外文资料

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 RISCV [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 market1 . 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.

In this work, we describe the architectural and microarchitectural support for virtualization in a 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 L1 TLB, (ii) a dedicated second stage TLB coupled to the PTW (i.e., GTLB in our lingo), and (iii) an 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 10 designs and analyze them in depth with post-layout simulations of implementations in 22nm FDX technology.

To summarize, with this work, we make the following contributions. Firstly, we provide hardware virtualization support in the CVA6 core. In particular, we design a set of (virtualization-oriented) microarchitectural enhancements to the nested memory management unit (MMU) (Section III). To the best of our knowledge, there is no work or study describing and discussing microarchitectural extensions to improve the hardware virtualization support in a RISC-V core. Second, we perform a design space exploration (DSE), encompassing dozens of design configurations. This DSE includes tradeoffs 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 Power, Performance, and Area (PPA) analysis (Sections V).

For the DSE evaluation, we ran the MiBench (automotive subset) benchmarks to assess functional performance. The virtualization-aware non-optimized CVA6 implementation served as our baseline configuration. The software setup encompassed a single Linux 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 can achieve a functional performance speedup up to 19benchmark); however, in some cases at the cost of a nonnegligible increase in area and power with respect to the baseline. 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 1and (iv) the L2 TLB introduces a non-negligible 8in area in some configurations. As a representative, wellbalanced 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 16and 0.33reports the first public work on a complete DSE evaluation and PPA analysis for a virtualization-enhanced RISC-V core.

Background

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; and (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 Mmode support, to fully blown server class systems with M/S/U.

CVA6

CVA6 (formerly known as Ariane) 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 memory management unit (MMU), making it suitable for running a fully-fledged OS such as Linux. Recent additions also include a energy-efficient Vector Unit co-processor [27]. 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 Page Table Walker (PTW) implements the Sv39 and Sv32 translation modes as defined by the privileged specification [17].

RISC-V Virtualization

Unlike other mainstream ISAs, the RISC-V privileged architecture was designed from the initial conception to be classically virtualizable [28]. 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 in 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 RISCV privileged architecture specification introduced hardware support for virtualization through the (optional) Hypervisor extension [17].

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 hypervisors2 . Additionally, two new privileged modes are added

and can be leveraged to run 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 stage of translation (G-stage in RISC-V lingo) to virtualize the guest memory by translating guestphysical addresses (GPA) into host-physical addresses (HPA). 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 respective 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).

Nested-MMU

The MMU is a hardware component responsible for translating virtual memory references to physical ones, while enforcing memory access permissions. The OS holds control over the MMU by assigning a virtual address space to each process and managing the MMU translation structures in order to correctly translate virtual addresses (VAs) into physical addresses (PAs). On a virtualized system, the MMU can translate from guestvirtual addresses (GVAs) to guest-physical addresses (GPAs) and from GPA into host-physical addresses (HPAs). In this case, this feature is referred to as nested-MMU. The RISCV ISA supports the nested-MMU through a new stage of translation that converts GPA into HPA, 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 RISCV privileged specification defines that a VA is converted into a PA by transversing a multi-level radix-tree table using one of four different topologies: (i) Sv32 for 32 virtual address spaces (VAS) with a 2-level hierarchy tree; (ii) Sv39 for 39- bit VAS with a 3-level tree; (iii) Sv48 for 48-bit VAS and 4-level tree; and (iv) Sv57 for 57-bit VAS and 5-level tree. Each level holds a pointer to the next table (non-leaf entry) or the final translation (leaf entry). This pointer and respective 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 on non-leaf, it supports superpages to reduce the TLB pressure, e.g., Sv39 supports 4KiB, 2MiB, and 1GiB page sizes.

