Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation

Michael Platzer
TU Wien, Institute of Computer Engineering, Austria

Peter Puschner
TU Wien, Institute of Computer Engineering, Austria

Abstract

In this work, we present Vicuna, a timing-predictable vector coprocessor. A vector processor can be scaled to satisfy the performance requirements of massively parallel computation tasks, yet its timing behavior can remain simple enough to be efficiently analyzable. Therefore, vector processors are promising for highly parallel real-time applications, such as advanced driver assistance systems and autonomous vehicles. Vicuna has been specifically tailored to address the needs of real-time applications. It features predictable and repeatable timing behavior and is free of timing anomalies, thus enabling effective and tight worst-case execution time (WCET) analysis while retaining the performance and efficiency commonly seen in other vector processors. We demonstrate our architecture's predictability, scalability, and performance by running a set of benchmark applications on several configurations of Vicuna synthesized on a Xilinx 7 Series FPGA with a peak performance of over 10 billion 8-bit operations per second, which is in line with existing non-predictable soft vector-processing architectures.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Real-time Systems, Vector Processors, RISC-V

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.1

Supplementary Material *Software*: `https://github.com/vproc/vicuna`

1 Introduction

Worst-Case Execution Time (WCET) analysis, which is essential to determine the maximum execution time of tasks for real-time systems [46], has struggled to keep up with the advances in processor design. Numerous optimizations such as caches, branch prediction, out-of-order execution, and speculative execution have made the timing analysis of processing architectures increasingly complex [45]. As a result, the performance of processors suitable for real-time systems usually lags behind platforms optimized for average computational throughput at the cost of predictability. Yet, the performance requirements of real-time applications are growing, particularly in domains such as advanced driver assistance systems and self-driving vehicles [23], thus forcing system architects to use multi-core architectures and hardware accelerators such as Graphics Processing Units (GPUs) in real-time systems [13]. Analyzing the timing behavior of such complex heterogeneous systems poses additional challenges as it requires a timing analysis of the complex interconnection network in addition to analyzing the individual processing cores of different types and architectures [36, 9].

However, current trends motivated by the quest for improved energy-efficiency and the emergence of massively data-parallel workloads [8] have revived the interest in architectures that might be more amenable to WCET analysis [29]. In particular, vector processors are promising improved energy efficiency for data-parallel workloads [7] and have the potential to reduce the performance gap between platforms suitable for time-critical applications and mainline processors [29].

Vector processors are single-instruction multiple-data (SIMD) architectures, operating on vectors of elements instead of individual values. The vector elements are processed simultaneously across several processing elements as well as successively over several cycles [2]. A single vector instruction can operate on a very large vector, thus amortizing the overhead created by fetching and decoding the instruction, which does not only increase its efficiency [4] but also means that complex hardware-level optimizations become less effective [29]. Therefore, vector processors can drop some of these optimizations and thus improve timing predictability without notable performance degradation.

While vector processors have the potential to greatly simplify timing analysis compared to other parallel architectures, existing vector processing platforms retain features that impact timing-predictability, such as out-of-order execution or banked register files [5]. Even if some vector architectures have simple in-order pipelines, they still exhibit timing anomalies (i.e., undesired timing phenomena which threaten timing predictability). Timing anomalies occur, for instance, when memory accesses are not performed in program order [16], such as when memory accesses by the vector unit interfere with accesses from the main core.

In this paper, we present a novel vector coprocessor addressing the needs of time-critical applications without sacrificing performance. Our key contributions are as follows:

1. We present a timing-predictable 32-bit vector coprocessor implemented in SystemVerilog that is fully compliant with the version 0.10 draft of the RISC-V vector extension [34]. All integer and fixed-point vector arithmetic instructions, as well as the vector reduction, mask, and permutation instructions described in the specification, have been implemented. Vicuna is open-source and available at `https://github.com/vproc/vicuna`.
2. We integrate our proposed coprocessor with the open-source RISC-V core Ibex [37] and show that this combined processing system is free of timing anomalies while retaining a peak performance of 128 8-bit multiply-accumulate (MAC) operations per cycle. The combined processing system runs at a clock frequency of 80 MHz on Xilinx 7 Series FPGAs, thus achieving a peak performance of 10.24 billion operations per second.
3. We evaluate the effective performance of our design on data-parallel benchmark applications, reaching over 90 % efficiency for compute-bound tasks. The evaluation also demonstrates the predictability of our architecture as each benchmark program always executes in the exact same number of CPU cycles.

This work is organized as follows. Section 2 introduces prior work in the domains of parallel processing and vector architectures. Then, Section 3 presents the design of our vector coprocessor Vicuna and Section 4 analyzes the timing behavior of our processing system. Section 5 evaluates its performance on several benchmark algorithms, and Section 6 concludes this article.

2 Background and Related Work

This section gives an overview of existing parallelized computer architectures and vector processors in particular and compares them to our proposed timing-predictable vector coprocessor Vicuna. Table 1 summarizes the main aspects.

2.1 Parallel Processing Architectures

In the mid-2000s, power dissipation limits put an end to the acceleration of processor clock frequencies, and computer architects were forced to exploit varying degrees of parallelism in order to further enhance computational throughput. A relatively simple approach is to

Table 1 Performance and timing predictability of parallel computer architectures.

Processor Architecture	Multi-Core CPU	General-purpose GPU	Domain-Specific Accelerators	Existing Vector Processors	Timing-Predictable Platforms	**Vicuna (Our work)**
General-purpose	✓	✓		✓	✓	✓
Efficient parallelism		✓	✓	✓		✓
Timing-predictable			✓		✓	✓
Max. OPs per sec ($\cdot 10^9$) FPGA / ASIC	2.2^{*} / $1\,200^{**}$	$3.2^{\dagger}$ / $35\,000^{\dagger\dagger}$	$5\,000^{\ddagger}$ / $45\,000^{\ddagger\ddagger}$	$15^{\S}$ / $128^{\S\S}$	$2.4^{\P}$ / $49^{\P\P}$	10 / —

