IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 13 October 2022, accepted 9 November 2022, date of publication 14 November 2022, date of current version 21 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3222346

RESEARCH ARTICLE

A General Framework for Accelerator Management Based on ISA Extension

ELHAM CHESHMIKHANI[1], **BIAGIO PECCERILLO**[1], **ANDREA MONDELLI**[2], **AND SANDRO BARTOLINI**[1]

[1]Department of Information Engineering and Mathematics, University of Siena, 53100 Siena, Italy
[2]Huawei Technologies Research & Development (UK) Limited, CB4 0FY Cambridge, U.K.

Corresponding author: Elham Cheshmikhani (e.cheshmikhani@unisi.it)

This work was supported by the Huawei Technologies Research and Development (U.K.) Ltd.

ABSTRACT Thanks to the promised improvements in performance and energy efficiency, hardware accelerators are taking momentum in many computing contexts, both in terms of variety and relative weight in the silicon area of many chips. Commonly, the way an application interacts with these hardware modules has many accelerator-specific traits and requires ad-hoc drivers that usually rely on potentially expensive system calls to manage accelerator resources and access orchestration. As a consequence, driver-based interfacing is far from uniform and can expose high latency, limiting the set of tasks suitable for acceleration. In this paper, we propose a uniform and low-latency interface based on Instruction Set Architecture (ISA) extension. All the previous studies that proposed extensions, were deeply tailored to address a single accelerator. One of the biggest disadvantages of those methods is their inability to scale. Adding more of these accelerators to one System-on-Chip (SoC) would result in ISA bloat, increasing power consumption and complexifying the decoding phase proportionally. Our proposed framework consists of a six-instruction ISA extension and the corresponding architectural support that implements the interface abstraction and the reservation logic at the hardware level. Our proposal allows controlling a broad class of integrated accelerators directly from the CPU. The proposed framework is ISA-independent, which means that it is applicable to all the existing ISAs. We implement it on the gem5 simulator by extending the RISC-V ISA. We evaluate it by simulating three compute-intensive accelerators and comparing our interfacing with a conventional driver-based one. The benchmarks highlight the performance benefits brought by our framework, with up to 10.38x speed up, as well as the ability to seamlessly support different accelerators with the same interface. The speed up advantage of our technique diminishes as the granularity of the workloads increases and the overhead for driver-based accelerators becomes less important. We also show that the impact of its hardware components on chip area and power consumption is limited.

INDEX TERMS Accelerators, domain-specific architectures, heterogeneous systems, ISA extension, RISC-V.

I. INTRODUCTION AND MOTIVATION

Domain-specific architectures or *accelerators* have recently emerged as a primary driving force of computer architecture [1], [2]. They are specialized hardware components located outside of a general-purpose CPU,[1] interacting with them to *accelerate* particular tasks, usually improving non-functional metrics such as throughput and efficiency.

The associate editor coordinating the review of this manuscript and approving it for publication was Sudipta Roy.

[1]We consider an accelerator as a "separate architectural substructure", in accordance with the definition expressed by Patel and Hwu in [3]. Some authors refer to these as "loosely-coupled accelerators", in opposition to "tightly-coupled accelerators", which are functional units inside the core [2], [4], [5], [6].

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Their strengths may derive from more efficient forms of parallelism, local/optimized memories and ad-hoc datapaths, reduced fetch and decode overhead, and support for ad-hoc data types [1], [7]. In the last years, accelerators have been commonly integrated into Systems on a Chip (SoCs) – mainly for mobile applications [8], [9], desktop workstations [10], [11], data-centers [10], [12], [13], and High-Performance Computing (HPC) systems [14].

Accelerators can be interconnected with the rest of the system in several ways [2], depending on their level of integration (e.g., integrated on chip, near-memory, discrete card) and their form factor. On-chip accelerators in SoCs usually communicate with other components through a Network on Chip (NoC) or a bus with a standard interface such as Advanced Microcontroller Bus Architecture (AMBA) [2], [15]. Discrete cards are usually shipped as PCIe cards, and thus implement the PCIe standard or others built on top of it such as Cache Coherent Interconnect for Accelerators (CCIX) [16] or Compute Express Link (CXL) [17].

From a software perspective, the *offload model* dictates the typical CPU-accelerator interaction, with sensible parts of an application offloaded to the accelerator [2], [18], [19]. This is generally achieved by invoking accelerator-specific API calls from the user application, triggering data movements to/from and computations on the accelerator. At a lower level, these API calls interact with device drivers that may live in user-space and/or kernel space, reaching the accelerator through MMIO techniques. These usually need some involvement from the Operating System (OS), which is responsible for: a) managing accelerator resources, b) orchestrating concurrent accesses from different processes to the accelerator, and c) taking care of virtual-to-physical address translation needs, or at least assisting the accelerator in this task [6].

Although this approach is the most common to communicate with accelerators, it exposes some limitations. First, the latency associated with core-accelerator interactions is typically high, compared to the rate of processor/core operations. It involves reaching an accelerator out of the cores, and this requires drivers/OS intervention. It poses a constraint to the size of offloadable tasks, since the whole offloading process is convenient only when the required computation time can amortize the interaction overhead, which can reach up to hundreds of thousands of CPU cycles [4]. Second, the interaction between cores and accelerators needs to be managed by the application developer and/or interfacing libraries, through SW-abstraction layers and OS support. This can hamper the compiler's capability to directly manage, orchestrate, and optimize integrated accelerators usage, which - for instance - is conversely consolidated for special functional units (e.g., vector units in modern processors).

In this paper, we present a general ISA extension for integrated accelerators. It targets the shortcomings listed above and can efficiently support a wide variety of accelerators with the same instructions, i.e., without the requirement of extending the ISA for each of them. It requires the addition of a thin hardware layer in the accelerators and the necessary transistors to interpret and execute the proposed instructions in the cores. Furthermore, we discuss our proposal as backed by a special NoC, but other solutions using the memory hierarchy are also possible.

Our proposal provides significantly lower latency than the driver-centric solution commonly adopted thanks to the proposed HW support, and makes accelerator management more compatible with the compiler activities and optimizations.

Our proposal is built around six instructions, which are added to the RISC-V instruction set. The essence of our solution can be distilled in the following major contributions:

- A general ISA extension for a wide variety of integrated accelerators, which is independent of the specific features of each of them while allowing their effective and low-latency usage;
- A limited architectural support for our ISA extension that allows reducing OS involvement in accelerator management through a distributed HW reservation mechanism of accelerators;
- Avoiding ISA bloating, which is particularly dangerous in the case of fixed-size instruction sets, and promoting easy scalability, as newly-introduced accelerators can be seamlessly supported;
- Efficient use of die-area and power, since the same ISA extension and micro-architecture can effectively interact with many accelerators;
- Design independence, as the proposed instructions do not dictate the design choices of the accelerator manufacturer, as many accelerators can be made compatible with our proposal using a thin hardware interfacing layer.

The rest of the paper is organized as follows: Section II summarizes the state-of-the-art in this field. Section III describes our proposed ISA extension framework. Section IV illustrates experimental results and, finally, Section V presents our conclusions.