RISC-V "stimecmp/vstimecmp" Extension(Sstc)

The RISC-V Sstc extension [26] aims at enhancing supervisor mode with its own 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 additionally CSRs: (i) stimecmp and (ii) vstimecmp. Whenever the value of time counter register is greater then the value of stimecmp, the supervisor timer interrupt pending (STIP) bit goes high and a timer interrupt is deliver to HS/S-mode. The same happens for VS-mode, with one subtle difference: the offset of the guest delta register (htimedelta) is added to the time value. If vstimecmp is greater than time+htimedelta, VSTIP goes high and the VS-mode 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].

CVA6 Hypervisor Support: Architecture And Microarchitecture

In this section, we describe the architectural and microarchitectural hardware virtualization support in the CVA6 (compliant with the RISC-V Hypervisor extension v1.0), illustrated in Figure 2.

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 HSmode and adds two extra orthogonal execution modes, denoted VS-mode and VU-mode. To add support for this 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 VTM bit is set on hstatus CSR). 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 I, which presents a summary of the features that were fully and partially implemented. We have implemented all mandatory features of the ratified 1.0 version of the specification; 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 hgeip depend 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 same translation and permission checks as in VS-mode or VU-mode. These instructions change the translation settings at the instruction granularity level, forcing a full 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 as specified in the hstatus.SPVP field.

C. Nested Page-Table Walker(PTW)

One of the major functional blocks of an MMU is the pagetable walker (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 stage of translation (Gstage) that is 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 (virtual space already supported by the CVA6).

We extended the existing finite state machine (FSM) used to translate VA to PA and added only a new control state to keep track of the current stage of translation 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 leaf GPA resulting from the VS-

Stage translation but also (ii) all nonleaf PTE GPA used 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 GPA; (ii) GStage Intermed - the PTW current state is translating non-leaf PTE GPA into HPA; and (iii) G-Stage Final - the PTW current state is translating the final output address from VS-Stage into an HPA. It is worth noting if hgatp is in Bare mode, no Gstage translation is in place, and we perform a standard VSStage translation. Once the nested walk completes, the PTW updates the TLB with the final PTE from VS-stage and Gstage, alongside the current address space identifier (ASID) and the VMID. One implemented optimization consists in storing the translation page size (i.e., 4KiB, 2MiB, and 1GiB) for both VS- and G-stages into the same TLB entry as well as permissions access bits for each stage. This helps to reduce the TLB hit time and improve hardware reuse.

D. Virtualization-aware TLbs(vTLB)

The CVA6 MMU microarchitecture has two small fully associative TLB: one for data (L1 DTLB) and the other for 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 microarchitecture of the L1 DTLB and ITLB to support two stages of translation, 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 4KiB page size, the TLB lookup would not benefit from superpages, since the translation would be stored as a 4KiB page size translation. The alternative approach would be to have separate TLBs for the VS-stage and G-stage final translations, but it would translate into higher hardware costs, less hardware reuse (if G-stage is not active), and a higher performance penalty on an L1 TLB hit (TLB search time increased by approximately a factor of 2). Since L1 TLBs are fully combinational circuits that lie on the critical path of the CPU, we decide 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. As a final note, the TLB also allows flushes by

guest physical address, i.e., hypervisor fence instructions (HFENCE.VVMA/GVMA) are fully supported.

E. Microarchitectural extension #1 - GTLB

A full nested table walk for the Sv39 scheme can take up to 15 memory accesses, five times more than a standard Sstage translation. This additional burden imposes a higher TLB miss penalty, resulting in a (measured) overhead of up to 13on 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 G-Stage TLB (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 Gstage, (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 Gstage translations cached, while avoiding unnecessary cache pollution and nondeterministic memory accesses. Another reason to support such an approach is related to the fact that PTEs are packed together in sequence in memory, i.e., multiple VS-Stage non-leaf PTE address translations will share the same GTLB entry [31].

