparator Register. Generates timer interrupts at VS-mode. Only available if the RISC-V Sstc extension is implemented. |
| *hgeie and hgeip* | Hypervisor Guest External Enable and Pending Registers. Deliver interrupts directly to the guest if the interrupt controller supports virtualization. |
| *hvencfg* | Hypervisor Virtual Environment Control Register. Controls hypervisor access (enabled or disabled) to optional platform-specific extensions (e.g., Sstc extension). |
| *hgatp* | Hypervisor Guest Address Translation and Protection Register. G-stage root table pointer and respective translation-specific configuration fields. |
| *menvcfg* | Machine Virtual Environment Control Register. Controls firmware access (enabled or disabled) to optional platform-specific extensions (e.g., Sstc extension). |
| *hfence* | Hypervisor Fence Instruction. Flush cached translations on TLBs and other related structures. The hfence.vvma flushes VS-Stage and G-Stage translations. The hfence.gvma flushes only G-Stage translations. |

the guest OS at *virtual supervisor mode* (VS-mode) and *virtual user mode* (VU-mode).

**Two-stage Address Translation.** The Hypervisor extension also defines a second translation stage (*G-stage* in RISC-V lingo) to virtualize the guest memory by translating guest-physical addresses into host-physical addresses. The HS-mode operates like S-mode but with additional hypervisor registers and instructions to control the VM execution and *G-stage* translation. For instance, the *hgatp* register holds the *G-stage* root table pointer and translation-specific configuration fields.

**Hypervisor Control and Status Registers (CSRs).** Each VM running in VS-mode has its own control and status registers (CSRs) that are shadow copies of the S-mode CSRs. These registers can be used to determine the guest execution state and perform VM switches. To control the virtualization state, a specific flag called *virtualization mode* (V bit) is used. When V=1, the guest is executing in VS-mode or VU-mode, normal S-mode CSRs accesses are actually accessing the VS-mode CSRs, and the *G-stage* translation is active. Otherwise, if V=0, normal S-mode CSRs are active, and the *G-stage* is disabled. To ease guest-related exception trap handling, there are guest-specific traps, e.g., guest page faults, VS-level illegal exceptions, and VS-level *ecalls* (a.k.a. hypercalls).

**Hypervisor Instructions.** The Hypervisor extension defines a set of hypervisor-related instructions to increase the virtualization efficiency. These instructions encompass: (i) hypervisor load/store instructions used to access guest memory with the exact translation and permissions rights (RWX) as the guest; and (ii) hypervisor fence instructions synchronization mechanisms to flush guest-related cached translation structures (e.g., TLBs) during context switching operations.

### *D. Nested Memory Management Unit (Nested-MMU)*

The MMU is a hardware component responsible for translating virtual memory references to physical ones while enforcing memory access permissions. The OS controls the MMU by assigning virtual address space to each process and managing the MMU translation structures to translate virtual addresses into physical addresses correctly. On a virtualized system, the MMU provides another layer of translation and protection controlled by the hypervisor. In this case, the MMU can translate from guest-virtual addresses to guest-physical addresses and from guest-physical addresses into host-physical addresses. This new feature is referred to as nested-MMU. The RISC-V ISA supports the nested-MMU through a new stage of translation that converts guest-physical addresses into host-physical addresses, denoted G-stage. The guest VM takes control over the first stage of translation (VS-stage in RISC-V lingo), while the hypervisor assumes control over the second one (G-stage). Originally, the RISC-V privileged specification defines that a virtual address is converted into a physical address by traversing a multi-level radix-tree table using one of four different topologies: (i) *Sv32* for 32 virtual address spaces with a 2-level hierarchy tree; (ii) *Sv39* for 39-bit virtual address spaces with a 3-level tree; (iii) *Sv48* for 48-bit VAS and 4-level tree; and (iv) *Sv57* for 57-bit virtual address spaces and 5-level tree. Each level holds a pointer to the following table (non-leaf entry) or the final translation (leaf entry). This pointer and permissions are stored in a 64-bit (RV64) or 32-bit (RV32) width page table entry (PTE). Note that RISC-V splits the virtual address into 4KiB page sizes, but since each level can either be a leaf or non-leaf, it supports superpages to reduce the TLB pressure, e.g., Sv39 supports 4KiB, 2MiB, and 1GiB page sizes.

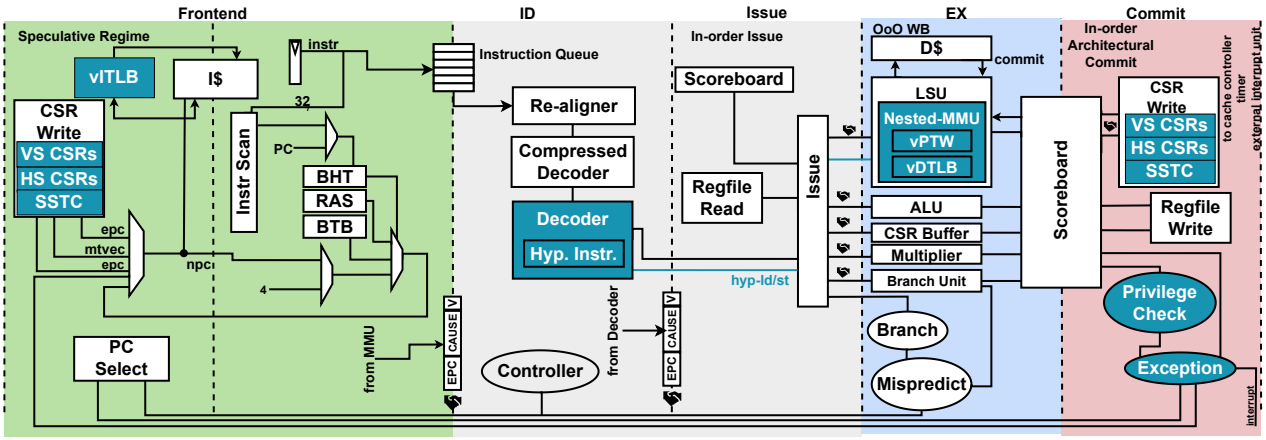


Fig. 2. CVA6 core microarchitecture featuring the RISC-V hypervisor extension. Major microarchitectural changes to the core functional blocks (e.g., Decoder, PTW, TLBs, and CSRs) are highlighted in blue. Adapted from [14].

### *E. RISC-V "stimecmp/vstimecmp" Extension (Sstc)*

The RISC-V Sstc extension [26] aims to enhance supervisor mode with its timer interrupt facility, thus eliminating the large overheads for emulating S/HS-mode timers and timer interrupt generation up in M-mode. The Sstc extension also adds a similar facility to the Hypervisor extension for VS-mode. To enable direct control over timer interrupts in the HS/S-mode and VS-mode, the Sstc encompasses two additional comparators: (i) *stimecmp* register to generate S/HS-mode timer interrupts and (ii) *vstimecmp* register to generate VS-mode timer interrupts. Whenever the value of the system clock *time* counter register is greater than the value on the comparators, a timer interrupt is generated. For a complete overview of the RISC-V timer architecture and a discussion on why the classic RISC-V timer specification incurs a significant performance penalty, we refer the interested reader to [23].

### *F. CVA6*

CVA6 is an application class RISC-V core that implements both the RV64 and RV32 versions of RISC-V ISA [14]. The core fully supports the 3 privilege execution modes M/S/U-modes and provides hardware support for memory virtualization by implementing a MMU, making it suitable for running a fully-fledged OS such as Linux. Recent additions also include a energy-efficient Vector Unit co-processor [28]. Internally, the CVA6 microarchitecture encompasses a 6-stage pipeline, single issue, with an out-of-order execution stage and 8 PMP entries. The MMU has separate TLBs for data and instructions and the PTW implements the *Sv39* and *Sv32* translation modes as defined by the privileged specification [17].

## III. CVA6 "From Scratch" Virtualization Support

This section reports our work in adding hardware support for virtualization in the CVA6 core (compliant with the RISC-V Hypervisor extension v1.0). To the best of our knowledge, this work is the first initiative to add hardware virtualization support in the CVA6 core. The following section divides into seven logical subsections: (i) Subsection III-A, III-B, III-C and III-D describes the modifications/additions to CVA6 microarchitecture enabling the Hypervisor extension support, specifically to the core registers and the MMU subsystem; (ii) Subsection III-E and III-F presents the microarchitectural enhancements designed in the MMU subsystem to increase virtualization efficiency; and (iii) Subsection III-G describes how we extend the CVA6 timer with the standard RISC-V Sstc extension to improve the timer management infrastructure. Figure 2 illustrates a high-level overview of the modifications/additions performed in the CVA6 microarchitecture.