II. RELATED WORK

This section sums up the main previous work proposing an extension of the processor ISA to facilitate CPU-accelerator interaction. We classify these studies into two groups: first, the studies that provide customized ISA extensions to use dedicated accelerators and functional units, which are the vast majority; and second, the ones proposing general ISA extensions to support multiple accelerators.

The authors of [20] discuss the trend toward graph computation and graph pattern mining in graph processing SoCs. They propose IntersectX, a vertically designed accelerator to increase both data movement and computation performance. The accelerator targets graph processing by the means of newly added instructions to the ISA. The ISA extension intrinsically operates on a sparse vector (stream), to directly work with added stream value processing units. They also provide a new compiler to generate new ISA-based graph pattern mining implementation.

Some works extend the MIPS ISA to support more special functions in the processor [21], [22] or navigation-data processing [23]. In [24] and [25], RISC-V and MIPS ISAs are extended for cryptography co-processors, respectively. Another work presents TIGRA, a Tightly Integrated, Generic RISC-V Accelerator interface [26]. It provides a custom logic accelerator with access to the necessary internal signals of the processor. TIGRA is controlled with custom RISC-V instructions expressed in the R-type format.

A mixed-signal accelerator in-SRAM for Machine Learning workloads, named PROMISE, is proposed in [27]. A comprehensive ISA is proposed to target the main machine learning operations. 48-bit VLIW instructions are used to specify up to four sequential operations. The authors provide also a compiler that translates high-level Julia code into an ISA-based PROMISE binary.

In Jain et al. [28], design an in-memory accelerator based on Spin-Transfer Torque Magnetic RAM (STT-MRAM). A modification to the on-chip bus architecture and an extension to the ISA is required to achieve integration with a general-purpose system. They perform simple logical and arithmetic operations like XOR, NOT, AND, and ADD on pairs of address lines.

A tightly-coupled RISC-V accelerator is presented in [29], wherein an out-of-order floating-point unit is implemented for the processor. In [30], an ARM ISA extension is proposed for improving public-key cryptography performance instead of using co-processor/accelerator as a usual solution. The authors choose ARM ISA as it is widely used for embedded applications.

In Liu et al. [31], extend the vector ISA to accelerate Machine Learning operations. In [32], Mazzola presents Xpulpimg, an extension of the RISC-V instruction set including domain-specific instructions for Digital Signal Processing (DSP). Xpulpimg combines signal processing functionalities with Snitch to leverage its already high level of parallelism.

Xuantie-910, an industrial 64-bit embedded RISC-V processor from Alibaba T-Head division is presented in [33]. It is based on the RV64GCV ISA and it features custom extensions for arithmetic operations, bit manipulation, loads/stores, TLB and cache operations. They deploy Xuantie-910 FPGA implementation in the data centers of Alibaba Cloud to accelerate blockchain transactions.

All the previous works fall in the first category, as the proposed extensions target a single accelerator or special function unit. Such approaches have the impossibility of scaling as their biggest downside. In fact, adopting more than one of these proposals in a SoC would lead to ISA bloat, increasing power consumption and complexity of the instruction decode phase proportionally with the number of supported accelerators. Also, the addition of a new accelerator would need further instructions and changes at the micro-architecture level. Conversely, our proposal aims at targeting a vast set of accelerators, making them controllable with the same instructions.

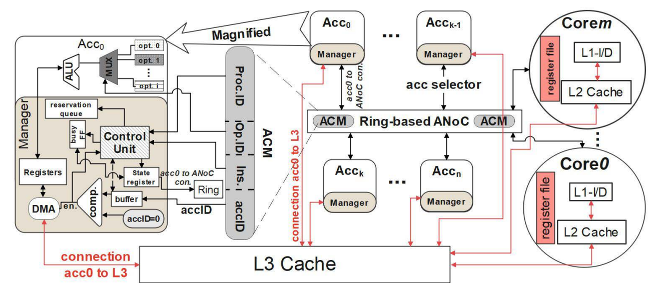


FIGURE 1. Architectural aspects of the proposed framework.

There are also a few works that fall into the second category and extend the ISA for general accelerators. ARM Coprocessor Interface [34] defines instructions to interact with up to 16 coprocessors. These are tightly coupled with the processor cores: they share fetch and execute control logic with the cores, so they need core's and coprocessor's pipelines to be synchronized. Thus, this proposal is more suitable for functional units, as it would not adapt to the "offload model" adopted by modern accelerators, in which the processor instructs an accelerator to perform a task that could last even thousands of clock cycles and is usually notified of the task completion by an interrupt.

In Cong et al. [4], discuss hardware architectural support for accelerator-rich CMPs, in which a SoC hosts a multi-core processor and many accelerators connected by a large NoC. They propose a central structure to be added to the chip named Global Accelerator Manager (GAM), so to filter, manage, and route accelerator-oriented instructions coming from the cores. The cores ask for accelerator availability to the GAM providing a description of the functionality that the required accelerator should implement and an estimation of the time they need it. With respect to our proposal, we do not need a central structure that could act as a bottleneck in the system. Moreover, we do not require users to know upfront an estimation of the time expected to perform a task.

III. PROPOSED ISA-EXTENSION FRAMEWORK

The main architectural aspects of the proposed framework are presented in this section. Then, we describe each of the six proposed instructions in detail and their intended usage.

A. ARCHITECTURAL ASPECTS OF THE PROPOSED FRAMEWORK

We assume that `n` accelerators can be connected to `m` cores within an ad-hoc on-chip interconnection (e.g., a simple bus for few accelerators or a ring-based interconnect for more, as shown in Figure 1) and the proposed solution does not impose specific requirements to the interconnection. Investigation of the design of such dedicated bus or Accelerator Network On Chip (ANoC) is beyond the scope of this work. In this regard, Figure 1 illustrates a reference SoC with an `m`-core processor connected to the network with `n` accelerators. Some registers, buffers, comparators, and queues are