中文译文

虚拟化是一种广泛应用的技术推动因素，从云计算和服务器到移动和嵌入式系统 [1]。作为云计算的基本基石，虚拟化为工作负载管理、数据保护和成本/功率效率提供了众多优势 [2]。另一方面，嵌入式和安全关键系统行业已将虚拟化作为一种基本方法来应对市场压力，以最大限度地减少尺寸、重量、功率和成本 (SWaP-C)，同时保证时间和空间认证隔离（例如，ISO26262）[3]-[5]。由于虚拟化在多个行业和用例中的激增，硅行业的知名企业开始在主流计算架构中引入硬件虚拟化支持（例如，分别为英特尔虚拟化技术、Arm 虚拟化扩展）[6]、[7]。

计算机体系结构的最新进展带来了一种名为 RISC-V [8] 的新型指令集架构 (ISA)。RISC-V 最近达到了一百亿个出口的大关 [9]。它通过提供免费和开放的标准 ISA 将自己与经典主流区分开来，具有模块化和高度可定制的扩展方案，使其能够从小型微控制器扩展到超级计算机 [10]-[16]。RISC-V 特权架构通过定义 Hypervisor 扩展 [17] 为虚拟化提供硬件支持，于 2021 年第四季度获得批准。

尽管 H 扩展已经获得了批准，但截至撰写本文时，在市场上仍然没有支持 H 扩展的芯片。有上游支持 Hypervisor 扩展的开源 hypervisor，例如 Bao [1]、Xvisor [18]、KVM [19] 和 seL4 [20]（Xen [21] 和 Jailhouse [22] 正在进行中])。然而，据我们所知，只有少数硬件实现部署在 FPGA 上，包括 Rocket 芯片 [23] 和 NOEL-V [24]（很快还有 SHAKTI 和 Chromite [25]）。尽管如此，现在还没有 (i) 专注于理解和增强虚拟化的微体系结构，以及 (ii) 执行设计空间探索 (DSE) 和伴随的功率、性能、面积 (PPA) 分析的工作。

在这项工作中，我们描述了基于开源 RISC-V CVA6 [14]（64 位）SoC 中虚拟化的架构和微架构支持。在体系结构层面，该实现符合 Hypervisor 扩展 (v1.0) [17]，并且还包括 RISC-V 计时器 (Sstc) 扩展 [26] 的实现。在微架构层面，我们修改了 CVA6 微架构以支持 Hypervisor 扩展，并提出了一组额外的扩展/增强功能以减少硬件虚拟化开销：(i) L1 TLB，(ii) 专用第二阶段 TLB PTW（即我们术语中的 GTLB），以及 (iii) L2 TLB。我们还介绍并讨论了对微体系结构的综合设计空间探索。我们首先评估部署在 FPGA (Genesys 2) 上的 23 个（共 288 个）硬件设计，并评估对功能性能（执行周期）和硬件的影响。然后，我们选择 10 种设计，并通过 22nm FDX 技术实现的后仿真技术对其进行深入分析。

总而言之，通过这项工作，我们做出了以下贡献。首先，我们在 CVA6 内核中提供硬件虚拟化支持。尤其是我们为嵌套内存管理单元 (MMU) 设计了一组（面向虚拟化的）微体系结构增强功能（第三节）。据我们所知，没有任何工作或研究描述和讨论微架构扩展以改进 RISC-V 内核中的硬件虚拟化支持。其次，我们进行了关于设计空间的探索 (DSE)，其中包含数十种设计配置。该 DSE 包括权

衡来自三个不同微架构组件（L1 TLB、GTLB、L2 TLB）的参数以及各自对功能性能和硬件成本的影响（第四节）。最后，我们对一些选定的设计配置进行布局后仿真，以评估功率、性能和面积 (PPA) 分析（第五节）。

对于 DSE 评估，我们运行了 MiBench（automotive 的子集）基准测试来评估功能性能。未优化的实现虚拟化科扩展的 CVA6 作为我们的基准配置。软件启动包括一个运行在 Bao 虚拟机管理程序之上的 Linux 虚拟机，用于单核设计。我们测量了托管 Linux VM 相对于基准配置的性能加速。DSE 探索的结果表明，所提出的微架构扩展可以实现高达 19