* 16-core Cobham LEON3
** 344-core Ambric Am2045B
† FlexGrip soft GPU [1]
†† NVIDIA RTX 3090
‡ Srinivasan et al. [42]
‡‡ Google TPU [21]
§ 32-lane VEGAS [6]
§§ 16-lane PULP Ara [5]
¶ 15-core T-CREST Patmos [38]
¶¶ 8-core ARM Cortex-R82

replicate a processor core several times, thus creating an array of independent cores each executing a different stream of instructions. This multiple-instruction, multiple-data (MIMD) paradigm [11] is ubiquitous in today's computer architectures and has allowed a continued performance increase. Timing-predictable multi-core processors have been proposed for time-critical parallel workloads, most notably the parMERASA [43] and the T-CREST [38] architectures, which demonstrated systems with up to 64 and 15 cores, respectively. A similar timing-predictable multi-core architecture utilizing hard processing cores connected by programmable logic has been implemented recently on an Multiprocessor System-on-Chip (MPSoC) platform [14]. However, several of the workloads capable of efficiently exploiting this parallelism are actually highly data-parallel, and as a consequence, the many cores in such a system frequently all execute the same sequence of instructions [7]. The fetching and decoding of identical instructions throughout the cores represent a significant overhead and increase the pressure on the underlying network infrastructure connecting these cores to the memory system [28, 41]. Consequently, the effective performance of a multi-core system does not scale linearly as more cores are added. For the T-CREST platform, Schoeberl et al. report that the worst-case performance for parallel benchmark applications scales only logarithmically with the number of cores [38]. As an alternative to multi-core architectures, some timing-predictable single-core processors exploit parallelism by executing multiple independent hardware threads [26, 50], thus avoiding the overhead of a complex interconnection network. Yet, the scalability of this approach is limited since it does not increase the available computational resources.

An architecture that overcomes many of the limitations of multi- and many-core systems for highly parallel workloads are general-purpose GPUs (also referred to as GPGPUs) [31]. GPUs utilize data-parallel multithreading, referred to as the single-instruction multiple-threads (SIMT) paradigm [27], to achieve unprecedented energy-efficiency and performance. GPUs are used as data-parallel accelerators in various domains and have found their way into safety-critical areas such as autonomous driving [23, 13]. However, their use in hard real-time systems still poses challenges [9]. GPUs are usually non-preemptive, i.e., tasks cannot be interrupted, which requires software-preemption techniques to be used instead [13]. Also, contention among tasks competing for resources is typically resolved via undisclosed arbitration schemes that do not account for task priorities [10].

Recently, special-purpose accelerators emerged as another type of highly parallel platform that sacrifices flexibility and often precision [42] to achieve impressive performance for domain-specific tasks. For instance, the Tensor Processing Unit (TPU) [21] is capable of 65536 8-bit MAC operations in one cycle, achieving a peak performance of $45 \cdot 10^{12}$ operations per second at a clock frequency of 700 MHz. Due to their simple application-specific capabilities, the timing behavior of these accelerators is generally much easier to analyze [29]. While domain-specific accelerators achieve impressive performance for a small subset of applications, they are very inefficient at or even incapable of running other important algorithms, such as Fourier Transforms, motion estimation, or encryption with the Advanced Encryption Standard (AES). By contrast, a vector processor can execute any task that can be run on a conventional processor.

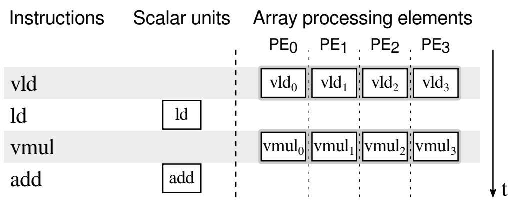
As an alternative to parallelizing tasks across several cores or threads, single-instruction multiple-data (SIMD) arrays have been added to several Instruction Set Architectures (ISAs). These are usually fixed-size arrays using special functional units, one for each element in the array, to apply the same operation to the entire array at once. However, array processors require that the computational resources are replicated for each element of the longest supported array [5].

2.2 Vector Processors

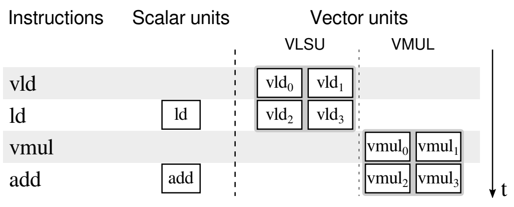
Vector processors are a time-multiplexed variant of array processors. Instead of limiting the vector length by the number of processing elements, a vector processor has several specialized execution units that process elements of the same vector across multiple cycles, thus enabling the dynamic configuration of the vector length [7]. Fig. 1 shows how an instruction stream with interleaved scalar and vector instructions executes on an array processor and a vector processor, respectively. In an array processor, the entire vector of elements is processed at once, and the processing elements remain idle during the execution of scalar instructions. In the vector processor, functionality is distributed among several functional units, which can execute in parallel with each other as well as concurrently with the scalar units.

Vector processors provide better energy-efficiency for data-parallel workloads than MIMD architectures [7] and promise to address the van Neumann bottleneck very effectively [4]. A single vector instruction can operate on a very large vector, which amortizes the overhead created by fetching and decoding the instruction. In this regard, vector processors even surpass GPUs, which can only amortize the instruction fetch over the number of parallel execution units in a processing block [5].

Several supercomputers of the 1960s and 1970s were vector processors, such as the Illiac IV [19] or the Cray series [35]. These early vector processors had functional units spread across several modules containing thousands of ICs in total. At the end of the century, they



(a) Array processor.



(b) Vector processor.

Figure 1 Comparison of the execution patterns of array and vector processors. Instructions prefixed with a *v* operate on a vector of elements, while the rest are regular scalar instructions.

were superseded by integrated microprocessor systems, which surpassed their performance and were significantly cheaper [2]. While disappearing from the high-performance computing domain, vector processors have continued their existence as general-purpose accelerators in Field-Programmable Gate Arrays (FPGAs). Several soft vector processors have been presented, such as VESPA [48], which adds a vector coprocessor to a 3-stage MIPS-I pipeline, VIPERS [49], a single-threaded core with a vector processing unit, VEGAS [6], a vector coprocessor using a cacheless scratchpad memory, VENICE [39], an area-efficient improved version of VEGAS, or MXP [40], which added additional support for fixed-point computation.

In addition to FPGA-based accelerators, vector processors have also been explored as energy-efficient parallel computing platforms. Lee et al. [25] proposed a vector architecture named Hwacha, which is based on the open RISC-V ISA. The instruction set for Hwacha has been implemented as a custom extension. Despite sharing some features, it is incompatible with the more recent official RISC-V vector extension. One of the first vector processors based on the new RISC-V V extension is Ara, developed by Cavalcante et al. [5], as a coprocessor for the RISC-V core Ariane. Another recent architecture implementing the RISC-V V extension named RISC-V^2 has been proposed by Patsidis et al. [32].