### *A. Hypervisor and Virtual Supervisor Execution Modes*

As previously described (refer to Section II-C), the Hypervisor extension specification extends the S-mode into the HS-mode and adds two extra orthogonal execution modes, denoted VS-mode and VU-mode. To add support for these new execution modes, we have extended/modified some of the CVA6 core functional blocks, in particular, the *CSR* and *Decode* modules. As illustrated by Figure 2, the hardware virtualization architecture logic encompasses five building blocks: (i) VS-mode and HS-mode CSRs access logic and permission checks; (ii) exceptions and interrupts triggering and delegation; (iii) trap entry and exit; (iv) hypervisor instructions decoding and execution; and (v) nested-MMU translation logic. The *CSR module* was extended to implement the first three building blocks that comprise the hardware virtualization logic, specifically: (i) HS-mode and VS-mode CSRs access logic (read/write operations); (ii) HS and VS execution mode trap entry and return logic; and (iii) a fraction of the exception/interrupt triggering and delegation logic from M-mode to HS-mode and/or to VS/VU-mode (e.g., reading/writing to *vsatp* CSR triggers an exception in VS-mode when not allowed by the hypervisor). The *Decode* module was modified to implement hypervisor instructions decoding (e.g., hypervisor load/store instructions and memory-management fence instructions) and all VS-mode related instructions execution access exception triggering.

We refer readers to Table II, which presents a summary of the features that were fully and partially implemented. We have implemented all mandatory features of the ratified specification (v1.0); however, we still left some optional features as partially implemented due to the dependency on upcoming or newer extensions. For example, *hvencfg* bits depend on *Zicbom* [29] (cache block management operations); *hgeie* and

TABLE II
HYPERVISOR EXTENSION FEATURES IMPLEMENTED IN THE CVA6 CORE: ● FULLY-IMPLEMENTED; ◐ PARTIALLY IMPLEMENTED.

| | | |
|---|---|---|
| CSRs | hstatus/mstatus | ● |
| | hideleg/hedeleg/mideleg | ● |
| | hvip/hip/hie/mip/mie | ● |
| | hgeip/hgeie | ◐ |
| | hcounteren | ● |
| | htimedelta | ● |
| | henvcfg | ◐ |
| | mtval2/htval | ● |
| | mtinst/htinst | ● |
| | hgapt | ◐ |
| | vsstatus/vsip/vsie/vstvec/vsscratch vsepc/vscause/vstval/vsatp | ● |
| Intructions | hlv/hlvx/hsv | ● |
| | hfence.vvma/gvma | ● |
| Exceptions & Interrupts | Environment call from VS-mode | ● |
| | Instruction/Load/Store guest-page fault | ● |
| | Virtual instruction | ● |
| | Virtual Supervisor sw/timer/external interrupts | ● |
| | Supervisor guest external interrupt | ● |

*hgeip* depends on the Advanced Interrupt Architecture (AIA) [30]; and *hgatp* depends on virtual address spaces not currently supported in the vanilla CVA6.

### B. Hypervisor Load/Store Instructions

The hypervisor load/store instructions (i.e., *HLV*, *HSV*, and *HLVX*) provide a mechanism for the hypervisor to access the guest memory while subject to the exact translation and permission checks as in VS-mode or VU-mode. These instructions change the translation settings at the instruction granularity level, forcing a complete swap of privilege level and translation applied at every hypervisor load/store instruction execution. The implementation encompasses the addition of a signal (identified as *hyp_ld/st* in Figure 2) to the CVA6 pipeline that travels from the decoding to the load/store unit in the execute stage. This signal is then fed into the MMU that performs (i) all necessary context switches (i.e., enables the *hgatp* and *vstap* CSRs), (ii) enables the virtualization mode, and (iii) changes execution mode.

### C. Nested Page-Table Walker

One of the major functional blocks of an MMU is the PTW. Fundamentally, the PTW is responsible for partitioning a virtual address accordingly to the specific topology and scheme (e.g., *Sv39*) and then translating it into a physical address using the memory page tables structures. The Hypervisor extension specifies a new translation stage (G-stage) used to translate guest-physical addresses into host-physical addresses. Our implementation supports *Bare* translation mode (no G-stage) and *Sv39x4*, which defines a 41-bit width maximum guest physical address space. As of this writing, the CVA6 only supported the *Sv39* scheme for the 64-bit core, so we extended the implementation to support the *Sv39x4*.

We extended the existing finite state machine (FSM) used to translate virtual addresses to physical addresses and added only a new control state to keep track of the current translation stage and assist the context switching between VS-Stage and G-Stage translations. With the G-stage in situ, it is mandatory to translate: (i) the resulting guest-physical address from the VS-Stage translation and (ii) all memory accesses made during the VS-Stage translation walk. To accomplish that, we identify three stages of translation that can occur during a PTW iteration: (i) *VS-Stage* - the PTW current state is translating a guest-virtual address into a guest-physical address; (ii) *G-Stage Intermed* - the PTW current state is translating memory access made from the VS-Stage during the walk to host-physical address; and (iii) *G-Stage Final* - the PTW current state is translating the final output address from VS-Stage into a host-physical address. It is worth noting that if MMU is configured in Bare mode for the G-stage translation, we perform a standard S/VS-Stage translation. Once the nested walk completes, the PTW updates the TLB with the final PTE from VS-stage and G-stage, alongside the current Address Space Identifier (ASID) and the Virtual-Machine Identifier (VMID). VMID tags each translation to a specific VM. One implemented optimization consists of storing the translation page size (i.e., 4KiB, 2MiB, and 1GiB) for both VS- and G-stages into the same TLB entry and permissions access bits for each stage.

### D. Virtualization-aware TLBs (vTLB)

The CVA6 MMU microarchitecture has two small, fully associative TLB: data (L1 DTLB) and instructions (L1 ITLB). Both TLBs support a maximum of 16 entries and fully implement the flush instructions, i.e., *sfence*, including filtering by ASID and virtual address. To support nested translation, we modified the L1 DTLB and ITLB microarchitecture to support two translation stages, including access permissions and VMIDs. Each TLB entry holds both VS-Stage and G-Stage PTE and respective permissions. The lookup logic is performed using the merged final translation size from both stages, i.e., if the VS-stage is a 4KiB and the G-stage is a 2MiB translation, the final translation would be a 4KiB. This is probably one of the major drawbacks of having both the VS-stage and G-stage stored together. For instance, hypervisors supporting superpages/hugepages use the 2MiB page size to optimize the MMU performance. Although this significantly reduces the PTW walk time, if the guest uses a 4KiB page size, the TLB lookup would not benefit from superpages since the translation would be stored as a 4KiB page size translation. A possible alternative would be having separated TLBs for each stage. We argue that this solution would have three major drawbacks: (i) negative impact on performance as it would be possible to search in G-Stage TLB after the VS-Stage TLB lookup (i.e., increase the TLB hit time penalty by a factor of 2); (i) less hardware reuse, i.e., the G-Stage TLB is only used when a guest is running and G-stage is active; and (iii) more impact on the area and energy, the CVA6 TLBs are fully combinational circuits with a significant impact on the area, energy, and timing. For all the above reasons, we decided to keep the VS-stage and G-stage translations in a single TLB entry.. Finally, the TLB also supports VMID tags allowing hypervisors to perform a more efficient TLB management using per-VMID flushes and avoiding full TLB flush on a VM context switch. Finally, the TLB also allows flushes by guest physical address, i.e., hypervisor fence instructions (*hfence.vvma/gvma*) are fully supported.

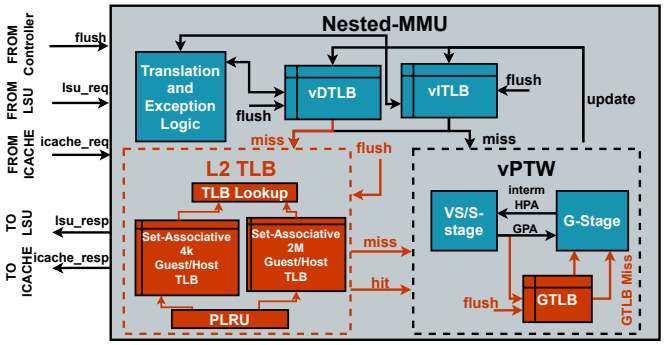


Fig. 3. MMU microarchitectural enhancements: high-level overview. Additions to the original MMU design highlighted in orange.

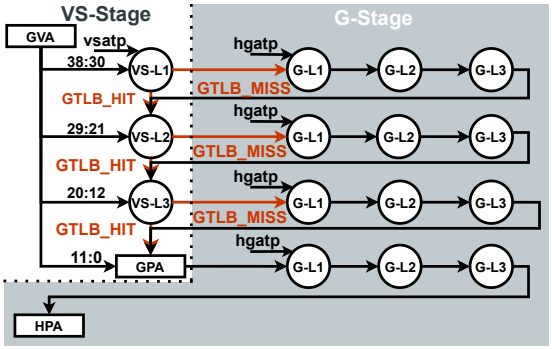


Fig. 4. RISC-V nested-MMU walk example for the SV39x4 scheme.