背景

A. RISC-V 特权级标准

RISC-V 特权指令集架构 (ISA) [17] 将其执行模型分为 3 个特权级别：(i) Machine 模式（M-Mode）是最高特权级别，托管实现 supervisor 二进制接口的固件 (SBI)（例如，OpenSBI）；(ii) Supervisor 模式 (S-Mode) 运行需要虚拟内存管理的 Unix 类型操作系统 (OSes)；(iii) User 模式（U-Mode）执行用户态应用程序。RISC-V ISA 提供的模块化无缝地允许在不同的设计点实现，范围从仅支持 M 模式的小型嵌入式平台到具有 M/S/U 的成熟服务器类系统。

B. CVA6

CVA6（以前称为 Ariane）是一个应用级 RISC-V 软核，它实现了 RISC-V ISA [14] 的 RV64 和 RV32 两个版本。该核完全支持 M/S/U 三种特权执行模式，并通过实现内存管理单元（MMU），提供了内存虚拟化的硬件支持，使其适合运行一个完整的操作系统，如 Linux。最近的新增功能还包括一个节能的向量单元协处理器 [27]。在内部，CVA6 微架构包含了一个六级流水线，单发射，带有乱序执行阶段和 8 个 PMP 条目。MMU 有分开的数据和指令 TLB，而页表遍历单元（PTW）实现了特权规范 [17] 定义的 Sv39 和 Sv32 转换模式。

C. RISC-V 虚拟化

与其他主流 ISA 不同，RISC-V 特权架构从最初的概念就被设计为可传统虚拟化 [28]。因此，尽管 ISA 本身允许直接实现 hypervisor，例如采用经典虚拟化技术（例如，陷入并模拟和影子页表技术），但众所周知，此类技术会导致严重的性能下降并且无法应对当前的嵌入式实时虚拟化要求（例如，中断延迟）[23]。因此，为了提高虚拟化效率，RISC-V 特权架构规范通过（可选）管理程序扩展 [17] 引入了对虚拟化的硬件支持。

特权级. 如图 1 所示，RISC-V 管理程序扩展执行模型遵循正交设计，其中 supervisor 模式（S 模式）被修改为 hypervisor-extended supervisor 模式（HS 模式），非常适合托管两种类型 -1 或类型 2 管理程序 2。此外，还添加了两种新的特权模式，可用于在 virtual supervisor 模式（VS 模式）和 virtual user 模式（VU

模式）下运行 guest 操作系统。

两阶段地址翻译. Hypervisor 扩展还定义了第二阶段地址转换（RISC-V 术语中的 G 阶段），通过将客户物理地址 (GPA) 转换为主机物理地址 (HPA) 来虚拟化客户内存。HS 模式像 S 模式一样运行，但有额外的 hypervisor 寄存器和指令来控制虚拟机执行和 G 阶段转换。例如，hgatp 寄存器保存 G 阶段根页表地址和相应的翻译特定配置字段。

Hypervisor 控制状态寄存器 (CSRs). 在 VS 模式下运行的每个虚拟机都有自己的控制状态寄存器 (CSR)，它们是 S 模式 CSR 的影子副本。这些寄存器可用于确定客户机执行状态和执行虚拟机切换。为了控制虚拟化状态，使用了一个称为虚拟化模式（V 位）的特定标志。当 V=1 时，guest 在 VS-mode 或 VU-mode 下执行，正常的 S-mode CSRs 访问实际上是在访问 VS-mode CSRs，并且 G-stage translation 开启。否则，如果 V=0，则正常的 S 模式 CSR 开启，并且 G 阶段被禁用。为了简化与 guest 相关的异常陷阱处理，存在特定于来宾的陷阱，例如，guest 页面错误、VS 级异常和 VS 级 ecall（又名 hypercalls）。