While existing vector processors are less complex and easier to analyze than other parallel architectures, they still use speed-up mechanisms which are a source of timing anomalies, such as run-time decisions for choosing a functional unit [44], banked register files, and greedy memory arbitration [16]. By contrast, our proposed vector processor avoids such mechanisms, with negligible impact on its performance thanks to the vector processing paradigm's inherent effectiveness. Vicuna is free of timing anomalies and hence suitable for compositional timing analysis.

3 Architecture of Vicuna

This section introduces the architecture of Vicuna, a highly configurable, fully timing-predictable 32-bit in-order vector coprocessor implementing the integer and fixed-point instructions of the RISC-V vector extension. The RISC-V instruction set is an open standard ISA developed by the RISC-V foundation. It consists of a minimalist base instruction set supported by all compliant processors and several optional extensions. The V extension adds vector processing capabilities to the instruction set. RISC-V and the V extension are supported by the GNU Compiler Collection (GCC) and the LLVM compiler.

Vicuna is a coprocessor and must be paired with a main processor. We use the 32-bit in-order RISC-V core Ibex, developed initially as part of the PULP platform under the name Zero-riscy [37], as the main processor. Ibex is a small core with only two pipeline stages: an instruction fetch stage and a combined decode and execute stage. Ibex executes all non-vector instructions, which we refer to as scalar instructions.

Vicuna is connected to the main core with a coprocessor interface through which instruction words and the content of registers are forwarded from the main core to the coprocessor, and results can be read back. We added a coprocessor interface to Ibex to extend it with Vicuna. Instruction words are forwarded to the vector core via this interface if the major opcode indicates that it is a vector instruction. In addition to the instruction word, scalar operands from the main core's register file are also transmitted to the coprocessor since these are required by some vector instructions which use the scalar registers as source registers, such as for instance, a variant of the vector addition which adds a scalar value to every element of a vector or the vector load and store instructions which read the memory address from a scalar register.

An overview of the architecture of Vicuna and its integration with Ibex as the main core is shown in Fig. 2. Vicuna comprises a decoder for RISC-V vector instructions, which parses and acknowledges valid vector instructions. Once Vicuna's decoder has successfully decoded a vector instruction, it acknowledges its receipt and informs the main core whether it needs to wait for a scalar result. If the vector instruction produces no scalar result but instead only writes to a vector register or memory, then the main core can proceed with further instructions in parallel with the vector instruction's execution on the coprocessor. However, when a vector instruction writes back to a register in the main core, then the main core stalls until the coprocessor has completed that instruction. Only four RISC-V vector instructions produce a scalar result. Hence this scenario occurs rarely. Decoded vector instructions are placed in an instruction queue where they await execution on one of the vector core's functional units. Vicuna is a strictly in-order coprocessor: Vector instructions from the instruction queue are issued in the order they are received from the main core. A vector instruction is issued as soon as any data hazards have been cleared (i.e., any instructions producing data required by that instruction are complete) and the respective functional unit becomes available.

Since our main goal is to design a timing-predictable vector processor, we refrain from any features that cause timing anomalies, such as run-time decisions for choosing functional units [44]. Both cores share a common 2-way data cache with a least recently used (LRU) replacement policy, which always gives precedence to accesses by the vector core. Once a vector instruction has been issued for execution on one of the functional units, it completes within a fixed amount of time that depends only on the instruction type, the throughput of the unit, and the current vector length setting. For vector loads and stores, the execution time additionally depends on the state of the data cache, which is the only source of timing variability. However, in-order memory access is guaranteed for scalar and vector memory

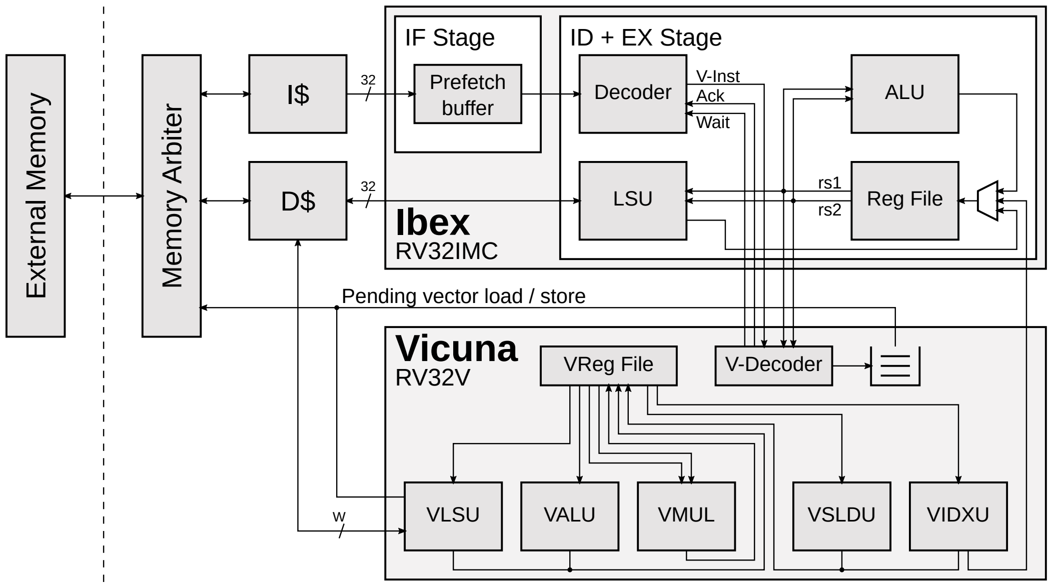


Figure 2 Overview of Vicuna's architecture and its integration with the main core Ibex. Both cores share a common data cache. To guarantee in-order memory access, the memory arbiter delays any access following a cache miss by the main core until pending vector load and store operations are complete. When accessing the data cache, the vector core always takes precedence.

operations by delaying any access following a cache miss in the main core until pending vector load and stores are complete. Note that vector load and store instructions stall the main core for a deterministic, bounded number of cycles since no additional vector instructions can be forwarded to the vector core while the main core is stalled. This method is an extension of the technique introduced by Hahn and Reineke [15] for the strictly in-order core SIC. Due to the simple 2-stage pipeline of Ibex, conflicting memory accesses between its two stages become visible simultaneously. In that situation, the memory arbiter maintains strict ordering by serving the data access first.