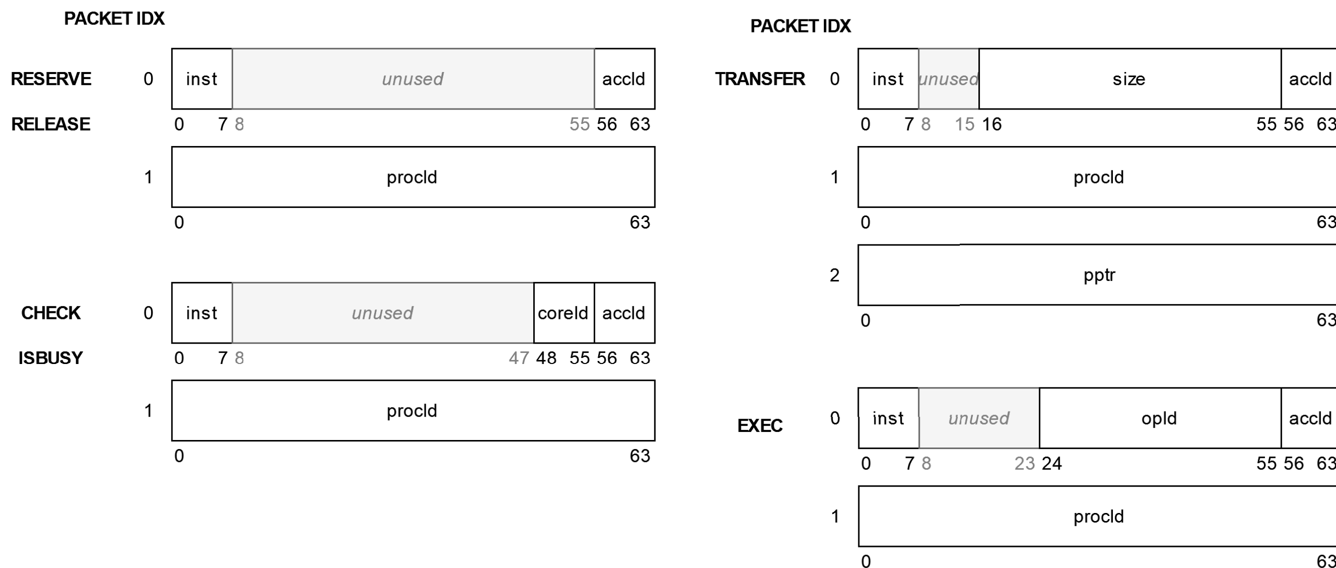


FIGURE 2. **Request ACMs generated by the execution of different instructions. In this example, they are organized in 64-bit packets.**

needed to support the proposed instruction set extension. They can be added to the accelerator architecture or provided as a thin interface layer towards an existing accelerator. Every accelerator has an immutable accelerator ID (`accId`), which can be assigned by the SoC designer. The register file of the processor cores is unmodified, and information used to interact with the accelerators (e.g., data, addresses, commands, etc.) is stored in general-purpose registers.

We define the messages exchanged between the cores and the accelerators as *Accelerator-Core Messages* (ACMs). There are two kinds of ACMs: a) *request ACM*, and b) *response ACM*. Request ACMs are sent to the accelerator on the ANoC when an accelerator-oriented instruction is executed on the core. They have the purpose of informing the accelerator about a task that the user wants to demand to the accelerator. Response ACMs flow in the opposite direction, and are sent from an accelerator to the caller core in response of a request ACM. This response contains a return value that will be written in a register on the core upon message reception. Figure 2 shows the request ACMs associated with the proposed instructions, which will be discussed in-depth in the next subsection. Without loss of generality, we adopt a sample format organized in indexed 64-bit packets.

Every ACM contains the parameters needed by the ANoC to route the message, and by the accelerator to execute the intended operation. The first byte, `inst`, identifies the instruction that has been executed on the core, and thus the task that is demanded to the accelerator. The instruction operands are added to the message in different positions. Between these, the operand `accId` is common to all the instructions, and is added to every ACMs in the eight least significant bits of the first packet. When an ACM is sent through the ANoC, every accelerator's comparator compares its ID to the `accId` contained in the ACM. If there is a match, the carried instruction is executed on that accelerator. Should the user provide a non-existent `accId`, an exception would be generated on the core.

The process originating the communication is identified through a *process specific identifier* (`procId`), which is written in the ACM by the processor. This value is never provided by the user for security reasons: a malevolent program could provide the process identifier of another running process, causing harm to that one. Thus, this value is read from the Control and Status Registers (CSRs). It can be any value uniquely identifying the running process, e.g., the pointer to the page table.

For some messages, also a `coreId` is added to the request ACM. It is used to identify the core where the instruction generating the ACM transmission is executed. Similar to `procId`, it is not provided by the user but automatically retrieved within each core.

Accelerators are equipped with a *reservation queue*. It is a FIFO queue storing the `procId`s of the reservation requests received by the accelerator. The head of the queue is considered the *owning process*, while the others are waiting to own the accelerator.

Although our configuration can work with other schemes, here we assume that accelerators access the memory system by having a direct connection to the shared L3 cache, which is a common choice for on-chip accelerators [2], and might take advantage of a Direct Memory Access (DMA) module to load the required operands and store results.

As we will explain in the next section, we assume each accelerator is able to perform different operations, that can be specified with the `opId` parameter, which is provided by the user as an operand of the EXEC instruction. A specific accelerator capability set needs to be mapped to different `opId`s that need to be shared between accelerator interface

TABLE 1. The proposed RISC-V R-type instructions. Unused operands are grayed out and set to `zero`. "Request" and "Response" specify the data carried by request ACMs and response ACMs, respectively.

	operands									
Instruction	**rd**	**rs1**	**rs2**	**Request**					**Response**	
RESERVE	zero	accId	zero	inst	accId	procId				
CHECK	ret	accId	zero	inst	accId	procId	coreId		ret	coreId
TRANSFER	size	accId	vptr	inst	accId	procId	pptr	size		
EXEC	zero	accId	opId	inst	accId	procId	opId			
ISBUSY	ret	accId	zero	inst	accId	procId	coreId		ret	coreId
RELEASE	zero	accId	zero	inst	accId	procId				

logic and applications (e.g., lightweight user-space driver and/or run-time system).

The `opId` value triggers a multiplexer connected to the accelerator ALU, as shown in Figure 1, and determines the operation to execute. When the accelerator starts executing, it sets a Flip-Flop (*busy-FF*). It is reset after completion, and its value can be read by the calling process any time to be informed about the execution state. When the *busy-FF* is reset, the accelerator can be released. When released, the owning process is pulled from the head of the reservation queue.

The hardware infrastructure described here needs to be added to an existing SoC for our proposal to work. Section IV shows that the impact of these additions is inconsequential.

B. ISA EXTENSION IN DETAIL

In this section, we present our ISA extension proposal based on the RV64I instruction set with Zicsr instructions [35]. It consists of six R-type instructions, which permit managing integrated accelerators: *RESERVE, CHECK, TRANSFER, EXEC, ISBUSY*, and *RELEASE*. All their operands are interpreted as general-purpose register operands holding the necessary data. However, although in this paper we propose a RISC-V implementation, the proposed instructions can be potentially implemented in other ISAs.

Each instruction invocation causes a request to be sent through the ANoC in the form of an ACM. Carried data include an instruction identifier, some values retrieved from processor registers, a `procId`, and a `coreId` in some cases. These two values are read from the CPU hardware registers and not explicitly provided by the user for security reasons. Every instruction has an `accId` operand that indicates the recipient accelerator and is used to determine the request target. If `accId` denotes a nonexistent accelerator, an illegal instruction error is raised.

All the proposed instructions are *asynchronous*, except CHECK and ISBUSY instructions. With this term, we denote instructions with no destination register operands, that do not need a response ACM to be sent back to the core. These instructions are executed on the core, a request ACM is sent, and they are committed on the core side. Conversely, *synchronous* instructions (i.e., CHECK and ISBUSY) have a destination register operand that needs to be set with a value carried by a response ACM, coming from an accelerator. They are committed only when a response ACM is received back and the value is written in the destination register. For this reason, synchronous instructions' request ACM carry a `coreId` that uniquely identifies the core where they are executed, so the response ACM can be delivered to the right core. From an out-of-order execution perspective, no special care is needed for these instructions apart from the ordinary *Read after Write* (RAW) dependency management of their result: since they have output registers, successive instructions having such register as input are blocked until the synchronous instruction is committed. Table 1 shows the instructions, their operands, and the data carried by both request ACMs and response ACMs.