D. Nested-MMU

MMU 是一个硬件组件，负责将虚拟内存转换为物理内存，同时强制进行内存访问权限检查。操作系统通过为每个进程分配虚拟地址空间并管理 MMU 转换结构来控制 MMU，以便将虚拟地址 (VA) 正确转换为物理地址 (PA)。在虚拟化系统上，MMU 可以将客户虚拟地址 (GVA) 转换为客户物理地址 (GPA)，并将 GPA 转换为主机物理地址 (HPA)。在这种情况下，此功能称为嵌套 MMU。RISC-V ISA 通过将 GPA 转换为 HPA 的新翻译阶段支持嵌套 MMU，表示为 G 阶段。guest VM 控制翻译的第一阶段（RISC-V 术语中的 VS 阶段），而 hypervisor 控制第二阶段（G 阶段）。最初，RISC-V 特权规范定义了通过使用四种不同拓扑之一遍历多级基数树表将 VA 转换为 PA：(i) Sv32 用于 32 位虚拟地址空间 (VAS)，具有 2 级层层级树；(ii) 39 位虚拟地址空间 Sv39，具有 3 级树；(iii) 48 位虚拟地址空间和 4 级树的 Sv48；(iv) 用于 57 位虚拟地址空间和 5 级树的 Sv57。每个级别都有一个指向下一个表（非叶条目）或最终翻译（叶条目）的指针。该指针和相应的权限存储在 64 位 (RV64) 或 32 位 (RV32) 宽度的页表条目 (PTE) 中。请注意，RISC-V 将虚拟地址拆分为 4KiB 页面大小，但由于每个级别都可以将非叶子节点作为叶子节点，因此它支持超页以减少 TLB 压力，例如，Sv39 支持 4KiB、2MiB 和 1GiB 页面大小。

E. RISC-V "stimercmp/vstimecmp" 扩展 (Sstc)

RISC-V Sstc 扩展 [26] 旨在通过其自身的定时器中断设施增强 supervisor 模式，从而消除模拟 S/HS 模式定时器和在 M 模式下产生定时器中断的大量开销。Sstc 扩展还为 VS 模式的 Hypervisor 扩展添加了类似的功能。为了能够在 HS/S 模

式和 VS 模式下直接控制定时器中断，Sstc 包含两个额外的 CSR：(i) stimecmp 和 (ii) vstimecmp。每当时间计数器寄存器的值大于 stimecmp 的值时，监控定时器中断挂起 (STIP) 位变为高电平并且定时器中断被传送到 HS/S 模式。VS 模式也是如此，只有一个细微差别：guest delta 寄存器 (htimedelta) 的偏移量被添加到时间值中。如果 vstimecmp 大于 time+htimedelta，则 VSTIP 变为 1 并产生 VS 模式定时器中断。有关 RISC-V 定时器架构的完整概述以及关于为什么经典 RISC-V 定时器规范会导致显着性能损失的讨论，我们建议有兴趣的读者参阅 [23]。