### E. Microarchitectural extension #1 - GTLB

A full nested table walks for the *Sv39* scheme can take up to 15 memory accesses, five times more than a standard S-stage translation. This additional burden imposes a higher TLB miss penalty, resulting in a (measured) overhead of up to 13% on the functional performance (execution cycles) (comparing the baseline virtualization implementation with the vanilla CVA6). To mitigate this, we have extended the CVA6 microarchitecture with a GTLB in the nested-PTW module to store intermediate GPA to HPA translations, i.e., VS-Stage non-leaf PTE guest physical address to host physical addresses translation. Figure 3 illustrates the modifications required to integrate the GTLB in the MMU microarchitecture. The GTLB structure aims at accelerating VS-stage translation by skipping each nested translation during the VS-stage page table walk. Figure 4 presents a 4KiB page size translation process featuring a GTLB in the PTW. Without the GTLB, each time the guest forms the non-leaf physical address PTE pointer during the walk, it needs: (i) to translate it via G-stage, (ii) read the next level PTE value from memory, and (iii) resume the VS-stage translation. When using superpages (2MiB or 1GiB), there are fewer translation walks, reducing the performance penalty. With a simple hardware structure, it is possible to mitigate such overheads by keeping those G-stage translations cached while avoiding unnecessary cache pollution and nondeterministic memory accesses.

**GTLB microarchitecture.** The GTLB follows a fully-associative design with support for all *Sv39* superpage sizes (4KiB, 2MiB, and 1GiB). Each entry is searched in parallel during the TLB lookup, and each translation can be stored on any TLB entry. The replacement policy evicts the least recently used entry using a pseudo-least recently used (PLRU) algorithm already implemented on the CVA6 L1 TLBs. The maximum number of entries that GTLB can hold simultaneously ranges from 8 to 16. The flush circuit fully supports hypervisor fence instruction (*HFENCE.GVMA*), including filtering flushed TLB entries by VMIDs and virtual address. It is worth mentioning that the GTLB implementation reused the already-in-place L1 TLB design and modified it to store only G-stage-related translations. We map TLB entries to flip-flops due to their reduced number.

### F. Microarchitectural extension #2 - L2 TLB

Currently, the vanilla CVA6 MMU features only a small separate L1 instruction and data TLBs, shared by guests and the hypervisor. The TLB can merge VS and G stage translations in one single entry, where the stored translation page size must be the minimum of the two stages. This may improve hardware reuse and save some additional search cycles, as no separated TLBs for each stage are required. However, as explained in Subsection III-D, one major drawback of this approach is that if the hypervisor or the guest uses superpages, the TLB would not benefit from them if the pages differ in size, i.e., there would be less TLB coverage than expected. This would result in more TLB misses and page-table walks and, naturally, a significant impact on the overall performance. To deal with the increased TLB coverage and TLB miss penalty caused by G-stage overhead, as well as with the inefficiency arising from the mismatch between translation sizes, we have augmented the CVA6 MMU with a large set-associative private unified L2 TLB as illustrated in Figure 3.

**L2 TLB microarchitecture.** The TLB stores each translation size in different structures to simplify the look-up logic. The L2 TLB follows a set-associative design with translation tags and data stored in SRAMs. As implemented in the GTLB, the replacement policy is also PLRU. To speed up the search logic, the L2 TLB look-ups and the PTW execute in parallel, thus not affecting the worst-case L1 TLB miss penalty and optimizing the L2 miss penalty. Moreover, the L2 TLB performs searches in parallel for each page size (4KiB or 2MIB), i.e., each page size translation is stored on different hardware structures with independent control and storage units. SFENCE and HFENCE instructions are supported, but a flush signal will flush all TLB entries, i.e., no filtering by VMIDs or ASIDs. We implemented the L2 TLB controller using a 4 state FSM. First, in *Flush* state, the logic encompasses walking through the entire TLB and invalidating all TLB entries. Next, in the *Idle* state, the FSM waits for a valid request or update signal from the PTW module. Third, in the *Read* state, the logic performs a read and tag comparison on the TLB set entries. If there is a hit during the look-up, it updates the correct translation and hit signals to the PTW. If there is no look-up hit, the FSM goes to the IDLE state and waits for a new request. Finally, in the *Update* state, we update the TLB upon a PTW update.

### G. Sstc Extension

Timer registers are exposed as MMIO registers (*mtime*); however, the Sstc specification defines that *stimecmp and vstimecmp* are hart CSRs. Thus, we exposed the time value

TABLE III
DESIGN SPACE EXPLORATION CONFIGURATIONS.

| Module | Parameter | Configuration | | |
|---|---|---|---|---|
| | | #1 | #2 | #3 |
| L1 TLB | size entries | 16 | 32 | 64 |
| GTLB | size entries | 8 | 16 | — |
| L2 TLB | page size | 4KiB | 2MiB | 4KiB+2MiB |
| | associativity | 4 | 8 | — |
| | size entries for 4KiB | 128 | 256 | — |
| | size entries for 2MiB | 32 | 64 | — |
| SSTC | status | enabled | disable | — |

to each hart via connection to the CLINT. We also added the ability to enable and disable this extension at S/HS-mode and VS-mode via *menvcfg.STCE* and *henvcfg.STCE* bits, respectively. For example, when *menvcfg.stce* is 0, an S-mode access to stimecmp will trigger an illegal instruction. The same happens for VS-mode, when *henvcfg.stce* is 0 (throwing a virtual illegal instruction exception). The Sstc extension does not break legacy timer implementations, as software that does not support the Sstc extension can still use the standard SBI interface to set up a timer.

## IV. Design Space Exploration: Evaluation

In this section, we discuss the conducted design space exploration (DSE) evaluation. The system under evaluation, the DSE configuration, the followed methodology, as well as benchmarks and metrics are described below. Each subsection then focuses in assessing the functional performance speedup per the configuration of each specific module: (i) L1 TLB (Section IV-A); (ii) GTLB (Section IV-B); (iii) L2 TLB (Section IV-C); and (iv) Sstc (Section IV-D).

**System and Tools.** We ran the experiments on a CVA6 single-core SoC featuring a 32KiB DCache and 16KiB ICache, targeting the Genesys2 FPGA at 100MHz. The software stack encompasses the (i) OpenSBI (version 1.0) (ii) Bao hypervisor (version 0.1), and Linux (version 5.17). We compiled Bao using SiFive GCC version 10.1.0 for riscv64 baremetal targets and OpenSBI and Linux using GCC version 11.1.0 for riscv64 Linux targets. We used Vivado version 2020.2 to synthesize the design and assess the impact on FPGA hardware resources.

**DSE Configurations.** The DSE configurations are summarized in Table III. For each module, we selected the parameters we wanted to explore and their respective configurations. For instance, for the L1 TLB we fixed the number of entries as the design parameter and 16, 32, and 64 as possible configurations. Taking all modules, parameters, and configurations into consideration, it is possible to achieve up to 288 different combinations. Due to the existing limitations in terms of time and space, we carefully selected and evaluated 23 out of these 288 configurations.

**Methodology.** We focus the DSE evaluation on functional performance (execution cycles). We collected 1000 samples and computed the average of 95 percentile values to remove any outliers caused by some sporadic Linux interference. We normalized the results to the reference baseline execution (higher values translate to higher performance speedup) and added the absolute execution time on top of the baseline bar. Our baseline hardware implementation corresponds to a system with the CVA6 with hardware virtualization support but without the proposed microarchitectural extensions (e.g., GTLB, L2 TLB). We have also collected the post-synthesis hardware utilization; however, we omit the results due to lack of space (although we may occasionally refer to them during the discussion of the results in this section).

**Benchmarks.** We used the Mibench Embedded Benchmark Suite. The Mibench is an embedded application benchmark widely used in mixed-critically systems to assess the performance [23], [31]. The Mibench incorporates a set of 35 application benchmarks grouped into six categories, targeting different embedded market segments. We focus our evaluation on the automotive subset. The automotive suite encompasses 3 high memory-intensive benchmarks (*qsort*, *susan corners* and *susan edges*) that exercise many components across the memory hierarchy (e.g. MMU, cache, and memory controller). To complement our evaluation, we ran San Diego Vision Benchmark [32] in the selected designs for the PPA analysis. San Diego Vision is a computer vision benchmark widely used to evaluate embedded and mixed-criticality systems [33], [34]. It runs a full range of vision applications (e.g., motion tracking (*tracking*)) using multiple datasets with different sizes. Instructions to run our experiments with Mibench and San Diego Vision can be found here[5].

### A. L1 TLB

In this subsection, we evaluate the functional performance for a different number of L1 TLB entries, i.e., 16, 32, and 64.

**L1 TLB Setup.** To assess the L1 TLB functional performance speedup, we ran the full set of benchmarks for three different setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); and (iii) Linux virtual/hosted execution for the *cva6-64* (*hosted-64*).

**L1 TLB Performance Speedup.** Figure 5 shows the assessed results. All results are normalized to the *baseline* execution. Several conclusions can be drawn. Firstly, as expected, the *hosted-64* is the best-case scenario with an average performance speedup increase of 3.6%. We can observe a maximum speedup of 7% in the *susanc* (small) benchmark and a minimum speedup of 2% in the *bitcount* (large) benchmark. However, although not explicitly shown in this paper, to achieve these results there is an associated impact of 50% increase in the FPGA resources. We expected these results because the L1 TLB is a fully-associative TLB implemented as a full combinational circuit, and the CVA6 design is not optimized for this particular FPGA architecture. Secondly, the *hosted-32* configuration increases the performance by a minimum of 1% (e.g., basicmath large) and a maximum of 4% (e.g., *susanc* small), at a cost of about 15%-17% in the area (FPGA). Finally, we can conclude that increasing the CVA6 L1 TLB size to 32 entries presents the most reasonable trade-off between functional performance speedup and hardware cost.

