A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine

DOI:
10.1109/asap.2018.8445115

Document Version
Accepted author manuscript

A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine

Jose Raul Garcia Ordaz and Dirk Koch
School of Computer Science
The University of Manchester, United Kingdom
{raul.garcia, dirk.koch}@manchester.ac.uk

***Abstract*—In this work, we present a soft dual-processor system that, as a distinctive feature, seamlessly integrates a partially run-time reconfigurable 128-bit SIMD engine. Importantly, the SIMD engine is tightly coupled to both scalar CPUs and it is shared amongst them with the purpose of drastically improving overall area utilization. We show that the proposed SIMD engine increases performance-per-area and that it can be used to substantially accelerate time consuming kernels for a set of media applications.**

I. Introduction

Embedded systems are often required to deliver high performance for a wide range of applications, to operate with relatively small energy budgets, and to be very cost efficient. In order to meet these stringent requirements, three types of architectures are commonly used to build embedded systems: (1) Multi-processor architectures, which have been proven to boost performance by exploiting program-level parallelism or task-level parallelism. (2) Vector architectures, which exploit data-level parallelism in applications to deliver better performance-per-area. (3) Adaptive architectures, which are based on reconfigurable devices and which can be tuned at run-time to enhance the performance of a running application.

Commonly, existing embedded systems only use one or two of the approaches described above. In contrast, in this article we present a novel system, targeting the embedded domain, that integrates (1) the multi-processor, (2) the SIMD, and (3) the adaptive approach to achieve high performance and increased performance-per-area efficiency.

The architecture proposed in this article consists of a soft dual-processor system augmented with a partially run-time reconfigurable (PRR) 128-bit SIMD engine. This PPR 128-bit SIMD engine can be split into 2, 4, 8, or 16 lanes according to the requirements at hand. The SIMD engine is integrated into the dual-processor system in such a way that programmability overhead is kept at a minimum as the PRR-related mechanism would be hidden to the software developer. Moreover, only a small set of special instructions would be required to invoke SIMD operations at run-time (see Section III).

Instead of implementing a costly SIMD engine per each system CPU, the proposed system implements a SIMD engine which is shared amongst both CPUs. This is achieved by putting in place a resource sharing infrastructure that allows the CPUs to time-share the reconfigurable SIMD unit. With this, the overlap of compute and I/O bursts in time can be leveraged. While standard operations such as arithmetic or boolean operations can be executed by the SIMD engine, *custom* SIMD instructions can also be integrated at run-time to accelerate time-consuming kernels for media applications (see Section V).

The contributions of this article are as follows: we reveal in detail the microarchitecture of an adaptable soft multi-processor system featuring a partially run-time reconfigurable shared 128-bit SIMD engine. Furthermore, we describe the design flow used to implement the static and the dynamic subsystems comprising our design. Moreover, we demonstrate that our system is more performance and area-efficient than a baseline soft dual-processor system.

The remainder of this article is organized as follows: in Section II we present related work, in Section III, we describe the architecture of our proposed design. In Section IV, we elaborate on the implementation details of the system, includ-

ing the PRR shared SIMD engine. In Section V we present a case study to showcase the benefits of the proposed system. Finally, in Section VI we present our conclusions.

II. Background

In order to deliver the high performance demanded by applications from compute-intensive domains such as the media domain, hardened general-purpose processors have been augmented with vector extensions. For example, ARM incorporated this by introducing the NEON engine, a 128-bit SIMD execution unit mainly targeted to accelerate media applications [1]. The NEON engine has its own register file and can move data to/from the scalar ARM CPU. The NEON engine is often found in ARM multi-processor systems such as the Cortex-A9 MPCore [2] system. A similar system is presented by Lee et al. in [3]. It is a dual-processor RISC-V system with a vector accelerator targeted for ASIC implementation called Hwacha. That vector accelerator is organized in execution banks. Hwacha breaks each vector instruction into micro-ops which are executed on its corresponding bank.