CVA6 Hypervisor 支持：体系结构和微体系结构

在本节中，我们描述了 CVA6 中的架构和微架构硬件虚拟化支持（符合 RISC-V Hypervisor 扩展 v1.0），如图 2 所示。

A. Hypervisor 和 Virtual Supervisor 执行模式

如前所述（参见第 II-C 节），Hypervisor 扩展规范将 S 模式扩展为 HS 模式，并添加了两个额外的正交执行模式，分别表示为 VS 模式和 VU 模式。为了增加对这种新执行模式的支持，我们扩展/修改了一些 CVA6 核心功能块，特别是 CSR 和解码模块。如图 2 所示，硬件虚拟化架构逻辑包含五个构建块：(i) VS 模式和 HS 模式 CSR 访问逻辑和权限检查；(ii) 异常和中断的触发和代理；(iii) 陷阱的进入和退出；(iv) hypervisor 指令解码和执行；(v) 嵌套 MMU 翻译逻辑。CSR 模块被扩展以实现构成硬件虚拟化逻辑的前三个构建块，具体而言：(i) HS 模式和 VS 模式 CSR 访问逻辑（读/写操作）；(ii) HS 和 VS 执行模式陷阱进入和返回逻辑；(iii) 从 M 模式到 HS 模式和/或 VS/VU 模式的异常/中断触发和代理逻辑的一部分（例如，当 VTM 位在 hstatus CSR 中被设置时，读/写 vsatp CSR 在 VS 模式下触发异常）。修改解码模块以实现 hypervisor 指令解码（例如，hypervisor 加载/存储指令和 memory fence 指令）和所有 VS 模式相关指令执行访问异常触发。

我们建议读者参阅表 I，其中总结了已完全和部分实现的功能。我们已经实现了已批准的 1.0 版规范的所有必要性功能；然而，由于对即将到来的或更新的扩展的依赖，我们仍然保留了一些可选功能作为部分实现。例如：hvencfg 位依赖于 Zicbom [29]（缓存块管理操作）；hgeie 和 hgeip 依赖于 Advanced Interrupt Architucture (AIA) [30]；和 hgatp 依赖于 vanilla CVA6 当前不支持的虚拟地址空间。

B. Hypervisor Load/Store 指令

Hypervisor 加载/存储指令（即 HLV、HSV 和 HLVX）为 hypervisor 提供了一种机制来访问 guest 内存，同时接受与 VS 和 VU 模式中相同的地址翻译和权限检查。这些指令在指令粒度级别更改地址翻译，强制完全交换特权级别和在每个 hypervisor 加载/存储指令执行时应用的翻译。该实现包括向 CVA6 管道添加信号（在图 2 中标识为 hyp ld/st），该管道从解码行进到执行阶段的加载/存储单元。然

后将该信号送入 MMU，MMU 执行 (i) 所有必要的上下文切换（即启用 hgatp 和 vstap CSR），(ii) 启用虚拟化模式，以及 (iii) 更改 hstatus.SPVP 中指定的执行模式字段。

C. 嵌套页表遍历 (PTW)

MMU 的主要功能块之一是页表遍历器 (PTW)。从根本上说，PTW 负责根据特定的拓扑和方案（例如 Sv39）对虚拟地址进行划分，然后使用内存页表结构将其转换为物理地址。Hypervisor 扩展指定了一个新的转换阶段（G 阶段），用于将客户物理地址转换为主机物理地址。我们的实现支持裸机转换模式（无 G 阶段）和 Sv39x4，它定义了一个 41 位宽的最大客户物理地址空间（CVA6 已经支持的虚拟空间）。

我们扩展了现有的用于将虚拟地址翻译成物理地址的有限状态机 (FSM)，并仅添加了一个新的控制状态来跟踪翻译的阶段并帮助 VS-Stage 和 G-Stage 翻译之间的上下文切换。使用 G 翻译，必须翻译 (i) VS-Stage 翻译产生的叶 GPA，以及 (ii) VS-Stage 遍历期间使用的所有非叶 PTE GPA。为实现这一目标，我们确定了 PTW 迭代期间可能发生的三个转换阶段：(i) VS-Stage——PTW 当前状态正在将 guest VA 转换为 GPA；(ii) G-Stage Intermed——PTW 目前的状态是将非叶页表项 GPA 转化为 HPA；(iii) G-Stage Final——PTW 当前状态正在将最终输出地址从 VS-Stage 转换为 HPA。值得注意的是，如果 hgatp 处于 Bare 模式，则没有 G-stage 转换，我们执行标准的 VS-Stage 转换。嵌套遍历完成后，PTW 使用最终的 PTE VS 阶段和 G 阶段以及当前地址空间标识符 (ASID) 和 VMID 更新 TLB。一项已经实现的优化包括将 VS 和 G 阶段的翻译页面大小（即 4KiB、2MiB 和 1GiB）存储到同一个 TLB 条目中，以及每个阶段的权限访问位。这有助于减少 TLB 命中时间并提高硬件重用率。