[5]https://github.com/ninolomata/bao-cva6-guide/tree/cva6-evaluation

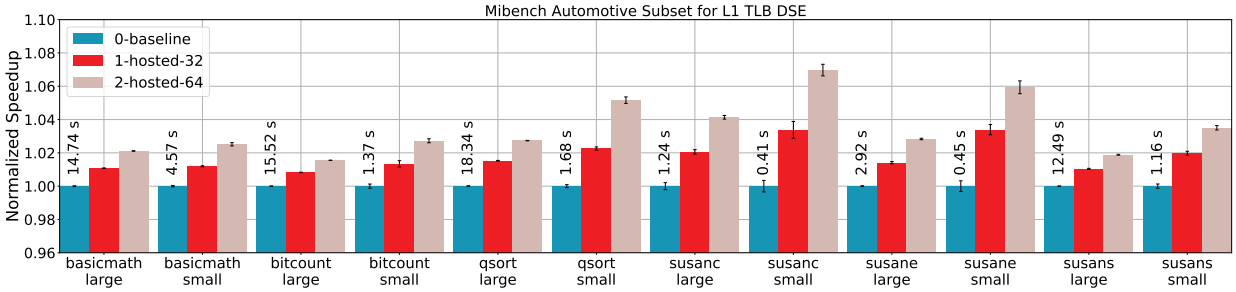


Fig. 5. Mibench results for L1 TLB design space exploration evaluation.

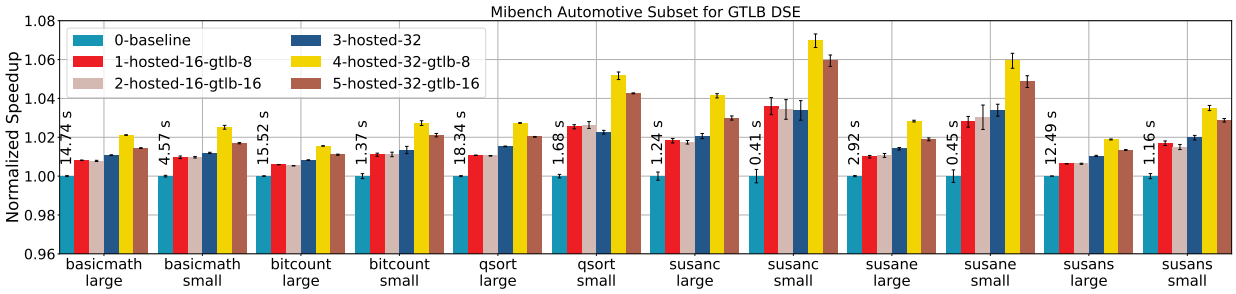


Fig. 6. Mibench results for GTLB design space exploration evaluation.

### B. GTLB

In this subsection, we assess the GTLB impact on functional performance. We evaluate this for a different number of GTLB entries (8 and 16) and L1 TLB entries (16 and 32).

**GTLB Setup.** To assess the GTLB functional performance speedup, we ran the full set of benchmarks for seven distinct setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); (iii) Linux virtual/hosted execution for the *cva6-16-gtlb-8* (*hosted-16-gtlb-8*); (iv) Linux virtual/hosted execution for the *cva6-32-gtlb-8* (*hosted-32-gtlb-8*); (v) Linux virtual/hosted execution for the *cva6-16-gtlb-16* (*hosted-16-gtlb-16*); and (vi) Linux virtual/hosted execution for the *cva6-32-gtlb-16* (*hosted-32-gtlb-16*).

**GTLB Performance Speedup.** Mibench results are depicted in Figure 6. We normalized results to *baseline* execution. We highlight a set of takeaways. Firstly, the *hosted-16-gtlb-8* and *hosted-16-gtlb-16* scenarios have similar results across all benchmarks with an average percentage performance speedup of about 2%; therefore, there is no significant improvement from configuring the GTLB with 16 entries over 8 when the L1 TLB has 16 entries. Secondly, the *hosted-32-gtlb-8* setup presents the best performance, i.e., a maximum performance increase of about 7% for the *susanc* (small) benchmark; however, at a non-negligible overall cost of 20% in the hardware resources. Surprisingly, the hosted execution with 32 entries and a GTLB with 16 entries (*hosted-32-gtlb-16*) perform slightly worst compared with an 8 entries GTLB (*hosted-32-gtlb-8*). We have also collected microarchitectural hardware events on the CVA6. We noticed a slight increase in data cache misses for the *hosted-32-gtlb-16* compared to the *hosted-32-gtlb-8* of roughly 10% in some benchmarks (e.g., *susanc-large*), which explain why the GTLB with 16 entries performs worse than the GTLB with 8 entries. For instance, for the *susane* (large) benchmark the *hosted-16-gtlb-8* setup achieves a 6% performance increase while the *hosted-16-gtlb-16* only 5%. Finally, the *hosted-32* achieves a performance increase in line with *hosted-16-gtlb8* and *hosted-16-gtlb16* configurations.

TABLE IV
L2 TLB DESIGN SPACE EXPLORATION CONFIGURATIONS.

| Configuration | L1 TLB | GTLB | L2 TLB |
|---|---|---|---|
| cva6-16 | 16 entries | — | — |
| cva6-16-gtlb8 | 16 entries | 8 entries | — |
| cva6-16-l2-1 | 16 entries | 8 entries | 4KiB - 128, 4-ways |
| cva6-16-l2-2 | 16 entries | 8 entries | 2MiB - 32, 4-ways |
| cva6-16-l2-3 | 16 entries | 8 entries | 4KiB - 128, 4-ways<br>2MiB - 32, 4-ways |
| cva6-32-gtlb8 | 16 entries | 8 entries | — |
| cva6-32-l2-1 | 32 entries | 8 entries | 4KiB - 128, 4-ways |
| cva6-32-l2-2 | 32 entries | 8 entries | 4KiB - 128, 8-ways |
| cva6-32-l2-3 | 32 entries | 8 entries | 4KiB - 256, 4-ways |
| cva6-32-l2-4 | 32 entries | 8 entries | 4KiB - 256, 8-ways |
| cva6-32-l2-5 | 32 entries | 8 entries | 2MiB - 32, 4-ways |
| cva6-32-l2-6 | 32 entries | 8 entries | 2MiB - 32, 8-ways |
| cva6-32-l2-7 | 32 entries | 8 entries | 2MiB - 64, 4-ways |
| cva6-32-l2-8 | 32 entries | 8 entries | 2MiB - 64, 8-ways |
| cva6-32-l2-9 | 32 entries | 8 entries | 4KiB - 128, 4-ways<br>2MiB - 32, 4-ways |
| cva6-32-l2-10 | 32 entries | 8 entries | 4KiB - 256, 4-ways<br>2MiB - 64, 4-ways |

### C. L2 TLB

In this subsection, we assess the L2 TLB impact on functional performance. We conduct several experiments focusing on two main L2 TLB configuration parameters: (i) multi-page size support (4KiB or 2MiB or both) and (ii) TLB associativity. Furthermore, we also evaluated the L2 TLB impact in combination with different L1 TLB entries (16 and 32) and the GTLB with 8 entries. Table IV summarizes the design configurations.

**Multi-page Size Support Setup.** To assess the performance speedup for the L2 TLB multi-page size, we have carried

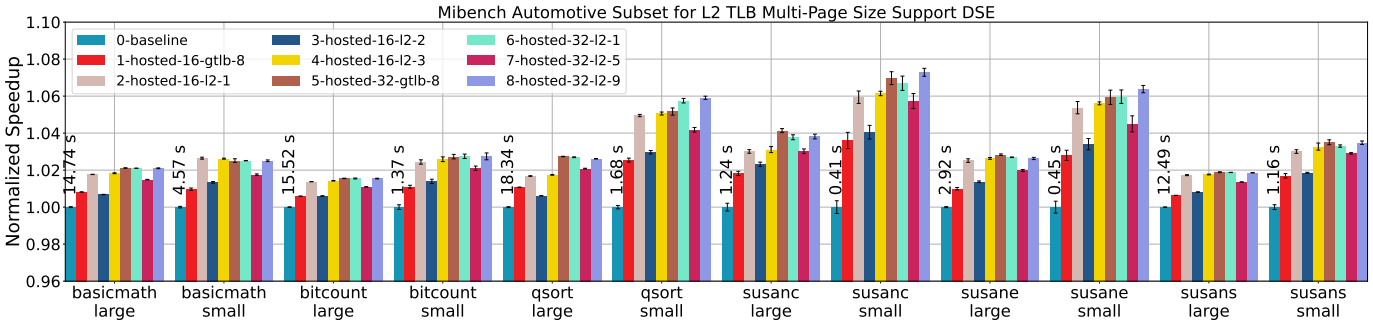


Fig. 7. Mibench results for L2 TLB multi-page size support design space exploration evaluation.