In both of the previously described systems, *an instance of the vector engine is implemented per scalar CPU* in the system. Note that implementing each individual vector engine has a substantial area cost (and an associated energy cost). Consequently, the overall multi-processor system featuring vector accelerators has a considerable area and energy overhead compared to a system comprised only of scalar CPUs.

Furthermore, as the vector instructions provided by each accelerator are hardwired, it is not possible to reconfigure NEON or Hwacha at run-time, for instance, to adapt to an application with requirements not considered at design time. In this case, more specialized functionalities, exposed to the programmer as vector instruction set extensions (ISE), would have to be added at design-time. However, while this approach can lead to enhanced performance, it would have substantial area and energy costs.

In terms of soft-processor based systems, Yu et al. presented in [4] a vector accelerator that can be efficiently implemented in FPGAs. This FPGA-specific vector accelerator is coupled to a scalar soft-processor in order to accelerate compute-intensive kernels by exploiting existing data-level parallelism. According to that work, vector accelerators are comparable in performance to HLS-generated custom accelerators. Moreover, both vector accelerators and HLS-generated custom hardware, offer a high degree of configurability and a straightforward programmability.

Similarly, Anjam et al. presented in [5] a VLIW-based dual-processor system that shares a single execution unit amongst the two CPUs. That system implements a resource controller that time-shares the VLIW execution unit. The rationale behind the described approach is that by having a shared execution unit, less resources and energy are required. However, that system [5] is completely *static* as it cannot be modified at run-time.

In contrast, our proposed architecture is comprised of a *static subsystem* that implements most of the multi-processor system, and a *dynamic subsystem* which implements the SIMD engine. In detail, we are reconfiguring the ALU datapath of a SIMD engine. In this case, the PRR SIMD unit can be reconfigured at run-time to integrate different customized SIMD instructions.

While soft multi-processor systems deliver a substantial level of performance, it would be highly desirable that they had the ability to adapt to the requirements of a certain application at run-time to achieve higher performance. To get this degree of flexibility, partial run-time reconfiguration has been used.

For example, Cazzaniaga et al. demonstrated in [6] that by using partial reconfiguration it is possible to change the number of processors available in a system to provide high performance and flexibility. Similarly, Nguyen et al. presented in [7] a MicroBlaze-based multi-processor system where one CPU remains static at run-time while the others can be partially reconfigured. Moreover, the authors discuss how to meet challenges such as task migration, bitstream relocation, and processor debug in the context of adaptable multi-processor systems.

Furthermore, Janßen et at. presented in [8] a heterogeneous multi-processor system based on the Zynq chip from the vendor Xilinx that leverages the concept of a Pynq overlay to provide *partially configurable overlays*. In that system, the hardened ARM multi-processor system is *loosely coupled* to reconfigurable overlay accelerators through a standard AXI bus. In contrast, our proposed PRR SIMD engine is very *tightly coupled* to the main pipelines of both system CPUs as described in Section III.

III. Proposed Architecture

As a starting point, we built a dual-processor system based on the Taiga soft single-processor [9]. Taiga implements a

subset of the 32-bit ISA of the RISC-V architecture [10]. In particular, Taiga implements integer (I), multiply/divide (M), and atomic (A) instructions. It was decided to leverage a RISC-V CPU because it is currently becoming a dominating open source solution providing well-maintained documentation, tool-chain, and a growing ecosystem.

Each scalar CPU features independent non-coherent instruction and data caches. Moreover, the CPUs are augmented with a 128-bit SIMD engine which is shared by both CPUs. In order to integrate the SIMD engine into the scalar CPUs, some modifications were performed to each processor: First, (1) the decode unit was extended to enable SIMD instruction encoding. Then, (2) a 16-entry 128-bit wide register file is implemented to hold vector operands/results. Finally, (3) the load/store unit was modified to enable SIMD load/store operations. This is achieved by loading/storing the four 32-bit elements that comprise the SIMD vector in a sequential fashion.