D. Virtualization-aware TLBs(vTLB)

CVA6 MMU 微架构有两个小的全关联 TLB：一个用于数据 (L1 DTLB)，另一个用于指令 (L1 ITLB)。两个 TLB 都最多支持 16 个条目并完全实现了刷新指令，即 sfence，包括按 ASID 和虚拟地址过滤。为了支持嵌套翻译，我们修改了 L1 DTLB 和 ITLB 的微体系结构以支持两阶段翻译，包括访问权限和 VMID。每个 TLB 条目都有 VS-Stage 和 G-Stage PTE 以及各自的权限。查找逻辑的实现是使用来自两个阶段结果的合并最终翻译大小执行的，即如果 VS 阶段是 4KiB 而 G 阶段是 2MiB，则最终翻译的结果将是 4KiB。这可能是将 VS 阶段和 G 阶段存储在一起的主要缺点之一。例如，支持超页/大页的 hypervisor 使用 2MiB 页面大小来优化 MMU 性能。尽管这会极大减少了 PTW 遍历时间，但如果 guest 使用 4KiB 页面大小，TLB 查找将不会从超页中获益，因为页表翻译将会转化为 4KiB 页面大小。另一种方法是为 VS 阶段和 G 阶段的最终转换使用单独的 TLB，但这

会转化为更高的硬件成本以及较少的硬件可重用（如果 G 阶段未激活）以及更高的性能损失如果 L1 TLB 命中的话（TLB 搜索时间增加了大约 2 倍）。由于 L1 TLB 是位于 CPU 关键路径上的完全组合电路，我们决定将 VS 阶段和 G 阶段转换保留在单个 TLB 条目中。最后，TLB 还支持 VMID 标签，允许 TLB 管理程序可以使用 per-VMID 刷新执行更高效的 TLB 管理，并避免在 VM 上下文切换时完全刷新 TLB。最后一点，TLB 还允许通过 guest 物理地址进行刷新，即完全支持 hypervisor fence 指令 (HFENCE.VVMA/GVMA)。

E. 微体系结构扩展 #1 - GTLB

Sv39 方案的完整嵌套表遍历最多需要 15 次内存访问，是标准 S 阶段翻译的五倍。这种额外的负担会带来更高的 TLB 未命中惩罚，导致功能性能（执行周期）的（测量的）开销高达 13% （将基准虚拟化实现与普通 CVA6 进行比较）。为了缓解这种情况，我们在嵌套 PTW 模块中使用 G-Stage TLB (GTLB) 扩展了 CVA6 微架构，以存储中间 GPA 到 HPA 的转换，即 VS-Stage 非叶子 PTE guest 物理地址到 host 物理地址翻译。图 3 说明了将 GTLB 集成到 MMU 微体系结构中所需的修改。GTLB 结构旨在通过在 VS 阶段页表遍历期间跳过每个嵌套翻译来加速 VS 阶段翻译。图 4 显示了一个 4KiB 页面大小的转换过程，在 PTW 中具有 GTLB。如果没有 GTLB，guest 每次在遍历过程中形成非叶子物理地址页表项指针时，它需要：(i) 通过 G-Stage 转换它，(ii) 从内存中读取下一级 PTE 值，以及 (iii) 恢复 VS 阶段翻译。使用超页（2MiB 或 1GiB）时，翻译路径更少，从而降低了性能损失。使用简单的硬件结构，可以通过缓存这些 G 阶段翻译来减轻此类开销，同时避免不必要的缓存污染和不确定的内存访问。支持这种方法的另一个原因与 PTE 在内存中按顺序打包在一起的事实有关，即多个 VS-Stage 非叶 PTE 地址转换将共享相同的 GTLB 条目 [31]。