Vicuna comprises several specialized functional units, each responsible for executing a subset of the RISC-V vector instructions, which allows executing multiple instructions concurrently. The execution units do not process an entire vector register at once. Instead, during each clock cycle, only a portion of the vector register is processed, which may contain several elements that are processed in parallel. Most array processors and several vector processors are organized in lanes. Each lane replicates the computational resources required to process one vector element at a time. In such a system, the number of lanes determines the number of elements that can be processed in parallel, regardless of the type of operation. By contrast, Vicuna uses dedicated execution units for different instruction types that each process several elements at once. The ability to individually configure the throughput for each unit improves the performance of heavily used operations by increasing the respective unit's data-path width (e.g., widening the data-path of the multiplier unit).

Some of the RISC-V vector instructions do not process the vector registers on a regular element-wise basis. Instead, they feature an irregular access pattern, such as indexed instructions, which use one vector register's values as indices for reading elements from another register, or the slide instructions, which slide all elements in a vector register up or down that register. Vicuna uses different functional units for each vector register access pattern, which allows us to implement regular access patterns more efficiently and hence to improve the throughput of the respective unit, while complex access patterns require more cycles.

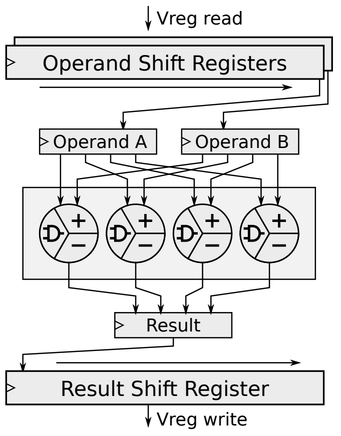
Vicuna comprises the following execution units:

- A *Vector Load and Store Unit* (VLSU) interfaces the memory and implements the vector memory access instructions.
- The *Vector Arithmetic and Logical Unit* (VALU) executes most of the arithmetic and logical vector instructions.
- A dedicated *Vector Multiplier* (VMUL) is used for vector multiplications.
- The *Vector Slide Unit* (VSLDU) handles vector slide instructions that move all vector elements up or down that vector synchronously.
- A *Vector Indexing Unit* (VIDXU) takes care of the indexing vector instructions. It is the only unit capable of writing back to a scalar register in the main core.

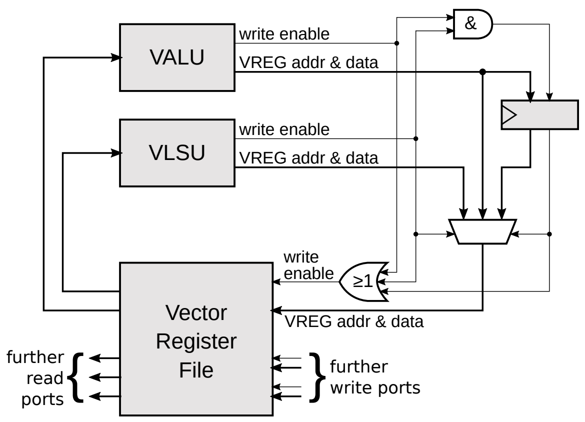
The VALU uses a fracturable adder for addition and subtraction, that consists of a series of 8-bit adders whose carry chains can be cascaded for wider operations. Four cascaded 8-bit adders perform four 8-bit, two 16-bit, or one 32-bit operation depending on the current element width. Similarly, the VMUL unit uses a fracturable multiplier to perform 8-bit, 16-bit, and 32-bit multiplications on the same hardware. Fracturable adders and multipliers are commonly used for FPGA-based vector processors. We base our implementation on the resource-efficient design that Chou et al. proposed for the VEGAS vector processor [6].

Selecting a relatively large sub-word from a large vector register consumes a substantial amount of logic resources. Therefore, we avoid sub-word selection logic for all functional units with a regular vector register access pattern. Instead, these units read the whole source vector registers into shift registers, as shown in Fig. 3 (a). The content of these is then shifted by the number of elements that can simultaneously be processed by the unit each cycle, thus making the next elements of the source vector register available to the processing pipeline. Similarly, the results are aggregated into another shift register that saves the computed elements until the entire vector is complete, upon which the whole vector register is written back to the register file. The amount of combinatorial logic resources consumed by the shift registers is less than those that are required by an index-based subword selection (they do, however, require some extra flip-flops for buffering the whole vector register).

Vicuna's vector register file contains 32 vector registers of configurable width. Multiple read and write ports are required in order to supply the execution units operating in parallel with operands and consume their results. We take advantage of the functional unit's shift registers, which fetch entire vector registers at once and accumulate results before storing a whole register, to implement both read and write port multiplexing. Each functional unit has a dedicated read port used to fetch the operand registers sequentially, storing them in shift registers from where they are consumed iteratively. This adds one extra cycle when fetching two operand registers but avoids the need for two read ports on each unit. As the only exception, the VMUL unit has two read ports to better support the fused multiply-add instruction, which uses three operands. Also, write ports are shared between units using the circuitry shown in Fig. 3 (b). Due to the accumulation of results in shift registers prior to write-back, a unit cannot write to the vector register file for two subsequent cycles. Hence, whenever a collision between two units occurs on a shared write port, one unit takes precedence and writes its result back first while the other unit writes its result into a temporary buffer, from where it is stored to the register file in the subsequent cycle. A second write request from the first unit cannot immediately follow the previous write. Hence this delayed write-back is guaranteed to succeed. Regardless of whether the write-back is delayed by one cycle or not, any data hazards of operations on units not taking precedence on their shared write port are cleared one cycle after the operation completes to maintain predictable instruction timings while accounting for a potentially delayed write-back.



(a) Organization of the vector ALU. Operand registers are read sequentially into shift registers and consumed over several cycles by processing a fixed-width portion each cycle. Results are again accumulated into a shift register before write-back.



(b) The VALU and VLSU share a common write port, with the VLSU always taking precedence. In case of a collision, the value and address of the VALU write request are temporarily saved and written to the vector register file in the next cycle. Neither unit can write for two subsequent cycles. Hence the delayed write always succeeds.

Figure 3 Reading and writing whole registers from the vector register file avoids subword selection logic and allows multiplexing of read and write ports without affecting timing predictability.

Although multiplexing of both read and write ports is used to reduce the required number of ports, the vector register file must still provide several concurrent ports. We decided against banked registers, which allow concurrent access to registers of different banks but introduce interdependencies between execution units which are a potential source of timing anomalies in case two registers within the same bank are accessed simultaneously. Since a large flip-flop-based register file does not scale well, we implemented it as multi-ported RAM. The design has been inspired by work from Laforest et al. [24], who investigated ways of constructing efficient multi-ported RAMs in FPGAs. We implemented it as an XOR-based RAM since this allows selectively updating individual elements of a vector register for masked operations.