In our design, the SIMD engine is logically placed inline with the scalar execution units and the reconfigurable region hosting SIMD instructions is tiled in three slots as presented in Figure 1. As that figure shows, each slot can be seen as a blackbox that can implement SIMD instructions (note that additional slots can be allocated at design-time to enable larger instructions if needed).

We partitioned our baseline design into (1) a *static subsystem*, comprising the CPUs and the memory subsystem, and (2) a *dynamic subsystem*, comprising the PRR shared SIMD engine.

The static subsystem is allocated to a *static region* on an FPGA and it is physically placed to the right and left sides of the dynamic subsystem. Correspondingly, the dynamic subsystem is implemented on top of a *partially run-time reconfigurable (PRR) region* in the FPGA, as described in Section IV.

A. Resource Sharing Components

To enable sharing of the SIMD engine, our proposed system implements the following components: (1) a *slot-based infrastructure* which propagates input operands and output results to and from both processors. (2) A *configuration controller* for the SIMD unit which is implemented as part of the static subsystem. (3) *A programmable instruction decoder* that is used to allow a user to retarget the system according to current requirements.

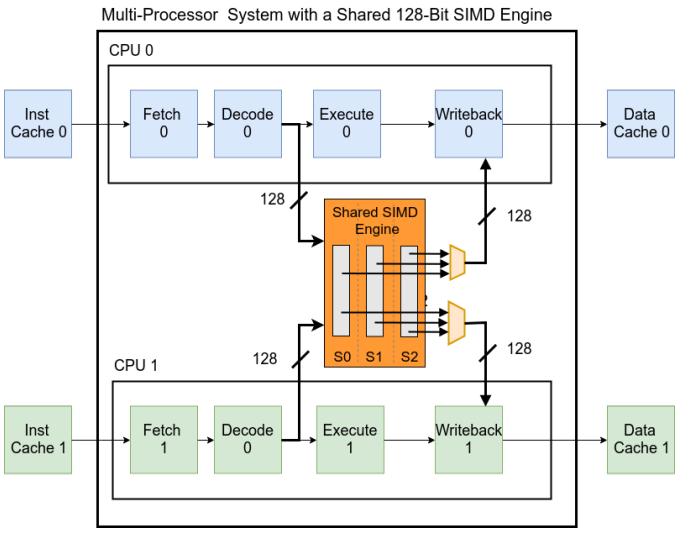

Fig. 1: A high-level block diagram of the proposed architecture. A dual-processor system provides a shared 128-bit SIMD engine that is placed logically inline with the two scalar execution units. The SIMD engine is tiled in three slots where each slot produces its own 128-bit result.

1) Slot-based Resource Sharing Infrastructure: The slots that comprise the SIMD unit are interconnected in the following fashion: operands coming from the decode unit of Core 0 enter the shared SIMD engine from its left-hand side. Then, these operands (A0, B0) are propagated through connection macros from Slot 0 to the next one until the operands reach the last slot (i.e. Slot 2). Similarly, operands coming from the decode unit of Core 1 enter the SIMD engine from the opposite side. Then, these operands (A1, B1) are propagated from the last slot (Slot 2) to the previous one until the operands reach the fist slot (i.e. Slot 0). Note that the connection macros are only used at design-time in order to constrain routing and connection points for vector custom instructions. The previously described wiring arrangement is shown in Figure 2.

With this arrangement, SIMD instructions implemented in any slot get access to the two 128-bit input operands coming from each CPU. Note that each SIMD instruction produces its own 128-bit result. Consequently, the SIMD engine will produce three different results for Core 0 (R0-0, R0-1, R0-2) and three different results for Core 1 (R1-0, R1-1, R1-2).