1) RESERVE INSTRUCTION

`RESERVE` `<accId>`

In order to work safely with an accelerator in a multi-process environment, the applicant process (identified by `procId`) needs exclusive ownership. This is achieved through the RESERVE instruction, which reserves the requested accelerator (identified by `accId`) to the calling process. The RESERVE success or failure depends on the state of the accelerator, which is based on the *Finite State Machine* (FSM) depicted in Figure 3.

If the accelerator state is *Idle*, the accelerator is reserved to `procId` and its state transits to *Reserved*. If the accelerator is not *Idle*, it means that there is already a process owning it. This can be the same as `procId`, and in that case the instruction has no effect. Another possibility is that `procId` is not the owning process, but it is already waiting in the reservation queue. Also in that case, the instruction has no effect. Conversely, if the accelerator is already reserved to another process and `procId` is not waiting in the reservation queue, the reservation request is enqueued into the reservation queue, and is discarded only if the queue is full.

2) CHECK INSTRUCTION

`CHECK` `<ret>, <accId>`

The CHECK instruction checks the reservation state of the requested accelerator `accId` with respect to `procId`. It is used to check the outcome of a previous RESERVE instruction, which can be one of these three values: "0" (Reserved) if the accelerator is owned by `procId`; "1" (Enqueued) if a reservation request is present in the reservation queue; and "2" (Missing) if no request is in the queue, i.e., no

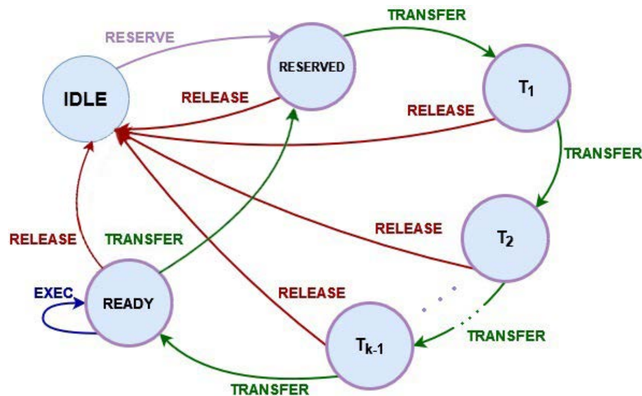


FIGURE 3. Finite state machine of a generic accelerator providing *k*-ary operations.

request was previously sent or the queue was full when it was received.

CHECK is synchronous: the accelerator is responsible for sending a response to the right core containing the output value. When received, this is written into the `<ret>` register and the instruction is committed.

3) TRANSFER INSTRUCTION

TRANSFER `<size>, <accId>, <vptr>`

Once the accelerator is reserved, the process identified by `procId` can start commanding tasks to it. If a TRANSFER with a `procId` that differs from the one owning the accelerator is received, the instruction is ignored.

Accelerated tasks, in general, will require processing some input data to produce output data. The TRANSFER instruction is used to indicate both input and output *memory buffers* located in the main memory to the accelerator. Buffers are not necessarily arrays of scalar elements (e.g., floats, integers, etc.), but can be *any* data structure. In general, buffers are memory regions that hold data that will be used as inputs in the accelerated tasks or memory regions that will be used to store the output produced by the accelerated tasks. Invoking a TRANSFER does not necessarily start a data transfer between accelerator and main memory, but is used to communicate a memory region to the accelerator that may be used later to load/store data from/to.

Depending on the *arity* of the operations performed by the accelerator, multiple TRANSFERs may be necessary to indicate all the involved buffers. For instance, a ternary operation would need two TRANSFERs to specify the input buffers and one to specify the output one. Figure 3 shows the FSM associated to an accelerator implementing k-ary operations. The first TRANSFER instruction (i.e., the first reception of request ACM containing a TRANSFER *inst*) causes the accelerator state to change from *Reserved* to T_1. Based on the number of needed input and outputs, the state of the accelerator goes through $T_2, \ldots, T_{k-1}$ and reaches *Ready* state after k TRANSFERS.

Each buffer is specified with two operands: a virtual address (`vptr`) and size. `vptr` is translated into a physical address (`pptr`) in the core, by searching the CPU's TLB. `pptr` is transmitted to the accelerator with the associated size, as shown in Figure 2. Then, the accelerator will use the DMA to connect to the L3 cache and retrieve the data contained in the buffer or store data to it.

Conceptually, if the received (`pptr`, `size`) pair indicates an input buffer, the accelerator can adopt an eager approach and start loading data immediately, or a lazy one and start loading data when execution is triggered, accommodating the scheme to different requirements and accelerators. In any case, the FSM regulating the accelerator functioning and how the communicated buffers are interpreted (i.e., whether input or output) is part of the accelerator programming interface and should be known by the programmer.

When the last TRANSFER is received (i.e., in T_{k-1} state) the accelerator has enough information to retrieve all the needed data to perform its k-ary task and knows where the produced output should be stored. It enters the *Ready* state and waits for an EXEC instruction to start execution.

The proposed scheme is suitable for a vast class of accelerators. Trivially, it can support accelerators proposing one or more operations with the same -arity. However, this constraint can be easily removed if the logic considers any state T_h as a "Ready" state for h-ary operations – thus, even operations with different -arity can be supported by the same accelerator.

4) EXEC INSTRUCTION

EXEC `<accId>, <opId>`

In the proposed scheme, the TRANSFER operation is separated from the execution so that the data transfer can take place independently and transfer-computation can overlap, allowing double-buffering, streaming, and pipelining techniques, accommodating different possible execution strategies that an accelerator could be able to implement.

The EXEC instruction starts the computation of a specific operation on the accelerator, identified by a numeric ID (`opId`). `accId` and `opId` are read from the source registers, `procId` is added as usual by the circuitry to let the accelerator determine the *legitimacy* of the request.

If TRANSFER is managed by the accelerator with an eager policy, the latter already started loading data and can start execution as soon as it completes its transfers. Conversely, if TRANSFER is managed lazily, data loading begins at this moment. In any case, when its source operands are ready, the selected accelerated computation is started.

When done, it stores the result in the TRANSFER-specified destination buffer (or *buffers*). Since a computation on the accelerator can last potentially several clock cycles, this instruction is designed as asynchronous, and the core does not wait for its completion. To check its completion state, ISBUSY instruction is used.

5) ISBUSY INSTRUCTION

ISBUSY `<ret>, <accId>`

TABLE 2. System configuration details.

CPU	Quad-core, 3.4GHz, RISC-V
L1 I/D Cache	32KB, 8-way, write-back, 64B block size, non-blocking, 2-cycles access time, private
L2 Cache	512KB, 8-way, write-back, 64B block size, non-blocking, 10-cycles access time, private
L3 Cache	8MB, 16-way, write-back, 64B block size, non-blocking, 36-cycles access time, shared
Interconnection	ring-based NoC, 15-cycles average latency
Main Memory	DRAM-DDR3, 2GB, 300-cycles access time

The ISBUSY synchronous instruction is used to check the completion of an operation on the accelerator `accId`. From a caller's perspective, it can be naturally included in a polling mechanism, that has the advantage of not needing additional access chips, is high-speed, and is relatively simple to program. On the other hand, an interrupt-based alternative would need OS intervention, more complex hardware/software, and would be slower. In any case, for instance, dedicated threads for checking execution completion, and inter-thread synchronization mechanisms, can be flexibly adapted to various software and timing requirements.