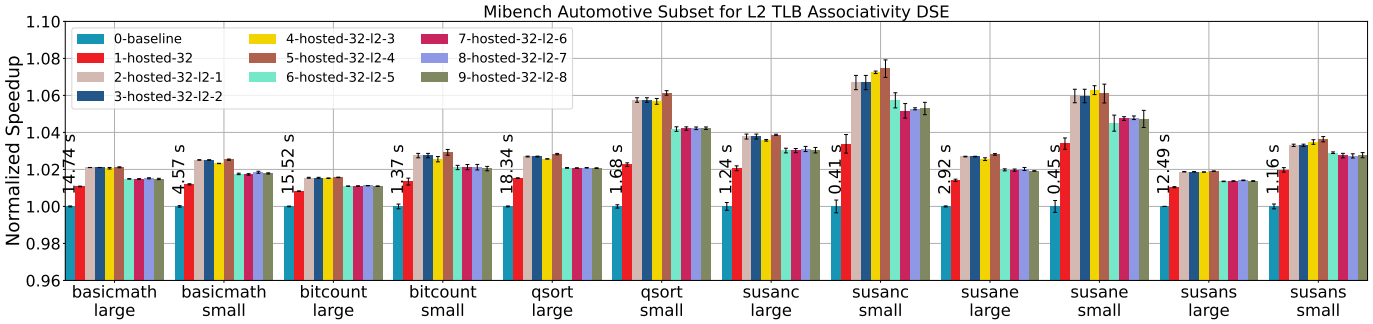


Fig. 8. Mibench results for L2 TLB associativity design space exploration evaluation.

out a set of experiments to measure the impact of: (i) 4KiB page size (configurations *cva6-16-l2-1*,*cva6-32-l2-1*); (ii) 2MiB page size (configurations *cva6-16-l2-2*, *cva6-32-l2-5*); (iii) both 4KiB and 2MiB page sizes (configurations *cva6-16-l2-3*, *cva6-32-l2-9*); and (iv) increase the L1 TLB capacity. For each configuration, we run the Mibench benchmarks for nine different scenarios: (i) Linux virtual/hosted execution for the baseline *cva6-l1-16* (*baseline*); (ii) Linux native execution for the *cva6-16* (*bare*); (ii) Linux virtual/hosted execution for the *cva6-16-gtlb8* (*hosted-16-gtlb8*); (iii) Linux virtual/hosted execution for the *cva6-16-l2-1* (*hosted-16-l2-1*); (iv) Linux virtual/hosted execution for the *cva6-16-l2-2* (*hosted-16-l2-2*); (v) Linux virtual/hosted execution for the *cva6-16-l2-3* (*hosted-16-l2-3*); (vi) Linux virtual/hosted execution for the *cva6-32-gtlb8* (*hosted-32-gtlb8*); (vii) Linux virtual/hosted execution for the *cva6-32-l2-1* (*hosted-32-l2-1*); (viii) Linux virtual/hosted execution for the *cva6-32-l2-5* (*hosted-32-l2-5*); and (ix) Linux virtual/hosted execution for the *cva6-32-l2-9* (*hosted-32-l2-9*).

**Multi-page Size Support Performance.** Mibench results are depicted in Figure 7. All results were normalized to *baseline* execution. Based on Figure 7, we can extract several conclusions. First, configurations that support only 2MiB page size, i.e., *hosted-16-l2-2* and *hosted-16-l2-5*, present little or almost no performance improvement compared with the hardware configurations including only the GTLB (i.e., *hosted-16-gtlb-8* and *hosted-16-gtlb-8*). Second, supporting 4KiB page sizes causes a noticeable performance speedup for the L1 TLB with 16 entries, especially in memory-intensive benchmarks, such as *qsort* (small), *susanc* (small) and *susane* (small). The main reason that justifies this improvement lies in the fact that Bao is configured to use 2MiB superpages; however, the Linux VM uses mostly 4KiB page size, i.e., most of the translations are 4KiB. From a different perspective, we observe similar results for the *hosted-16-l2-1* and *hosted-32-gtlb-8*, with few exceptions in some benchmarks (e.g., *susane* (large), *susanc* (large) and *qsort* (large)). Moreover, for an L1 TLB configured with 32 entries, the average performance increase is roughly equal in less memory-intensive benchmarks (e.g. *basicmath* and *bitcount*). This is explained by the reduced number of L1 misses, due to its larger capacity which leads to fewer requests to the L2 TLB. Finally, we observe minimal speedup improvements on several benchmarks (e.g., *susans*) when supporting both 4KiB and 2MiB (*hosted-16-l2-3* and *hosted-32-l2-9*).

**TLB associativity Setup.** To evaluate the performance speedup for the L2 TLB associativity, we select eight designs from previous experiments and modified the following parameters: (i) the number of 4KiB entries (128 or 256) and 2MiB (32 or 64) entries for the L2 TLB; and (ii) the L2 TLB associativity, by configuring the L2 TLB in the 4-way or 8-way scheme. For each configuration, we have run the selected benchmarks for ten configurations: (i) Linux virtual/hosted execution for the baseline *cva6-l1-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); (iii) Linux virtual/hosted execution for the *cva6-32-l2-1* (*hosted-32-l2-1*); (iv) Linux virtual/hosted execution for the *cva6-32-l2-2* (*hosted-32-l2-2*); (v) Linux virtual/hosted execution for the *cva6-32-l2-3* (*hosted-32-l2-3*); (vi) Linux virtual/hosted execution for the *cva6-32-l2-4* (*hosted-32-l2-4*); (vii) Linux virtual/hosted execution for the *cva6-32-l2-5* (*hosted-32-l2-5*); (viii) Linux virtual/hosted execution for the *cva6-32-l2-6* (*hosted-32-l2-6*); (ix) Linux virtual/hosted execution for the *cva6-32-l2-7* (*hosted-32-l2-7*); and (x) Linux virtual/hosted execution for the *cva6-32-l2-7* (*hosted-32-l2-7*).

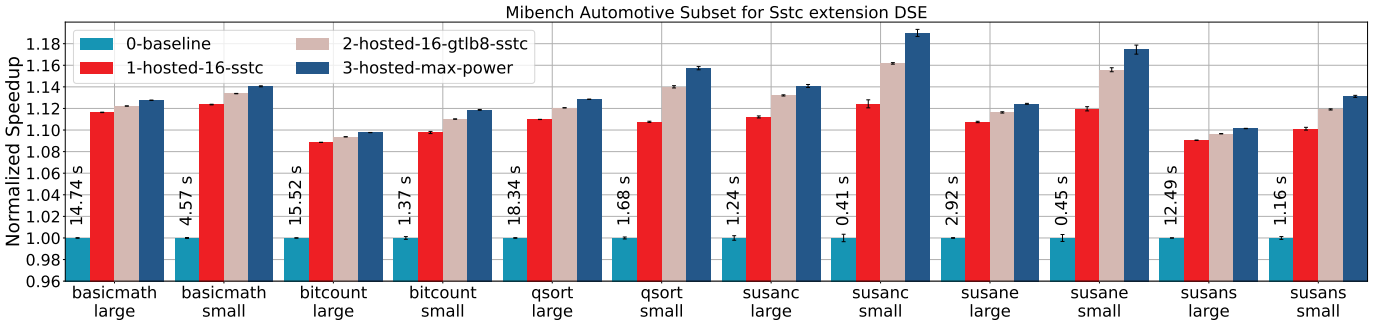


Fig. 9. Mibench results for Sstc extension design space exploration evaluation.

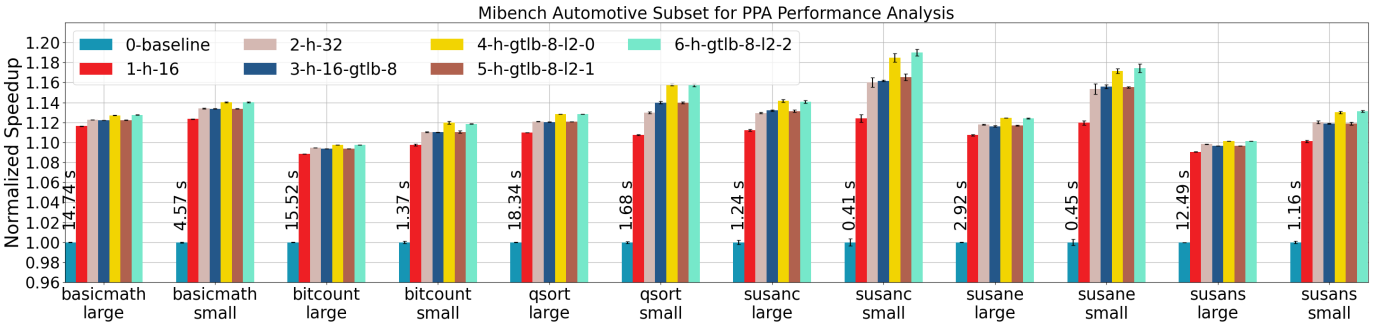


Fig. 10. Summary of MiBench functional performance results for selected configurations.