4 Timing-Predictability

In this section, we analyze the timing-predictability of Vicuna and argue that it is free of timing anomalies, thus enabling compositional timing analysis.

Timing predictability and timing compositionality are both essential properties to avoid the need for exhaustively exploring all possible timing behaviors for a safe WCET bound estimation. In particular, timing compositionality is necessary to safely decompose a timing analysis into individual components and derive a global worst case based on local worst-case behavior [18]. The presence of timing anomalies can violate both timing predictability and compositionality.

A timing anomaly can either be a counterintuitive timing effect or a timing amplification. Counterintuitive timing anomalies occur whenever the locally better case leads to a globally worse case, such as a cache hit leading to increased global timing, thus inverting the expected behavior. Amplification timing anomalies occur when a local timing variation induces a larger global timing variation. While counterintuitive timing anomalies threaten the timing predictability, amplification timing anomalies affect the timing compositionality [20].

Counterintuitive timing anomalies can occur, for instance, when an execution unit is selected at run-time rather than statically [44]. In-order pipelines can also be affected by this kind of anomalies for instructions with multi-cycle latencies [3]. While vector instructions executed within Vicuna can occupy the respective functional unit for several cycles, there is only one unit for each type of instruction, and hence there is no run-time decision involved in the choice of that unit. The execution time of all vector instructions is completely deterministic, thus avoiding counterintuitive timing anomalies.

Amplification timing anomalies can be more subtle to discover, as recently shown by Hahn et al. [17], who identified the reordering of memory accesses on the memory bus as another source for timing anomalies. The presence of amplification timing anomalies is due to the non-monotonicity of the timing behavior w.r.t. the progress order of the processor pipeline [15].

We show that Vicuna is free of amplification timing anomalies by extending the formalism introduced by Hahn and Reineke [15] for their timing-predictable core SIC to our vector processing system. A program consists of a fixed sequence of instructions $\mathcal{I} = \{i_0, i_1, i_2, \dots\}$. During the program's execution, the pipeline state is a mapping of each instruction to its current progress. The progress $\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$ of an instruction is given by the pipeline stage $s \in \mathcal{S}$ in which it currently resides, as well as the number $n \in \mathbb{N}_0$ of cycles remaining in that stage. For our processing system, comprising the main core Ibex and the vector coprocessor Vicuna, we define the following set of pipeline stages:

$$\mathcal{S} = \{pre, IF, ID{+}EX, VQ, VEU, post_S, post_V\}$$

Analogous to the pipeline model used by Hahn and Reineke [15], we use the abstract stages *pre* and *post* to model instructions that have not yet entered the pipeline or have already left the pipeline, respectively. However, we distinguish between completed regular (scalar) instructions and completed vector instruction by dividing the *post* stage into $post_{\mathcal{S}}$ and $post_V$, respectively. *IF* is the main core's fetch, while *ID+EX* denotes its combined decode and execute stage. The vector coprocessor is divided into two abstract stages: *VQ* represents the vector instruction queue, and *VEU* comprises all the vector execution units. Vector instructions awaiting execution in the vector queue remain in program order, and once a vector instruction has started executing on one of the vector core's functional units, it is no longer dependent on any other instruction since there are no interdependencies between the individual vector units. Hence we do not need to explicitly model each of the concrete stages in the vector core.

Guaranteeing the strict ordering of instructions requires the following ordering $\sqsubset_{\mathcal{S}}$ of these pipeline stages:

$$\begin{tikzcd}[column sep=small] & post_{\mathcal{S}} \\ pre \sqsubset_{\mathcal{S}} IF \sqsubset_{\mathcal{S}} ID{+}EX \arrow[ur, phantom, sloped, "\sqsubset_{\mathcal{S}}"] \arrow[dr, phantom, sloped, "\sqsubset_{\mathcal{S}}"] & \\ & VQ \sqsubset_{\mathcal{S}} VEU \sqsubset_{\mathcal{S}} post_V \end{tikzcd}$$

Non-vector instructions exit the pipeline after the *ID+EX* stage, while vector instructions enter the vector queue and eventually start executing on a vector execution unit. An instruction that has fewer remaining cycles in a stage or is in a later stage than another instruction has made more progress. Hence, for two instruction with current progress $(s, n), (s', n') \in \mathcal{P}$ respectively, an order on the progress is defined as:

$$(s, n) \sqsubseteq_{\mathcal{P}} (s', n') \Leftrightarrow s \sqsubset_{\mathcal{S}} s' \vee (s = s' \wedge n \geq n')$$

The cycle behavior of a pipeline is monotonic w.r.t. the progress order $\sqsubseteq_{\mathcal{P}}$, if an instruction's execution cannot be delayed by other instructions making more progress. For this property to hold, an instruction's progress must depend on previous instructions only and never on a subsequent instruction [20]. Instructions are delayed by stalls in the pipeline. Hence any pipeline stage must only be stalled by a subsequent stage.

The vector execution units cannot stall, except for the vector load and store unit in case of a cache miss. Due to the strict ordering of memory accesses, the vector core cannot be delayed by a memory access of the main core. Hence the *VEU* stage cannot be stalled by any other stage. The vector queue holds instructions that await execution on a vector unit. Thus the *VQ* stage can only be stalled by the *VEU* stage. The *ID+EX* stage, in turn, can be stalled by an ongoing memory access of the vector core (the *VEU* stage), by a vector instruction writing back to a scalar register, when a vector instruction has been decoded, but the vector queue is full, or during memory loads and stores. Loads and stores are executed while the *IF* stage fetches the next instruction. Hence in case of an instruction cache miss on the subsequent instruction, a memory access by the *ID+EX* takes precedence over the *IF* stage. Finally, the *IF* stage can be stalled by the *ID+EX* or by a memory access of the vector core. Therefore, any pipeline stage of our processing system can only be stalled by a subsequent stage. Hence, the progress order $\sqsubseteq_{\mathcal{P}}$ of instructions is always maintained, and instructions can only be delayed by previous instructions, but not by subsequent ones. Consequently, the cycle behavior of our architecture is monotonic and hence free of timing anomalies, which in turn is a sufficient condition for timing compositionality [20].

5 Evaluation

This section evaluates our vector coprocessor's performance by measuring the execution time of parallel benchmark applications on a Xilinx 7 Series FPGA with an external SRAM with a 32-bit memory interface and five cycles of access latency. We evaluate a small, medium, and fast configuration of Vicuna with vector register lengths of 128, 512, and 2048 bits, respectively. Table 2 lists the parameters for each configuration, along with the peak multiplier performance and the maximum clock frequency.