The ISBUSY instruction has two operands: `accId` to identify the accelerator and a destination register. Same as CHECK, the output is written in this register when the response arrives from the accelerator. `procId` is added in the ACM to avoid a process enquiring about another process' computation, collecting information that it should not be allowed to have.

The result can be either "0" or "1" to indicate that the requested accelerator is free or busy, respectively, or an error code to inform the caller of a previous error (i.e., non-existent `opId` requested in EXEC instruction). The process can read the destination register to implement its logic depending on the job completion state.

6) RELEASE INSTRUCTION

RELEASE `<accId>`

When the process has finished its work and does not need the accelerator anymore, it can release it through the RELEASE instruction. As in the RESERVE case, `accId` is used to identify the accelerator and `procId` to identify the calling process. When the accelerator is released, it is assigned to the first process in the reservation queue and sets to the *Reserved* state. If the queue is empty, it enters the *Idle* state.

A RELEASE request is executed immediately only if the accelerator is not busy. If it is busy, the accelerator is released as soon as it completes the execution.

IV. EXPERIMENTAL RESULTS

To evaluate our framework, we implement the proposed ISA extension framework on gem5 full-system simulator [36] modeling a quad-core RISC-V processor with 3-level on-chip caches. Table 2 lists the details of the system configuration.

To conduct our experiments, we extend the simulator with specific SimObjects representing accelerators and the ANoC. The ANoC is connected to the CPU cores through request-response port pairs, where the cores can push commands that cause request ACM sending, and the interconnect can respond with response ACMs. The ANoC is also connected to the accelerators to deliver packets coming from the cores and retrieve packets addressed to the cores. Accelerators are also connected to the memory system with other ports that are used to load/store data. The additional instructions in our proposal require specific classes representing them, the extension of the decoding phase to associate opcodes, and the extension of the execution stage of the CPU pipeline to implement the interaction with the ANoC.

We integrate four different accelerators in the simulated environment in order to demonstrate the flexibility of the proposed approach both at ISA level and at HW-support level:

- Vector: A vector accelerator that performs the most common *vector floating-point* operations (e.g., `add`, `mul`, `dot-product`, `reduce-sum`, etc.) on vectors, with a variable number of lanes;
- FFT: An FFT accelerator that calculates the Fast Fourier Transform of an input array of complex floating-point numbers;
- Crypto: A cryptographic accelerator that encrypts/decrypts an input array;
- Conv: A convolutional accelerator that calculates the convolution between two floating-point tensors of variable size.

All the modeled accelerators receive requests from the CPU via the 64-bit ANoC discussed before. They access the memory hierarchy through a load-port and a store-port connected to the L3 cache, and they work according to a three-stage load-execute-store pipeline in a *strip mining* fashion.

As a baseline comparison, we implement a driver-based solution, which is the most widespread method to communicate with accelerators [2], [4]. Then, we compare our proposal and the generally-used driver-based interfacing from a performance standpoint in the case of four benchmarks. We select various computation-intensive workloads to elucidate the differences.

We experimentally measured the latency associated with a driver-controlled accelerator by implementing and benchmarking a classic core-accelerator interaction mediated by a Linux driver, and found that it is about 9000 clock cycles. The driver is organized into two main components: a user-space driver and a kernel-space one. The former provides a convenient API that is intended to be used in user

TABLE 3. Convolutional accelerator.

Frequency	1 GHz
RESERVE	3 cycles
CHECK	3 cycles
TRANSFER	1 cycles
EXEC	1 cycle
RELEASE	3 cycles
ISBUSY	1 cycle
vector add	2 cycles
vector sub	2 cycles
vector mul	5 cycles
vector div	14 cycles
vector slide	1 cycle
vector min	4 cycles
vector max	4 cycles

(a) Vector accelerator

Frequency	1 GHz
RESERVE	3 cycles
CHECK	3 cycles
TRANSFER	1 cycles
EXEC	1 cycle
RELEASE	3 cycles
ISBUSY	1 cycle
FFT 4 elems	2 cycles
FFT 16 elems	9 cycles
FFT 64 elems	52 cycles
FFT 256 elems	277 cycles
FFT 1K elems	1380 cycles
FFT 4K elems	21600 cycles
FFT 16K elems	76900 cycles
FFT 64K elems	306300 cycles
FFT 256K elems	1285400 cycles
FFT 1M elems	5407500 cycles

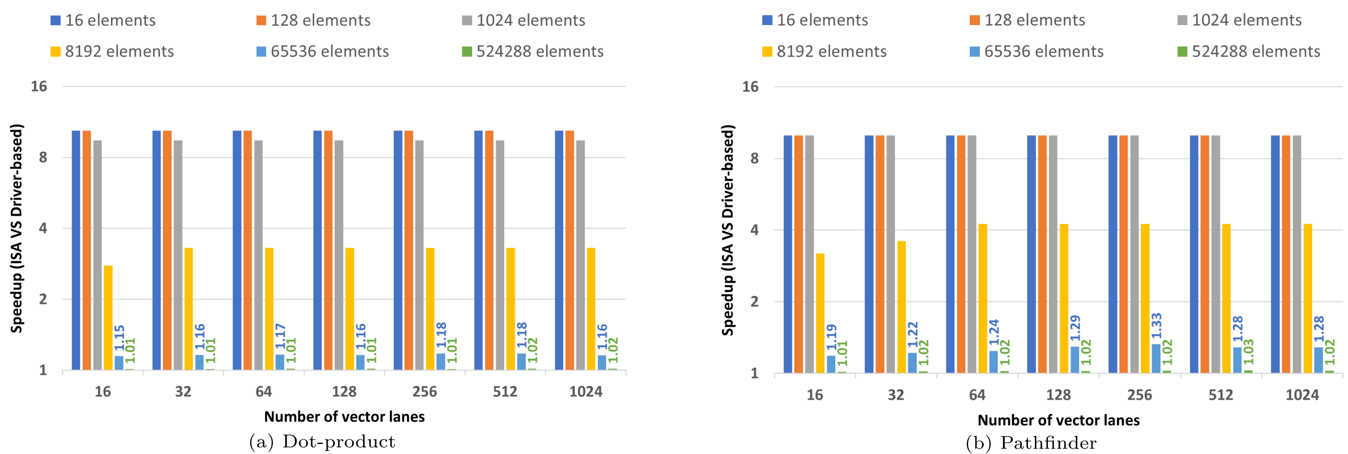
(b) FFT accelerator

Frequency	1 GHz
RESERVE	3 cycles
CHECK	3 cycles
TRANSFER	1 cycles
EXEC	1 cycle
RELEASE	3 cycles
ISBUSY	1 cycle
encr AES128 block	356 cycles
decr AES128 block	356 cycles

(c) Cryptographic accelerator

Frequency	606 MHz
RESERVE	3 cycles
CHECK	3 cycles
TRANSFER	1 cycles
EXEC	1 cycle
RELEASE	3 cycles
ISBUSY	1 cycle
throughput	2.090 TFLOPs

(d) Convolutional accelerator



(a) Dot-product (b) Pathfinder

FIGURE 4. Vector accelerator. Speedup of our proposal (ISA) VS a driver-based interface. Two benchmarks with five workload sizes (128, 1K, 8K, 64K, 512K elements) are executed. The number of lanes of the accelerator is varied between 16 and 1024.

code. The latter, which is invoked by the former, implements the necessary kernel-space functions to interact with the device file. The programmer willing to communicate with the accelerator must invoke an API function provided by the user-space driver. In its body, this function calls a write operation on the device file, whose implementation is provided by the kernel-space driver as a *file operation* [37], [38]. Here, concurrent accesses from different processes are serialized through OS semaphores, and the intended command is delivered to the accelerator by *memcpying* data on a memory-mapped region of the accelerator (e.g., its control registers).