TABLE V
SSTC DESIGN SPACE EXPLORATION CONFIGURATIONS.

| Configuration | L1 TLB | GTLB | L2 TLB | SSTC |
|---|---|---|---|---|
| cva6-sstc | 16 | — | — | enabled |
| cva6-sstc-gtlb8 | 16 | 8 | — | enabled |
| cva6-max-power | 16 | 8 | 4KiB - 128, 4-ways<br>2MiB - 32, 4-ways | enabled |

**TLB associativity Performance.** The results in Figure 8 demonstrate that there is no significant improvement in modifying the number of sets and ways in 2MiB page size configurations (i.e., *hosted-32-l2-5* to *hosted-32-l2-8*). Additionally, we found that: (i) increasing the associativity from 4 to 8 or doubling the capacity in low memory-intensive benchmarks (e.g., *bitcount*) had little impact on functional performance; and (ii) memory-intensive benchmarks (e.g., *susanc* (small)) normaly present a speedup when increasing the number of entries for the same associativity (although with a larger standard deviation). For instance, in the *susanc* (small) benchmark, we observer a performance speedup from 6% to 7% when the page support was 4KiB with 128 and 256 capacity organized into a 4-way per set (*hosted-32-l2-1* and *hosted-32-l2-3*).

### D. Sstc Extension

In this subsection, we assess the Sstc extension impact on performance. We assess the Sstc performance speedup for a few configurations exercised in the former subsections. Table V summarizes the design configurations.

**Sstc Extension Setup.** To assess the Sstc extension performance speedup, we ran the full set of benchmarks for four different setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for *cva6-sstc* (*hosted-16-sstc*); (iii) Linux virtual/hosted execution for *cva6-gtlb8-sstc* (*hosted-16-gtlb8-sstc*); and (iv) Linux virtual/hosted execution for *cva6-max-power* (*hosted-max-power-sstc*).

**Sstc Extension Performance.** The results depicted in Figure 9 show that for the *hosted-16-sstc* scenario, there is a significant performance speedup. For example, in the memory-intensive *qsort* benchmark, the performance speedup is 10.5%. Timer virtualization is a major cause of pressure on the MMU subsystem due to the frequent transitions between the Hypervisor and Guest. The "hosted-max-power" scenario performs the best, with an average performance increase of 12.6%, ranging from a minimum of 10% in the *bitcount* benchmark to a maximum of 19% in memory-intensive benchmarks such as the *susanc*. Finally, in less memory-intensive benchmarks, e.g., *basicmath* (large) and *bitcount* (large), the *hosted-16-gtlb8-sstc* scenario performs similarly to the *hosted-max-power* scenario, but with significantly less impact on hardware resources. For example, in the *basicmath* (large) benchmark, there is a negligible difference of 0.5 % comparing the *hosted-16-gtlb8-sstc* with the *hosted-max-power* scenario.

### E. Designs for PPA Analysis Selection

We selected six configurations for the PPA analysis (see Table VI) based on the functional performance setup results summarized in Figure 10. We also ran San Diego Vision Benchmark to evaluate the selected designs with different workloads.

**San Diego Vision Workloads Results.** We ran the San Diego Vision Benchmark for the configurations elected for the PPA analysis. The results depicted in Figure 11 are in line with the results collected for the Mibench benchmark (Figure 10). First, for the *h-16* scenario, there is an average performance speedup of 10%. Second, the *h-32* outperforms *h-16-gtlb-8* in some benchmarks with few exceptions. For instance, for the

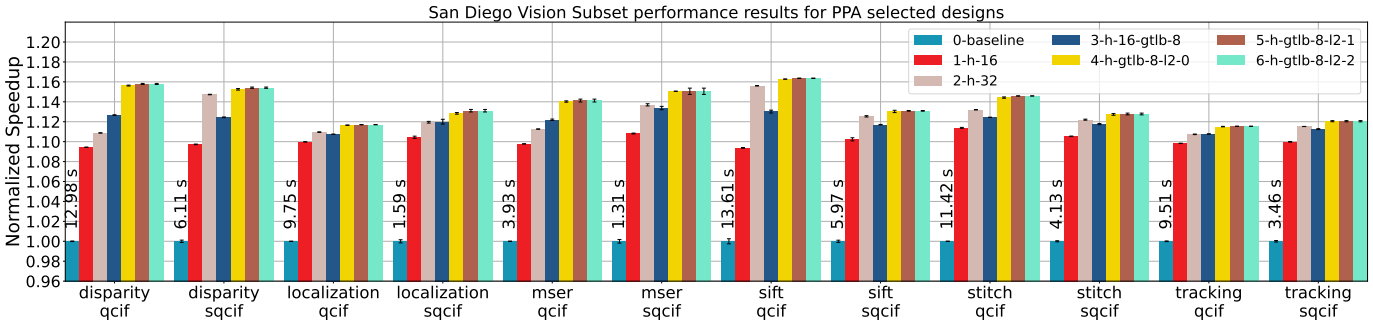


Fig. 11. San Diego Vision Benchmark functional performance results for PPA analysis selected configurations.

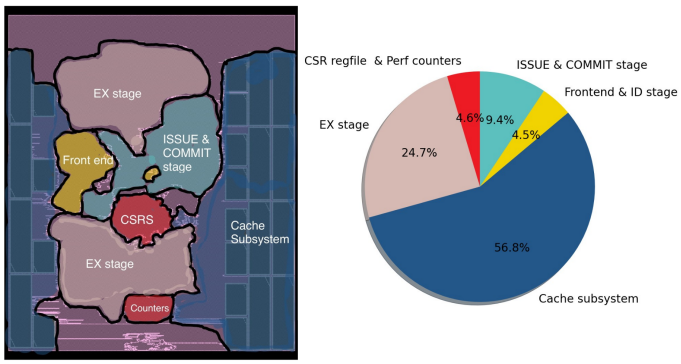


Fig. 12. Baseline CVA6 layout 0.75mmx0.65mm

*disparity qcif*, there is a slight difference of 2%. Third, *h-16-gtlb-8* has reasonable results with an average performance increase of about 12% (maximum of 14% and minimum of 10%). Finally, for the *h-gtlb-8-l2-0*, *h-gtlb-8-l2-1*, and *h-gtlb-8-l2-2* configurations, the performance is identical, with a pattern ranging a maximum of 16% and a minimum of 11%.

## V. Physical Implementation

### A. Methodology

Based on the functional performance results discussed in Section IV and summarized in Figure 10, we select six configurations to compare with the baseline implementation of CVA6 with hardware virtualization support. Table VI lists the hardware configurations under study. We implemented these configurations in 22 nm FDX technology from Global Foundries down to ready-for-silicon layout to reliably estimate their operating frequency, power, and area.

To support the physical implementation and the PPA analysis, we used the following tools: (i) Synopsys Design Compiler 2019.03 to perform the physical synthesis; (ii) Cadence Innovus 2020.12 for the place & route; (iii) Synopsys PrimeTime 2020.09 for the power analysis - extracting value change dump (VCD) traces on the post layout; and (iv) Siemens Questasim 10.7b to design the parasitics annotated netlist. Figure 12 shows the layout of CVA6, featuring an area smaller than $0.5\text{mm}^2$.

### B. Results

We fix the target frequency to 800MHz in the worst corner (SSG corner at 0.72 V, -40/125 °C). All configurations manage to reach the target frequency. We then compare the area and power consumption while running a dense 16x16 FP matrix multiplication at 800MHz with warmed-up caches (TT corner, 25 °C). Leveraging the extracted power measurements and the functional performance on the Mibench benchmarks, we further obtain the relative energy efficiency.

| | SSTC support | I/DTLB #entries | 8-entries GTLB | 4k L2 TLB | 2MB L2 TLB |
|---|---|---|---|---|---|
| 0-vanilla | x | 16 | x | x | x |
| 1-h-16 | ✓ | 16 | x | x | x |
| 2-h-32 | ✓ | 32 | x | x | x |
| 3-h-gtlb-8 | ✓ | 16 | ✓ | x | x |
| 4-h-gtlb-8-l2-0 | ✓ | 16 | ✓ | ✓ | x |
| 5-h-gtlb-8-l2-1 | ✓ | 16 | ✓ | x | ✓ |
| 6-h-gtlb-8-l2-2 | ✓ | 16 | ✓ | ✓ | ✓ |

TABLE VI
7 SELECTED CONFIGURATIONS FOR PPA ANALYSIS IN GF22NM

Figure 13 depicts the PPA results. Figure 13(a) shows that the Sstc extension has negligible impact on the area. In fact, as expected, the MMU is the microarchitectural component with a higher impact on power and area. Figure 13(b) highlights the MMU area comparison. The configuration with 32 entries doubles the ITLB and DTLB area, while the other configurations have little to no impact on the ITLB and DTLB; they, at most, add the GTLB and the L2-TLB modules on top of the existing MMU configuration. Figure 13(c) shows the measured power consumption. We observe that increasing the number of L1 TLB entries from 16 to 32 (2-h-32) increases the power by 31%, while the other configurations impact less than 5% on power compared with the vanilla CVA6. Figure 14 shows the relative energy efficiency on the MiBench benchmarks. The second configuration (2-h-32) is less energy efficient than the baseline since the performance gain ($\leq 15\%$) is smaller than the power increase ($\geq 30\%$). It is therefore excluded in the graph analysis. On the other hand, all the other configurations increase the energy efficiency up to 16%.

Figure 13(d) plots the measured power consumption of the energy-efficient configurations against the average performance improvement on the Mibench benchmarks. The SSTC extension alone brings an average performance improvement of around 10% with a negligible power increase. At the same time, the explored MMU configurations can offer a clean trade-off between performance and power. The most expensive configuration can provide an extra performance of 4% gain for a 4.47% power increase. Lastly, the hardware configuration including the Sstc support and a GTLB with 8 entries (3-h-16-gtlb-8), is the most energy-efficient one, with the highest ratio between performance and power increase.