The performance of parallel computer architectures on real-world applications is often degraded by various bottlenecks, such as the memory interface. While a large number of parallel cores or execution units might yield an impressive theoretical performance figure, efficiently utilizing these computing resources can be challenging. The roofline model [47] visualizes the performance effectively achieved by application code w.r.t. a processor's peak performance and memory bandwidth. The model shows the theoretical peak performance in operations per cycle in function of the arithmetic intensity, which is the ratio of operations per byte of memory transfer of an application. According to the roofline model, an algorithm can be either compute-bound or memory-bound [30], depending on whether the memory bandwidth or the computational performance limits the effectively achievable performance. The computational capability of a core can only be fully utilized if the algorithmic intensity of an application is larger than the core's performance per memory bandwidth.

Fig. 4 shows the roofline performance model of each of the three configurations of Vicuna, along with the effectively achieved performance for three benchmark applications, namely weighted vector addition, matrix multiplication, and the 3×3 image convolution. The dashed lines show each configuration's performance boundary, i.e., the maximum theoretical performance in function of arithmetic intensity. The horizontal part of these boundaries corresponds to the compute-bound region, where the throughput of the multipliers limits the performance. The diagonal portion of the performance boundary shows the memory-bound region, where the memory bandwidth limits the performance. Applications with a high arithmetic intensity are compute-bound, while memory-intensive applications with a low arithmetic intensity are memory-bound. Markers indicate the effectively achieved performance for each benchmark program.

The first benchmark is AXPY, a common building block of many Basic Linear Algebra Subroutine (BLAS). AXPY is defined as $Y \leftarrow \alpha X + Y$, where X and Y are two vectors, and α is a scalar. Hence, this algorithm adds the vector X weighted by α to the vector Y. We implement AXPY for vectors of 8-bit elements. For a vector of length n, it requires n 8-bit MAC operations and $3n$ bytes of memory transfer, which gives the algorithm an arithmetic intensity of 1/3, thus placing it in the memory-bound region for all three configurations.

Table 2 Configurations of Vicuna for evaluation on a Xilinx 7 Series FPGA. Note that for larger configurations, the maximum clock frequency decreases slightly as these require more resources which complicates the routing process.

Config. Name	Configuration Parameters			8-bit MACs per cycle	Clock frequency (MHz)
	Vector Reg. Width (bit)	Multiplier Data-Path Width (bit)	Data-Cache Size (kB)		
Small	128	32	8	4	100
Medium	512	128	64	16	90
Fast	2048	1024	128	128	80

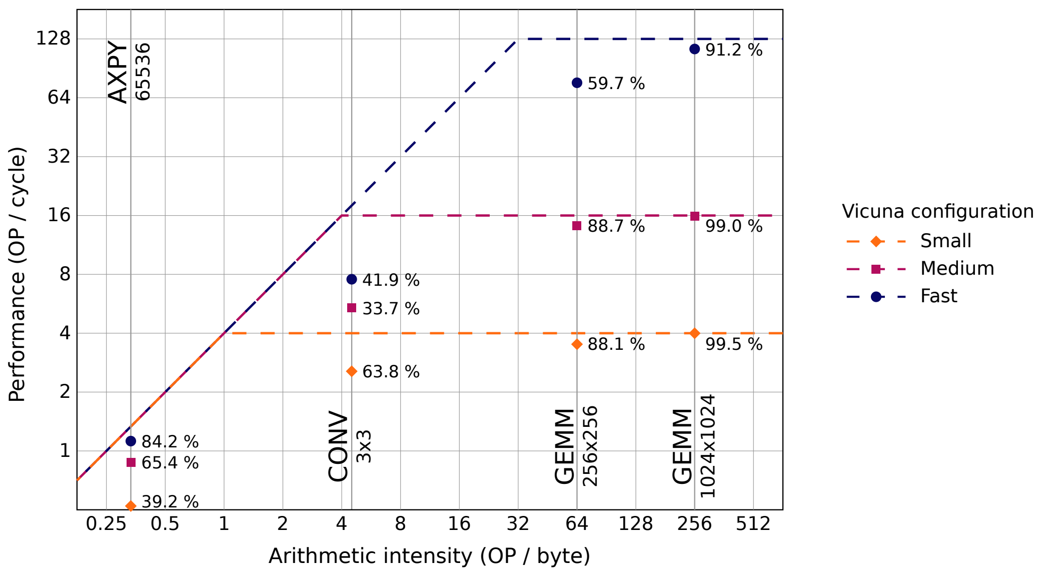


Figure 4 Roofline plot of the performance results for the benchmark algorithms for each of Vicuna's three configurations listed in Table 2. The dashed lines are the performance boundaries of each configuration, and the markers show the measured effective performance. The percentages indicate the ratio of effective vs. theoretical performance.

The next benchmark program that we consider is the generalized matrix multiplication (GEMM) $C \leftarrow AB + C$, which adds the product of two matrices, A and B, to a third matrix, C. The arithmetic intensity of this algorithm depends on the size $n \times n$ of the matrices. It requires loading each of the matrices A, B, and C and storing the result, which corresponds to a minimum of $4n^2$ values that must be transferred between the core and memory. The matrix multiplication itself requires n^3 MAC operations. We again use 8-bit values, which gives an arithmetic intensity of $n/4$ MACs per byte transferred. We evaluate Vicuna's performance for two matrix sizes, 256×256 and 1024×1024, with an arithmetic intensity of 64 and 256, respectively, which are heavily compute-bound.

Finally, we use the 3×3 image convolution, which is at the core of many convolutional neural networks (CNNs). This algorithm loads an input image, applies a 3×3 convolution kernel, and then stores the result back to memory. Hence, each pixel of the image must be transferred through the memory interface twice, once for loading and once for storing. A total of 9 MACs are applied per pixel. Thus the arithmetic intensity is 4.5.

The benchmark programs have been executed on all three configurations of Vicuna, and the execution times were measured with performance counters. Table 3 lists the recorded execution times. For all measurements, both data and instruction caches were initially cleared. The results show that the performance of Vicuna scales almost linearly w.r.t. the maximum throughput of its functional units, which is consistent with the capabilities observed in high-performance vector processors. For highly compute-bound applications, such as the matrix multiplication of size 1024×1024, the multipliers are utilized over 90 % of the time for the fast configuration and over 99 % of the time for the smaller variants.