2) SIMD Engine Configuration Controller: The configuration controller is used (1) to time-share the SIMD engine amongst the CPUs and (2) to integrate vector instructions at run-time as required. The controller will exploit compute and I/O bursts that are commonly observed in CPUs in order to assign the SIMD engine to the processor that currently re-

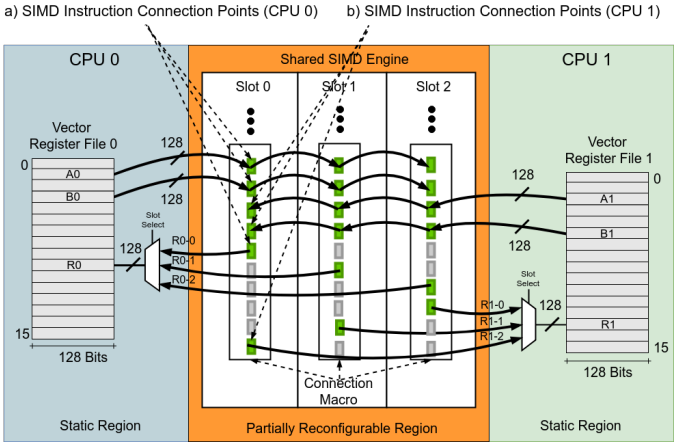

Fig. 2: High-level view of the static/dynamic interconnection in our proposed system. (a) Shows possible connection points for a reconfigurable SIMD instruction for CPU 0 (A0, B0, R0), and (b) Shows possible connection points for a reconfigurable SIMD instruction for CPU 1 (A1, B1, R1).

quires executing a vector instruction during a particular SIMD processing burst. In this way, a CPU executing a compute-intensive kernel at a certain point in time will make use of the SIMD engine while the other CPU could be performing an I/O task. In case of a conflict, one CPU is stalled. The stalling CPUs are changed after each collision in order to guarantee fairness.

We are using sharing of functional units for achieving overall better performance-per-area. Moreover, we anticipate that collisions of the two CPUs caused by simultaneous access requests to a SIMD instruction will likely be seldom as exclusive access to a particular SIMD slot is only needed for one issue cycle. In cases where sharing of resources is not desired, a second instance of the SIMD instruction may be considered. Furthermore, two separate instances of the same instruction (one for CPU 0 and one for CPU 1) may be implemented into the same slot provided that the SIMD instructions in question are lightweight enough to fit into the same slot.

Note that the controller will also handle the (re)configuration of SIMD instructions. We imagine an scenario where application-specific SIMD instructions are identified offline through program profiling. The selected application-specific SIMD instructions are synthesized as partial bitstreams to be integrated at run-time. Note that a SIMD instruction is selected only if it provides substantial speedups over scalar instructions for a given kernel. The kernel speedup S_k provided by a specific SIMD instruction is expressed as: $S_k = t_{kISA}/t_{kSIMD}$, where the execution time of the kernel using instructions from the existing RISC-V ISA (t_{kISA}) is divided by the execution time of the same kernel using that specific SIMD instruction (t_{kSIMD}). Moreover, each individual SIMD instruction has to be invoked a substantial number of times to amortize the configuration overhead.

This is expressed as: $t_{SIMD} * n > t_{RCFG}$, where t_{SIMD} is the execution time of each individual SIMD instruction, n represents the number of times that the SIMD instruction is invoked, and t_{RCFG} is the reconfiguration time corresponding to that SIMD instruction.

TABLE I: Example settings of the programmable instruction decoder.