In conclusion, we can argue that the hardware configuration with Sstc support and a GTLB with 8 entries (3-h-16-gtlb-8)

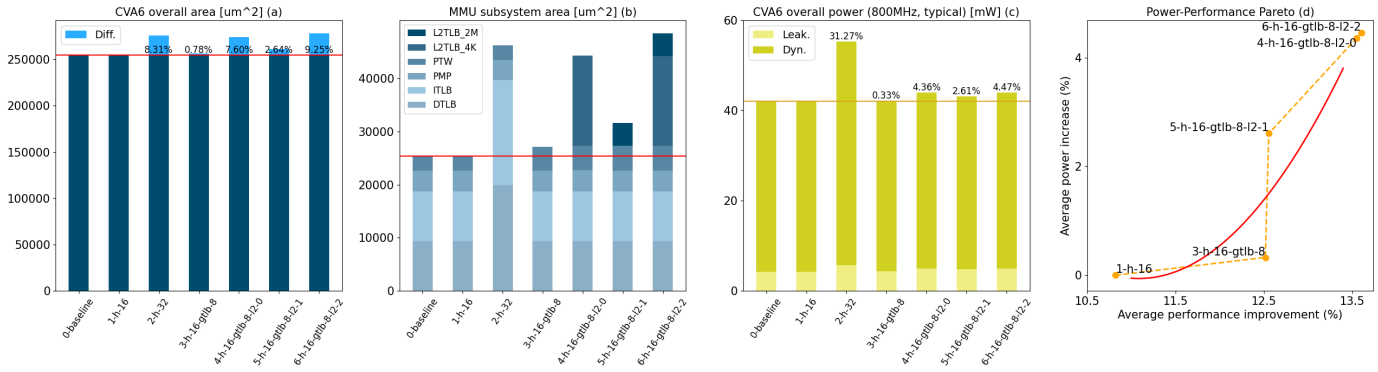


Fig. 13. Area and Power results

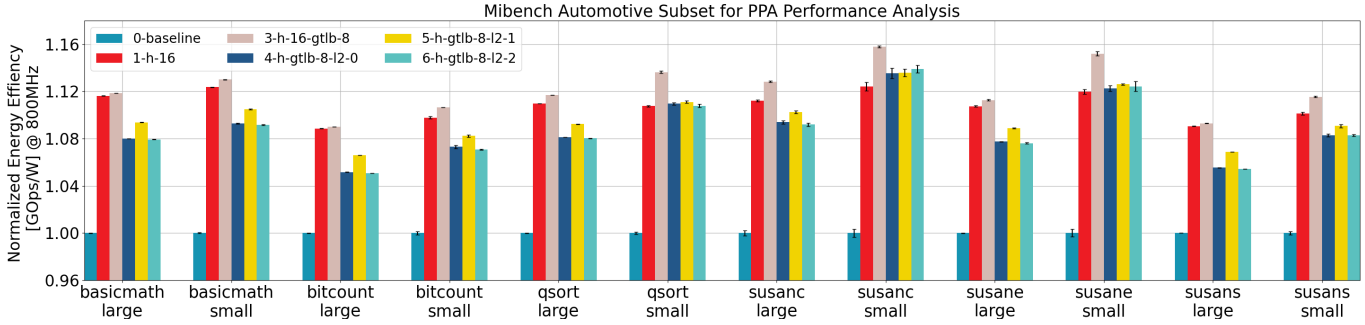


Fig. 14. Energy efficiency results.

is the optimal design point since we can achieve a functional performance speedup of up to 16% (approx. 12.5% on average) at the cost of 0.78% in area and 0.33% in power.

## VI. Discussion

**Performance Evaluation.** We carried out an extensive functional performance evaluation for multiple FPGA design configurations in a single-core environment. As part of future work, we plan and encourage the evaluation of additional configurations and software stacks. Possible directions include (i) experimenting with other hypervisors like XVisor [18] in different configurations (e.g., having two guests running in a single core); (ii) running other commercial benchmarks like SPEC17; and (iii) shifting to multi-core environments, e.g., available in frameworks such as OpenPiton [35].

**System-level Virtualization.** The RISC-V ISA has reached a new state of maturity, and some of its previously existing gaps [23] (e.g., no standard interrupt controller with interrupt virtualization support) are already addressed. The Advanced Interrupt Architecture (AIA) [30] specification has been ratified, and it provides preliminary support for interrupt virtualization using Message Signaled Interrupts (MSI). Hypervisors can leverage the AIA to reduce interrupt latency to guests and improve overall performance. As part of our ongoing efforts, we are working on providing an open-source reference implementation of the AIA IPs. The RISC-V I/O memory management unit (IOMMU) [6] is also now ratified. Fundamentally, the IOMMU protects memory accesses from DMA-capable devices while guaranteeing isolation between VMs and the hypervisor. For instance, in Bao, all devices are assigned in a one-to-one scheme to a single guest using pass-through device mechanisms, which means Bao entirely relies on the IOMMU to do this. As part of our ongoing efforts, we are working on providing open-source reference implementations of the AIA IPs and the IOMMU IP.

**Memory Subsystems Improvements.** From an MMU microarchitectural point of view, there is ample room for improvement. We demonstrated that the CVA6 memory subsystem is the major cause of performance degradation, and further improvements are of utmost importance. For instance, we could redesign the CVA6 MMU to support TLB coalescing by augmenting the PTE to support NAPOT pages (Snapot extension). Although we have accelerated the PTW with dedicated TLB for the second stage, a few memory accesses to the memory hierarchy are still needed, i.e., side-effects are still expected at the cache level, and memory bandwidth at the system interconnect. One could explore using a PTE cache to store PTE entries close to the PTW and thus reduce the number of memory accesses and cache pollution. From the platform standpoint, RISC-V rich environment offers extensions that can benefit virtualized environments. We already discussed the standard cache management instructions, i.e., CMOs, defined in the Zicbom and Zicboz extensions. Hypervisors like Bao depend upon such instructions to implement partitioning mechanisms such as cache coloring, which otherwise would have to flush caches using firmware *ecalls*. Finally, this work targeted a single-core platform with no Last-Level of Cache (LLC). Hence, the effectiveness of such mechanisms is yet to be proved in a multi-core environment where inter-core interference is visible throughout the memory hierarchy. An all-new class of security challenges is present, e.g., timing side-channels [36].

[6]https://github.com/riscv-non-isa/riscv-iommu

TABLE VII
CVA6 ALIKE RISC-V CORES WITH HYPERVISOR EXTENSION SUPPORT FEATURES: ✓- SUPPORTED; X - NOT SUPPORTED; AND ? - NO INFORMATION AVAILABLE.

| Processor | Pipeline | Priv. Hyp. | SSTC | 2D-MMU | L1 Cache | L1 TLB | L2 TLB | PTW Opts | Status |
|---|---|---|---|---|---|---|---|---|---|
| Rocket | 5-stage in-order | v0.6 | X | SV39x4 | 32 KiB, I/DCache | set-assoc. 4KiB I/DTLB or full-assoc. superpages I/DTLB, 4-entries (configurable) | dir.-mapped, (configurable size) | PTE Cache | Done |
| NOEL-V | 7-stage dual-issue in-order | v0.6 | ✓ | SV32x2 SV39x4 | 32 KiB I/DCache | I/DTLB, (configurable) | X | hTLB | Done |
| Chromite | 6-stage in-order | v1.0 | ? | SV32x4 SV39x4 SV48x4 SV57x4 | 32 KiB I/DCache | set-assoc. split I/DTLB, (configurable page sizes support) or full-assoc. I/DTLB, ?-entries (def. 4) | X | X | WIP |
| Shaktii C Class | 5-stage in-order | v1.0 | ? | SV32x4 SV39x4 SV48x4 | 16 KiB I/DCache | fully assoc. L1 I/DTLB, 4-entries | X | X | WIP |
| **CVA6** | **6-stage in-order** | **v1.0** | ✓ | **SV39x4** | **32 KiB DCache, 16KiB ICache** | **full-assoc. I/DTLB, 4-entries** | **set assoc. 4KiB 128-256 entries, 4-8 ways or/and set assoc. 2MiB 32-64 entries, 4-8 ways** | **GTLB** | **Done** |

## VII. RELATED WORK

The RISC-V hypervisor extension is a relatively new addition to the privileged architecture of the RISC-V ISA, ratified as of December 2021. Notwithstanding, from a hardware perspective, there are already a few open-source and commercial RISC-V cores with hardware virtualization support (or ongoing support): (i) the open-source Rocket core [37]; (ii) the space-grade NOEL-V [38], [39], from Cobham Gaisler; (iii) the commercial SiFive P270, P500, and P600 series; (iv) the commercial Ventana Veryon V1; (v) the commercial Dubhe from StarFive; (vi) the open-source Chromite from InCore Semiconductors; and (vii) the open-source Shakti core from IIT Madras.