The resource usage of Vicuna is similar to that of other FPGA-based vector processors. Fig. 5 shows a radar chart that compares the fast configuration of Vicuna to the VESPA [48] and the VEGAS [6] architectures (we compare configurations that have the same theoretical peak performance of 128 8-bit operations). Other FPGA-based vector architectures, such

Table 3 Execution time measurements of the benchmark applications for each configuration.

Benchmark	Execution time in CPU cycles on the respective configuration		
	Small	Medium	Fast
AXPY	108 985	58 693	41 989
CONV	214 486	92 852	61 719
GEMM 256 × 256	4 758 824	1 164 797	665 596
GEMM 1024 × 1024	268 277 942	67 467 224	9 182 492

as VIPERS or VENICE, have only demonstrated smaller configurations and thus are not included in this comparison. While the amount of logic resources consumed by Vicuna is similar to that of the other soft vector processors, its minimum clock period is larger. This is primarily due to the latency of the vector register file's read and write ports. VESPA can only execute one operation at a time and does not support a fused multiply-add instruction, thus requiring much fewer register file ports than Vicuna. VEGAS replaces the vector register file with a scratchpad memory with only two read and write ports. Despite its lower clock frequency, Vicuna achieves a higher effective performance than VESPA and VEGAS because of its ability to execute several operations in parallel, which allows it to better utilize its computational resources. For VEGAS, Chou et al. report an execution time of 4.377 billion cycles for a 4096 × 4096 matrix multiplication on a 32-lane configuration, which corresponds to a multiplier utilization of only 49 %. Vicuna achieves an efficiency of over 90 % for compute-bound workloads.

The efficiency of Vicuna is more in line with recent ASIC-based vector architectures, such as Cavalcante et al.'s Ara [5] and Lee et al.'s Hwacha [25]. Both of these architectures achieve over 90 % utilization of computational units, with Ara reaching close to 98 % for a 256 × 256

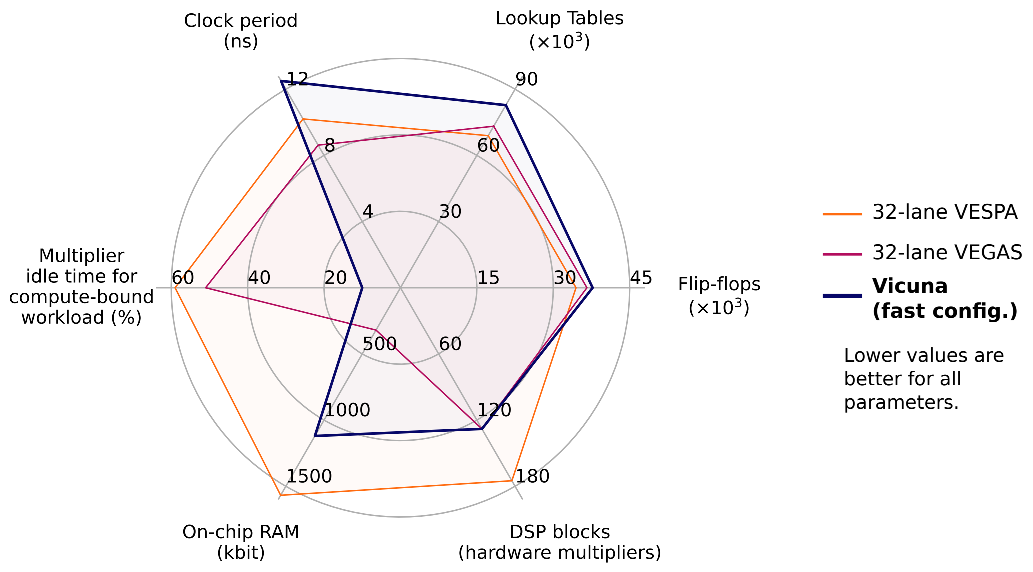


Figure 5 Resource utilization and performance of the FPGA-based vector processors Vicuna, VESPA, and VEGAS (each configured for a peak performance of 128 8-bit operations per cycle).

matrix multiplication on a configuration with 16 64-bit lanes. Yet, both Ara and Hwacha use features that are a source of timing anomalies. Ara resolves banking conflicts for its banked vector register file dynamically with a weighted round-robin arbiter that prioritizes arithmetic operations over memory operations. Therefore, run-time decisions are involved in the progress of instructions, and slow memory operations can be delayed by subsequent arithmetic instructions. Hence, Ara likely exhibits both counterintuitive and amplification timing anomalies [44]. While Hwacha sequences the accesses of vector register elements in a way that avoids banking conflicts, it uses an out-of-order write-back mechanism and consequently also suffers from timing anomalies. In addition, none of the existing vector processors that we investigated maintains the ordering of memory accesses, particularly when the main core and the vector core both access the same memory. Thus all these architectures are plagued by amplification timing anomalies [16].

A feature distinguishing Vicuna from other vector processors is its timing-predictability and compositionality. Vicuna is free of timing anomalies, enabling compositional timing analysis required for efficient WCET estimation in real-time systems. While the performance figures for Vicuna were obtained via measurements instead of a timing analysis, the predictable nature and low timing variability of Vicuna, as well as the absence of data-dependent control-flow branches in the benchmark programs, implies that their execution time is constant (assuming that the cache is initially idle). Hence, the measured execution times in Table 3 are equal to the respective WCET. Repeating the measurements with varying input data does not alter the timing and always yields the same execution times.

In contrast to timing-predictable multi-core architectures, Vicuna's performance scales significantly better. The performance of multi- and many-core systems typically does not scale linearly with the number of cores since contention on the underlying network connecting these cores to the memory interface becomes a limiting factor [28, 41]. This is particularly true in real-time systems where tasks require guarantees regarding the bandwidth and latency available to them [22, 33]. Schoeberl et al. found that the worst-case performance of the T-CREST platforms scales only logarithmically with the number of cores [38]. Similar results have been reported for the parMERASA multi-core architecture [12]. By contrast, the fast configuration of Vicuna achieves over 90 % multiplier utilization for compute-bound workloads, thus scaling almost linearly with the theoretical peak performance.

The combination of timing-predictability, efficiency, and scalability for parallel workloads makes Vicuna a prime candidate for time-critical data-parallel applications. Besides, Vicuna uses the RISC-V V extension as its instruction set, rather than custom extensions, as do most vector processors, which eases its adoption.