Our proposal does not need the traversing of all these layers and can work with no driver in the middle. Since we map accelerator-oriented commands on user-space instructions, these can be directly invoked in user code with inline assembly. Concurrent commands are serialized by the ANoC, and no reservation logic needs to be implemented by the kernel because we move the *reservation queue* and its management in the hardware layer. Thus, the interaction latency of our proposal is dominated by the execution time of our instructions, the interconnection and the proposed accelerator interfacing logic, which are all taken into account in the simulator. The interconnection latency, in particular, is set to 16 cycles, and includes three contributions: the time spent in the core-network interface, the time spent in the accelerator-network interface, and the routing time.

Table 3 lists the latencies adopted to simulate the accelerators. These latencies can be divided in two groups: those related to the interpretation of the commands received as request ACMs, generated as a product of an accelerator-oriented instruction execution; and those related to the execution of the accelerated task in the accelerator. For those in the first group, we assign the same latencies to all the accelerators. For those in the second group, we adopt latencies associated to real accelerators from the literature. The vector accelerator is modeled after the RISC-V vector functional unit described by Ramírez et al. [39]. The FFT accelerator is modeled as described by Chen et al. in [40]. The cryptographic accelerator is modeled according to the description by Good and Benaissa in [41]. Finally, the convolutional accelerator is modeled as a single DaDianNao chip [42].

TABLE 4. Tensor dimensions of the convolutional layers of LeNet-5, AlexNet, and ResNet CNNs. H, W, C, and N represent height, width, number of channels, and number of filters, respectively.

	Input			Filter			
	H	W	C	H	W	C	N
LeNet-5	32	32	1	5	5	1	6
	14	14	6	5	5	6	16
	5	5	16	5	5	16	120
AlexNet	227	227	3	11	11	3	96
	27	27	96	5	5	96	256
	13	13	256	3	3	256	384
	13	13	384	3	3	384	384
	13	13	384	3	3	384	256
ResNet	228	228	3	7	7	3	64
	58	58	64	3	3	64	64
	30	30	64	3	3	64	128
	16	16	128	3	3	128	256
	9	9	256	3	3	256	512

Figure 4 shows the performance of two benchmarks executed on the Vector accelerator. These are *dot-product*, which calculates the dot-product between two vectors, and *pathfinder*, a graph-traversal benchmark from the Rodinia Benchmark Suite [43]. For both, the workload varies from 128 to 512K `double` elements. The figure compares the performance obtained with our proposed interface with a conventional driver-based one, showing the speedup of our proposal with respect to the driver-based solution. The workload size and the number of lanes supported by the accelerator are varied.

Accelerator *FFT* calculates the Fast Fourier Transform of arrays of `complex` numbers. Figure 5 shows the performance of a benchmark in which an array of `complex` elements is offloaded to the accelerator, transformed, and the output stored in main memory. The figure shows the speedup of our proposed interface with respect to a driver-based one, varying the workload size from 256 to 1M elements.

Accelerator *Crypto* encrypts/decrypts arrays of data using the NIST AES standard. It uses the AES128 configuration, with a 128-bit key and encryption/decryption of 128-bit blocks. Figure 6 shows the performance of a benchmark in which an array of 256 up to 1M bytes is offloaded to the accelerator, encrypted, and the result is retrieved. Also in this case, we show the speedup of our proposed interface with respect to a driver-based one.

Accelerator *Conv* calculates the convolution between two tensors of single-precision floating-point values. Figure 7 shows the performance speedup of our proposal with respect to the driver-based solution, in the case of a single benchmark in which the convolution between two tensors is calculated. The workloads are characterized by the dimensions of the tensors involved. These dimensions reflect those used in the convolutional layers of three popular Convolutional Neural Networks (CNNs): LeNet-5 [44], AlexNet [45], and ResNet [46]. Table 4 lists the sizes associated to each layer.

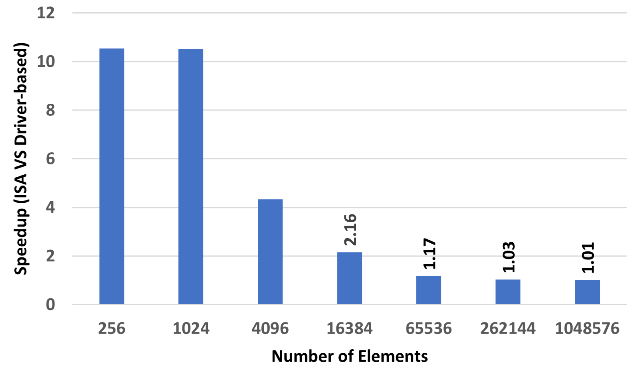


FIGURE 5. FFT accelerator. Speedup of our proposal (ISA) in comparison with a driver-based interface. One benchmark with seven workload sizes (256, 1K, 4K, 16K, 64K, 256K, 1M elements) is executed.

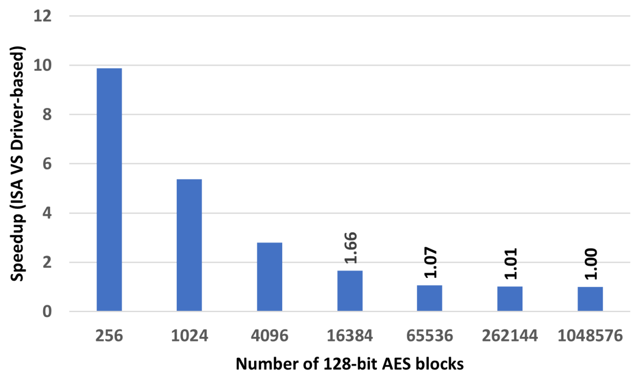
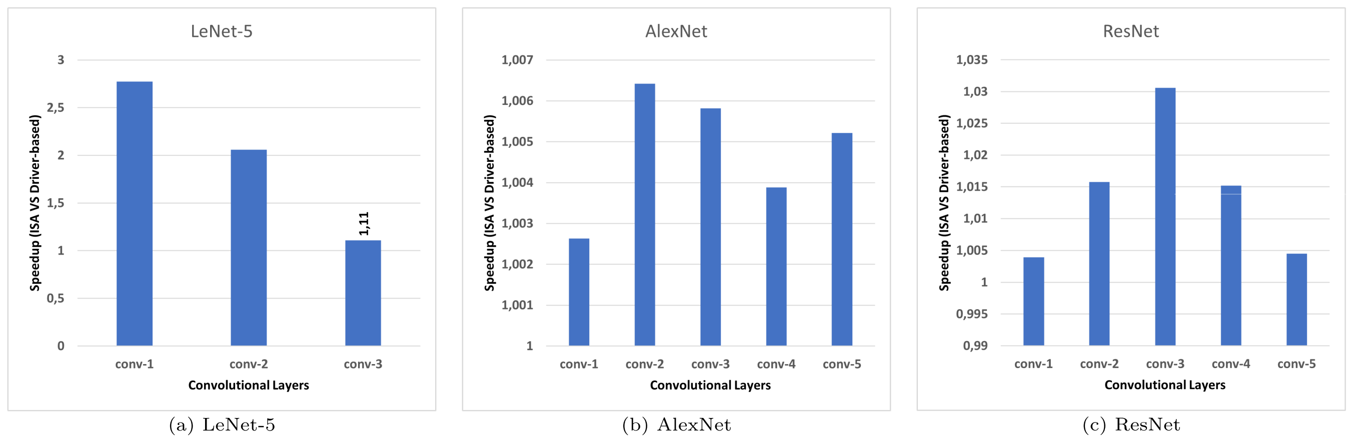