Table VII summarizes some information about the open-source cores, focusing on their microarchitectural and architectural features relevant to virtualization. Table VII shows that RISC-V cores with full support for the hypervisor extension (e.g., the Rocket core and NOEL-V) are only compliant with version 0.6 of the specification. The work described in this paper is already fully compliant with ratified version 1.0. Regarding the microarchitecture: (i) the Rocket core has some general MMU optimizations (e.g., PTE cache and L2 TLB), non of which are special targeting virtualization (like our GTLB); (ii) Chromite and Shaktii have no MMU optimizations for virtualization (and their support for virtualization is still in progress); and (iii) NOEL-V has a dedicated hypervisor TLB (hTLB). From a Cache hierarchy point of view, all cores have identical sizes of L1 Cache, except the CVA6 ICache, which is smaller (16KiB).

In terms of functional performance, our prior work on the Rocket Core [23] concluded an average of 2% performance overhead due to the hardware virtualization support. Findings in this work are in line with the ones presented in [23] (however, with a higher penalty in performance due to the area-energy focus of the microarchitecture of the CVA6). Notwithstanding, we were able to optimize and extend the microarchitecture to significantly improve performance at a fraction of area and power. Results in terms of performance are not available for other cores, i.e., NOEL-C, Chromite, and Shaktii. From an energy and area breakdown perspective, none of these cores have made a complete public PPA evaluation. NOEL-V only reported a 5% area increase for adding the hypervisor extension.

A large body of literature also covers techniques to optimize the memory subsystem. Some focus on optimizing the TLB [40]–[48], while others aim at optimizing the PTW [49], [50]. For instance, in [50], authors proposed a dedicated TLB on the PTW to skip the second translation stage and reduce the number of walk iterations. Our proposal for the GTLB structure is similar. Nevertheless, to the best of our knowledge, this is the first work to present a design space exploration evaluation and PPA analysis for MMU microarchitecture enhancements in the context of virtualization in RISC-V, fully supported by a publicly accessible open-source design [7].

## VIII. CONCLUSION

This article reports our work on hardware virtualization support in the RISC-V CVA6 core. We start by describing the baseline extension to the vanilla CVA6 to support virtualization and then focus on the design of a set of enhancements to the nested MMU. We first designed a dedicated G-Stage TLB to speedup the nested-PTW on TLB Miss. Then, we proposed an L2 TLB with multi-page size support (4KiB and 2MiB) and configurable size. We carried out a design space exploration evaluation on an FPGA platform to assess the impact on functional performance (execution cycles) and hardware. Then, based on the DSE evaluation, we elected a few hardware configurations and performed a PPA analysis based on 22nm FDX technology. Our analysis demonstrated several design points on the performance, power, and area trade-off. We were able to achieve a performance speedup of up to 19% but at the cost of a 10% area increase. We selected an optimal design configuration where we observed an average performance speedup of 12.5% at a fraction of the area and power, i.e., 0.78% and 0.33%, respectively.

## ACKNOWLEDGMENTS

This work has been supported by Technology Innovation Institute (TII), the European PILOT (EuroHPC

[7] https://github.com/minho-pulp/cva6/tree/feature-hyp-ext-final

JU, g.a. 101034126), the EPI SGA2 (EuroHPC JU, g.a. 101036168), and FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope UIDB/00319/2020 and Scholarships Project Scope SFRH/BD/138660/2018 and SFRH/BD/07707/2021.

## REFERENCES


[1] J. Martins and et al., "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Proc. of Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, 2020.

[2] G. Heiser, "Virtualizing embedded systems - why bother?" in *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011.

[3] M. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," in *IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2019.

[4] S. Pinto and et al., "Virtualization on TrustZone-Enabled Microcontrollers? Voilà!" in *IEEE RTAS*, 2019.

[5] D. Cerdeira and et al., "ReZone: Disarming TrustZone with TEE privilege reduction," in *USENIX Security Symposium*, 2022.

[6] R. Uhlig and et al., "Intel virtualization technology," *Computer*, no. 5, 2005.

[7] Arm Ltd., "Isolation using virtualization in the Secure world Secure world software architecture on Armv8.4," 2018.

[8] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[9] N. Flaherty, "Europe steps up as RISC-V ships 10bn cores ," Jul 2022. [Online]. Available: https://www.eenewseurope.com/en/europe-steps-up-as-risc-v-ships-10bn-cores/

[10] M. Gautschi and et al., "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[11] L. Valente and et al., "HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC," 2022.

[12] D. Rossi and et al., "Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode," *IEEE Journal of Solid-State Circuits*, 2022.

[13] A. Garofalo and et al., "Darkside: 2.6GFLOPS, 8.7mW Heterogeneous RISC-V Cluster for Extreme-Edge On-Chip DNN Inference and Training," in *IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, 2022.

[14] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Trans. on VLSI Systems*, 2019.

[15] C. Chen and et al., "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[16] F. Ficarelli and et al., "Meet Monte Cimone: Exploring RISC-V High Performance Compute Clusters," in *Proc. of the 19th ACM International Conference on Computing Frontiers*, 2022.

[17] A. Waterman and et al., "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.12-draft," *RISC-V Foundation, Jul*, 2022.

[18] A. Patel and et al., "Embedded Hypervisor Xvisor: A Comparative Analysis," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[19] S. Zhao, "Trap-less Virtual Interrupt for KVM on RISC-V," in *KVM Forum*, 2020.

[20] G. Heiser, "seL4 is verified on RISC-V!" in *RISC-V International*, 2020.

[21] J. Hwang and et al., "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in *IEEE Consumer Communications and Networking Conference*, 2008.

[22] R. Ramsauer and et al., "Static Hardware Partitioning on RISC-V–Shortcomings, Limitations, and Prospects," in *IEEE World Forum on Internet of Things*, 2022.

[23] B. Sa and et al., "A First Look at RISC-V Virtualization from an Embedded Systems Perspective," *IEEE Transactions on Computers*, 2021.

[24] J. Andersson, "Development of a NOEL-V RISC-V SoC Targeting Space Applications," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020.

[25] N. Gala and et al., "SHAKTI Processors: An Open-Source Hardware Initiative," in *International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*.

[26] J. Scheid and J. Scheel, "RISC-V "stimecmp / vstimecmp" Extension Document Version 0.5.4-3f9ed34," Oct 2021. [Online]. Available: https://github.com/riscv/riscv-time-compare/releases/download/v0.5.4/Sstc.pdf

[27] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, 1974.

[28] M. Cavalcante and et al., "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI," *IEEE Transactions on VLSI Systems*, 2020.

[29] D. Kruckemyer and et al., "RISC-V Base Cache Management Operation ISA Extensions Document Version 1.0-fd39d01," Dec 2022. [Online]. Available: https://raw.githubusercontent.com/riscv/riscv-CMOs/master/specifications/cmobase-v1.0.pdf

[30] J. Hauser, "The RISC-V Advanced Interrupt Architecture Document Version 0.3.0-draft," Jun 2022. [Online]. Available: https://github.com/riscv/riscv-aia

[31] J. Martins and S. Pinto, "Shedding Light on Static Partitioning Hypervisors for Mixed-Criticality Systems," in *IEEE Symposium on RTAS*, 2023.

[32] S. Venkata and et al., "SD-VBS: The San Diego Vision Benchmark Suite," 2009.

[33] T. Kloda and et al., "Lazy Load Scheduling for Mixed-Criticality Applications in Heterogeneous MPSoCs," *ACM Trans. Embed. Comput. Syst.*, 2023.

[34] G. Gracioli and et al., "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[35] J. Balkind and et al., "OpenPiton: An Open Source Manycore Research Framework," in *Proc. of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2016.

[36] Q. Ge and et al., "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2018.

[37] K. Asanović et al., "The Rocket Chip Generator," EECS Department, Univ. of California, Berkeley, Tech. Rep., 2016.

[38] Gaisler, "GRLIB IP Core Users Manual," Jul 2022. [Online]. Available: https://www.gaisler.com/products/grlib/grip.pdf

[39] J. Andersson, "Development of a NOEL-V RISC-V SoC Targeting Space Applications," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020.

[40] Y.-J. Chang and M.-F. Lan, "Two New Techniques Integrated for Energy-Efficient TLB Design," *IEEE Transactions on VLSI Systems*.

[41] B. Pham and et al., "CoLT: Coalesced Large-Reach TLBs," in *IEEE/ACM International Symposium on Microarchitecture*, 2012.

[42] M. Talluri and et al., "Tradeoffs in Supporting Two Page Sizes," in *Proc. of the Annual ISCA*, 1992.

[43] B. Pham and et al., "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[44] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proc. of the 22th International Conference on ASPLOS*, 2017.

[45] J. H. Ryoo and et al., "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Proc. of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[46] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proc. 29th Annual ISCA*, 2002.

[47] A. Bhattacharjee and et al., "Shared Last-level TLBs for Chip Multiprocessors," in *IEEE International Symposium on HPCA*, 2011.

[48] S. Bharadwaj and et al., "Scalable distributed last-level tlbs using low-latency interconnects," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[49] T. Barr and et al., "Translation Caching: Skip, Don't Walk (the Page Table)," *SIGARCH Comput. Archit. News*, 2010.

[50] R. Bhargava and et al., "Accelerating Two-Dimensional Page Walks for Virtualized Systems," in *Proc. of the 13th International Conference on ASPLOS*, 2008.