6 Conclusion

The performance-enhancing features in modern processor architectures impede their timing-predictability. Therefore, the performance of architectures suited for time-critical systems lags behind processors optimizing for high computational throughput. However, the increasingly demanding tasks in real-time applications require more powerful platforms to handle complex parallel workloads.

In this work, we presented Vicuna, a timing-predictable, efficient, and scalable 32-bit RISC-V vector coprocessor for massively parallel computation. We have integrated Vicuna with the Ibex processor as the main core and demonstrated that the combined processing system is free of timing anomalies, thus enabling compositional timing analysis.

The inherent efficiency of the vector processing paradigm allows us to drop common micro-architectural optimizations that complicate WCET analysis without giving rise to a significant performance loss. Despite its timing predictability, the effective performance of Vicuna scales almost linearly w.r.t. the maximum throughput of its functional units, in line with other high-performance vector processing platforms. Therefore, our vector coprocessor is better suited for time-critical data-parallel computation than the current timing-predictable multi-core architectures.

References

1 K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 230–237, December 2013. doi:10.1109/FPT.2013.6718358.

2 Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, CA, USA, 1998.

3 Mihail Asavoae, Belgacem Ben Hedia, and Mathieu Jan. Formal Executable Models for Automatic Detection of Timing Anomalies. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2018.2.

4 S. F. Beldianu and S. G. Ziavras. Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*, 64(3):805–817, 2015. doi:10.1109/TC.2013.2295820.

5 Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP:1–14, December 2019. doi:10.1109/TVLSI.2019.2950087.

6 Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950413.1950420.

7 Daniel Dabbelt, Colin Schmidt, Eric Love, Howard Mao, Sagar Karandikar, and Krste Asanovic. Vector processors for energy-efficient embedded systems. In *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, MES '16, page 10–16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2934495.2934497.

8 J. Dean. The deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 8–14, February 2020. doi:10.1109/ISSCC19947.2020.9063049.

9 G. A. Elliott and J. H. Anderson. Real-world constraints of GPUs in real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 2, pages 48–54, 2011. doi:10.1109/RTCSA.2011.46.

10 Glenn A. Elliott and James H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 2012. doi:10.1007/s11241-011-9140-y.

11 Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972. doi:10.1109/TC.1972.5009071.

12 Martin Frieb, Ralf Jahr, Haluk Ozaktas, Andreas Hugl, Hans Regler, and Theo Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *Int. J. Parallel Program.*, 44(6):1296–1336, December 2016. doi:10.1007/s10766-016-0432-7.

13 V. Golyanik, M. Nasri, and D. Stricker. Towards scheduling hard real-time image processing tasks on a single GPU. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 4382–4386, 2017. doi:10.1109/ICIP.2017.8297110.

14 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Dagstuhl, Germany, May 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.27.

15 S. Hahn and J. Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481, 2018. doi:10.1109/RTSS.2018.00060.

16 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 299–308, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2997465.2997471.

17 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. *Toward Compact Abstractions for Processor Pipelines*, pages 205–220. Springer International Publishing, 2015. doi:10.1007/978-3-319-23506-6_14.

18 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

19 R. M. Hord. *The Illiac IV: The First Supercomputer.* Springer-Verlag Berlin Heidelberg GmbH, 1982.

20 M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee. Formal semantics of predictable pipelines: a comparative study. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 103–108, 2020. doi:10.1109/ASP-DAC47756.2020.9045351.

21 Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. doi:10.1145/3140659.3080246.

22 Nassima Kadri and Mouloud Koudil. A survey on fault-tolerant application mapping techniques for network-on-chip. *Journal of Systems Architecture*, 92:39–52, 2019. doi:10.1016/j.sysarc.2018.10.001.

23 Junsung Kim, Ragunathan (Raj) Rajkumar, and Shinpei Kato. Towards adaptive gpu resource management for embedded real-time systems. *SIGBED Rev.*, 10(1):14–17, 2013. doi:10.1145/2492385.2492387.

24 Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G. Liu, and J. Gregory Steffan. Composing multi-ported memories on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3), September 2014. doi:10.1145/2629629.

25 Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202, September 2014. doi:10.1109/ESSCIRC.2014.6942056.

26 Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 137–146, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1450095.1450117.

27 Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008. doi:10.1109/MM.2008.31.

28 Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(1):3–21, January 2009. doi:10.1109/TCAD.2008.2010691.

29 Tulika Mitra. Time-predictable computing by design: Looking back, looking forward. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3316781.3323489.

30 G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, March 2014. doi:10.1109/ISPASS.2014.6844463.

31 John Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU computing. *Proceedings of the IEEE*, 96:879–899, May 2008. doi:10.1109/JPROC.2008.917757.

32 Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch. Sirakoulis, and Giorgos Dimitrakopoulos. RISC-V2: A scalable RISC-V vector processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, September 2020. doi:10.1109/ISCAS45731.2020.9181071.

33 Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Hard real-time application mapping reconfiguration for NoC-based many-core systems. *Real-Time Systems*, 55:433–469, 2019. doi:10.1007/s11241-019-09326-y.

34 RISC-V International. *Working draft of the proposed RISC-V V vector extension*, January 2021. Version 0.10. URL: https://github.com/riscv/riscv-v-spec.

35 Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978. doi:10.1145/359327.359336.

36 S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 220–229, 2015. doi:10.1109/CODESISSS.2015.7331385.

37 P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, September 2017. doi:10.1109/PATMOS.2017.8106976.

38 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

39 Aaron Severance and Guy Lemieux. VENICE: A compact vector processor for FPGA applications. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–5, 2011. doi:10.1109/HOTCHIPS.2011.7477515.

40 Aaron Severance and Guy Lemieux. Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign* *and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. doi:10.1109/CODES-ISSS.2013.6658993.

41 Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Comput. Surv.*, 50(2), 2017. doi:10.1145/3057267.

42 Sudarshan Srinivasan, Pradeep Janedula, Saurabh Dhoble, Sasikanth Avancha, Dipankar Das, Naveen Mellempudi, Bharat Daga, Martin Langhammer, Gregg Baeckler, and Bharat Kaul. High performance scalable FPGA accelerator for deep neural networks, 2019. URL: https://arxiv.org/abs/1908.11809.

43 Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernandez, Francisco Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka. Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15(3), May 2016. doi:10.1145/2910589.

44 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 295–303, 2005. doi:10.1109/QSIC.2005.49.

45 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

46 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.

47 Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. doi:10.1145/1498765.1498785.

48 Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 61–70, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1450095.1450107.

49 Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core CPU accelerator. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, page 222–232, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1344671.1344704.

50 M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–110, 2014. doi:10.1109/RTAS.2014.6925994.