FIGURE 6. Crypto accelerator. Speedup of our proposal (ISA) in comparison with a driver-based interface. One benchmark with seven workload sizes (256, 1K, 4K, 16K, 64K, 256K, 1M AES128 blocks) is executed.

The figures describe a similar scenario. For workloads with small and medium granularity, the weight of communication and management (C&M in the following) dominates the total execution time. Since our proposal permits a significantly lower C&M overhead, we reach up to 10x speedup in our favor with respect to a classic driver-based solution. Our advantage gets smaller when the workload granularity becomes larger, since the execution time grows accordingly, while the C&M cost remains constant independently of the workload size. C&M gets diluted in the total runtime for the biggest workloads, ultimately becoming negligible. In those cases, our solution is on par with a driver-based one, with a speedup that is approximately 1×.

From another point of view, a little sensitivity to the number of lanes can be observed in Figure 4 for both dot-product and pathfinder. In general, a higher number of lanes reduces the computation cost and increases the relative weight of interface overhead, giving a slightly higher speedup in our favor. This phenomenon is especially evident for pathfinder with a workload of 8192 elements, with the speedup growing from 3.19× with 16 lanes up to 4.24× with 1024 lanes.

Overall, the performance advantage given by our proposal is more evident for smaller workloads. However, hardware



(a) LeNet-5 (b) AlexNet (c) ResNet

FIGURE 7. Convolutional accelerator. Speedup of our proposal (ISA) VS a driver-based interface. We execute a single convolution between two tensors. The tensor dimensions are those used in the convolutional layers of three popular CNNs: LeNet-5, AlexNet, and ResNet.

acceleration is less palatable for extremely small workloads, since a pure CPU implementation may be competitive: it would not leverage the high-performance, high-efficiency special-purpose hardware to perform ad-hoc calculations as a dedicated accelerator, but it would have the advantage of not requiring expensive data migrations towards the accelerator and back. Considering dot-product, which is the benchmark where the performance advantage of a dedicated accelerator over the CPU is smaller, the proposed ISA-based accelerated version becomes more convenient than a CPU-only implementation for workloads with around 1200 elements and greater. Conversely, for the driver-based version, the same happens for workloads with around 8500 elements and greater. Thus, our proposal performs better than the two alternatives (i.e., CPU-only and driver-based acceleration) in the middle ranges, where the granularity of tasks delivered to the accelerator are large enough to justify an accelerated implementation, but not so large to make a driver-based solution as convenient as our proposal. Therefore, our proposal proves fundamental to enable the opportunity of hardware acceleration, effectively decreasing the workload size threshold at which hardware acceleration becomes amenable to gain performance, and breaking even with driver-based acceleration for tasks with the larger granularity.

We estimate the area and power consumption overheads besides performance. The proposed framework architecture (shown in Figure 1) added a reservation queue with four 32-bit entries for every accelerator. It also requires three 32-bit registers as the status register and buffers. One bit for busy Flip-Flop is also needed. Besides, a control unit includes a few multiplexers, decoders, and registers. Therefore, the proposed framework contains at most 1K bits. On the other hand, each accelerator can provide at least a local memory with a size of 256KB or more. Compared with the total area an accelerator can obtain, the added area of these modules and circuits is negligible (less than 1%). From the power consumption perspective, the overhead is proportional to the number of transistors added on the chip to implement the hardware portion of our proposal. Since we have shown that they are less than 1% for each accelerator, the power increase is modest. However, from an energy perspective, their addition causes a significant reduction in communication time with respect to a driver-based solution, and thus the total dissipated energy decreases accordingly.

V. CONCLUSION

In this paper, we presented a RISC-V based ISA extension to manage the interaction between general-purpose cores and a vast class of on-chip accelerators. Its main purpose is to move the communication and management logic from the usual driver-layer to the ISA and HW support as to gain in latency and generality. We presented the proposed six instructions and described the required architectural support. These can be utilized by the programmer to easily manage and exploit the accelerators without OS intervention.

We evaluated our proposal in gem5, using some benchmarks offloading work to four quite different accelerators to prove generality: one for simple vector operations, one for FFT, one for AES128 encryption/decryption, and one for tensor-tensor convolution. Comparison against a canonical driver-based solution demonstrated that our proposal is faster for every working set due to the significantly smaller overhead. The advantage is more evident for small workloads, reaching up to 10.38x, and is particularly amenable for middle workloads, where accelerated implementations based on our proposal score better performance than both a driver-based solution and a CPU-only one.

The core-side and accelerator-side controller logic are almost the same as standard interfaces and require a modest addition to the chip area (i.e., less than 1% on accelerator-side). These additional transistors can increase power consumption, but the communication time savings are such that the total dissipated energy decreases.

These results demonstrate that our proposed solution can be adopted to effectively improve the management

of integrated accelerators, reducing latency, providing a common interface for a broad variety of accelerators, and widening the range of workloads for which acceleration is advantageous.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
[2] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *J. Syst. Archit.*, vol. 129, Aug. 2022, Art. no. 102561.
[3] S. Patel and W. M. W. Hwu, "Accelerator architectures," *IEEE Micro*, vol. 28, no. 4, pp. 4–12, Jul. 2008.
[4] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 843–849.
[5] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. 52nd Annu. Design Autom. Conf.*, Jun. 2015, pp. 1–6.
[6] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu, and L. Benini, "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–19, Oct. 2017.
[7] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020.
[8] Huawei. (2020). *Huawei Kirin 990—Rethink Evolution*. [Online]. Available: https://www.consumer.huawei.com/en/campaign/kirin-990-series
[9] J. Cross. (Oct. 2020). *A14 Bionic FAQ: What You Need to Know About Apple's 5 nm Processor*. [Online]. Available: https://www.macworld.com/article/234595/a14-bionic-faq-performance-features-cpu-gpu-neural-engine.html
[10] NVIDIA. (2020). *NVIDIA Ampere GA102 GPU Architecture*. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf
[11] AMD. (2019). *RDNA Architecture*. [Online]. Available: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf
[12] Google. (2020). *Cloud Tensor Processing Units (TPUs)*. [Online]. Available: https://cloud.google.com/tpu/docs/tpus
[13] N. Mohammedali and M. O. Agyeman, "A study of reconfigurable accelerators for cloud computing," in *Proc. 2nd Int. Symp. Comput. Sci. Intell. Control*, New York, NY, USA, Sep. 2018, pp. 1–5.
[14] J. Dongarra and P. Luszczek, *TOP500*. Boston, MA, USA: Springer, 2011, pp. 2055–2057.
[15] ARM. *AMBA Overview*. Accessed: Sep. 2022. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba
[16] CCIX Consortium. *CCIX Website*. Accessed: Sep. 2022. [Online]. Available: https://www.ccixconsortium.com/
[17] CXL Consortium. *Compute Express Link: The Breakthrough CPU-to-Device Interconnect*. Accessed: Sep. 2022. [Online]. Available: https://www.computeexpresslink.org/
[18] NVIDIA. (Aug. 2021). *CUDA C Programming Guide*. [Online]. Available: https://www.docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[19] Khronos OpenCL Working Group. (Nov. 2012). *The OpenCL Specification, Version 1.2*. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
[20] G. Rao, J. Chen, J. Yik, and X. Qian, "IntersectX: An efficient accelerator for graph mining," 2020, *arXiv:2012.10848*.
[21] N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "Arise machines: Extending processors with hybrid accelerators," in *Proc. Int. Workshop Appl. Reconfigurable Comput.*, 2008, pp. 196–208.
[22] N. P. Jachimiec, F. Martinez-Vallina, and J. Saniie, "Acceleration of finite field arithmetic algorithms in embedded processing platforms utilizing instruction set extensions," in *Proc. IEEE Int. Conf. Electro/Inf. Technol.*, May 2007, pp. 135–139.
[23] E. Liventsev, A. Silantiev, E. Primakov, and O. Telminov, "Extending MIPSfpga instruction set for navigation data processing," in *Proc. IEEE Conf. Russian Young Researchers Electr. Electron. Eng. (EIConRus)*, 2017, pp. 480–484.
[24] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 239–280, Aug. 2020.
[25] G. H. Eisenkraemer, F. G. Moraes, L. L. de Oliveira, and E. Carara, "Lightweight cryptographic instruction set extension on Xtensa processor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.
[26] D. W. Todd, "Tightly coupling the PicoRV32 RISC-V processor with custom logic accelerators via a generic interface," Ph.D. dissertation, Dept. Comput. Eng., Clemson Univ., Clemson, SC, USA, 2021.
[27] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 43–56.
[28] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 416–429.
[29] V. Patil, A. Raveendran, P. M. Sobha, A. D. Selvakumar, and D. Vivian, "Out of order floating point coprocessor for RISC V ISA," in *Proc. 19th Int. Symp. VLSI Design Test*, Jun. 2015, pp. 1–7.
[30] S. Bartolini, R. Giorgi, and E. Martinelli, "Instruction set extensions for cryptographic applications," in *Cryptographic Engineering*. Boston, MA, USA: Springer, 2009, pp. 191–233.
[31] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 393–405.
[32] S. Mazzola, "ISA extensions in the snitch processor for signal processing," M.S. thesis, Dept. Comput. Eng., Polytech. Univ. Turin, Turin, Italy, 2021.
[33] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi, "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 *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 52–64.
[34] ARM. (2009). *ARM1176JZF-S Technical Reference Manual*. [Online]. Available: https://documentation-service.arm.com/static/5e8e294efd977155116a6ca3?token=
[35] A. Waterman and K. Asanovic. (Feb. 2020). *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. [Online]. Available: http://uglyduck.vajn.icu/PDF/GigaDevice/RISCV-Spec.pdf
[36] J. Lowe-Power et al., "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.
[37] *LINUX Device Drivers*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
[38] The Kernel Development Community. (2022). *The Linux Kernel Documentation*. [Online]. Available: https://docs.kernel.org/
[39] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A RISC-V simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–30, Dec. 2020.
[40] X. Chen, Y. Lei, Z. Lu, and S. Chen, "A variable-size FFT hardware accelerator based on matrix transposition," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 10, pp. 1953–1966, Oct. 2018.
[41] T. Good and M. Benaissa, "692-nW advanced encryption standard (AES) on a 0.13-μm CMOS," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 12, pp. 1753–1757, Dec. 2010.
[42] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.
[43] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.
[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

ELHAM CHESHMIKHANI received the B.Sc. degree from the Iran University of Science and Technology (IUST), in 2011, the M.Sc. degree from the Amirkabir University of Technology (Tehran Polytechnic), in 2013, and the Ph.D. degree from the Sharif University of Technology (SUT), Tehran, Iran, in 2020, all in computer engineering. Since February 2021, she has been a Postdoctoral Researcher with the Department of Information Engineering and Mathematical Sciences, University of Siena. Her research interests include hardware accelerator, RISC-V ISA design, emerging nonvolatile memory technologies, dependability analysis, fault tolerance, and storage systems. She was a member of the Design and Analysis of Dependable Systems (DADS) at AUT, from 2011 to 2015. She has been a member of the Dependable Systems Laboratory (DSL) and the Data Storage, Networks and Processing Laboratory (DSN) with SUT, since 2015 and 2017, respectively. She received the Best Paper Award from IEEE/ACM Design, Automation, and Test in Europe (DATE), in 2019, during her Ph.D. career.

BIAGIO PECCERILLO is a Postdoctoral Researcher with the Department of Information Engineering and Mathematical Sciences, University of Siena. His research interests include heterogeneous architectures, hardware accelerators, parallel algorithms, and productivity-oriented high-level abstraction mechanisms. He participated in various Research and Development projects involving hardware accelerators, haptic algorithms in virtual and augmented reality environments, and pharmaceutical supply chain simulation.

ANDREA MONDELLI received the Ph.D. degree in computer architecture. He is CPU Chief Architect with Huawei and a Principal Researcher of cybersecurity and architecture design. He is a Technology Manager and responsible for Huawei research projects collaborations with European universities. He has was a Researcher and an Architect in various countries, such as Italy, USA, France, China, and UK. He published multiple manuscripts and conference papers and a book on memory coherence protocols. His research interests include high performance and low power CPUs. He was part of RISC-V International as the Chair of virtual memory area.

SANDRO BARTOLINI is an Associate Professor with the Department of Information Engineering and Mathematical Sciences, University of Siena. His main research interests include high-performance chip multi processors (CMPs), new approaches to productive programming of heterogeneous architectures (CPUs and GPUs), integrated photonics for CMPs, feedback-driven compiler optimizations for cache hierarchy performance and low power, and hardware accelerators. He has led and participated in various Research and Development projects. He is an Active Member of the HiPEAC NoE. He is an Associate Editor of the *EURASIP Journal on Embedded Computing*. He is the Co-Guest Editor of the *Transactions on High Performance Architectures and Compilers* (Springer) journal and *ACM SigArch Computer Architecture Newsletter*.

Open Access funding provided by ‘Università degli Studi di Siena’ within the CRUI CARE Agreement