CPU 0		CPU 1	
SIMD Inst. Encoding	Inst. Settings	SIMD Inst. Encoding	Inst. Settings
0	Trap	0	Slot 0, 2 Cycles, λ=1$\times$
1	Slot 1, 1 Cycle, λ=4$\times$	1	Trap
2	Slot 3, 2 Cycles, λ=2$\times$	3	Slot 3, 2 Cycles, λ=2$\times$
3	Trap	4	Trap

3) Programmable Instruction Decoder: A programmable instruction decoder (PID) is used to provide the ability to retarget the system at run-time without having to perform a costly reconfiguration of the static subsystem. The PID is an extension to the existing decode unit. The PID decodes SIMD instruction information including: (1) the slot (i.e. S0-S2) from which the result has to be retrieved, (2) the number of execution cycles corresponding to that particular vector instruction, and (3) the operation folding factor [11] (λ = 1$\times$, 2$\times$, 4$\times$), which indicates that the vector operation will be fractioned into N sub-operations of a smaller size and executed sequentially for allowing cheaper implementation cost and enhanced flexibility. Note that these settings can be overwritten at run-time, for instance, to retarget vector instructions to a different slot result or alternatively, to modify vector instruction latency.

For example, consider the PID settings presented in Table I. SIMD Instruction 0 is currently used only by CPU 1 and it is executed in two CPU cycles. Note that this SIMD instruction has a folding factor of 1, which indicates that it gets executed on a 128-bit execution unit. Similarly, SIMD Instruction 1 is used by CPU 0 only and has a folding factor of 4, which indicates that the 128-bit operation is split into 4 32-bit sub-operations where each one takes 1 CPU cycle to be issued to the execution stage (i.e. 4 CPU cycles are required in total to execute this SIMD instruction). In contrast, SIMD Instruction

2 is shared by the two processors and it has a folding factor of $\lambda = 2\times$.

Note that a *trap mechanism* is triggered by the decode unit in case an issued SIMD instruction is not present in any of the available engine slots. Here we decided for a software-based approach where a trap handler performs either software emulation or reconfiguration.

IV. System Implementation

Our proposed system is implemented on Digilent's Zedboard, which features a Zynq device (Z-7020) from the vendor Xilinx. Implementation details of both, the static (scalar) subsystem and the partially run-time reconfigurable (SIMD) subsystem are described next.

A. Static (Scalar) Subsystem Implementation

The static subsystem is comprised of two scalar RISC-V CPUs and a common supporting infrastructure which is used to interconnect the CPUs with the DDR memory. Note that we are not using the ARM CPUs for processing in our design, however the ARM subsystem is used as a link to the DDR memory as direct communication between the FPGA fabric and DDR memory in this device is not available [12]). The operating frequency of the resulting static subsystem is 96 MHz and the corresponding resource utilization is presented in Table II.

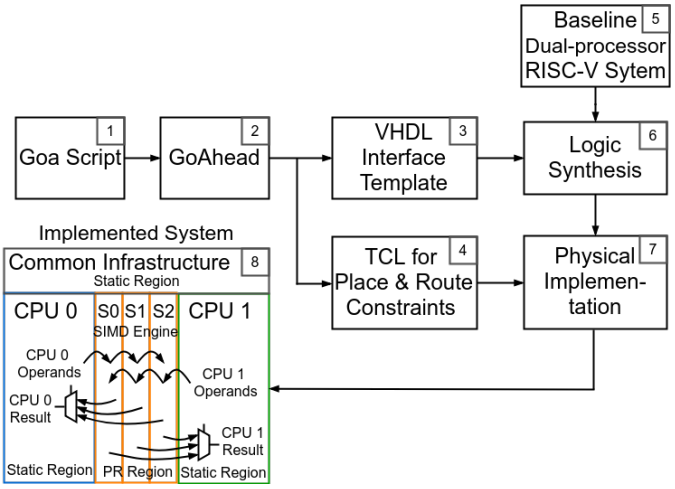

Fig. 3: Partial reconfiguration design flow for the proposed system.

While the Vivado tool from the vendor Xilinx provides a flow for partial reconfiguration (PR) [13], it is not well suited for implementing our design as it has some important limitations: (1) in the Vivado PR flow the routing to the anchor points from the PR region to the static region *generally*

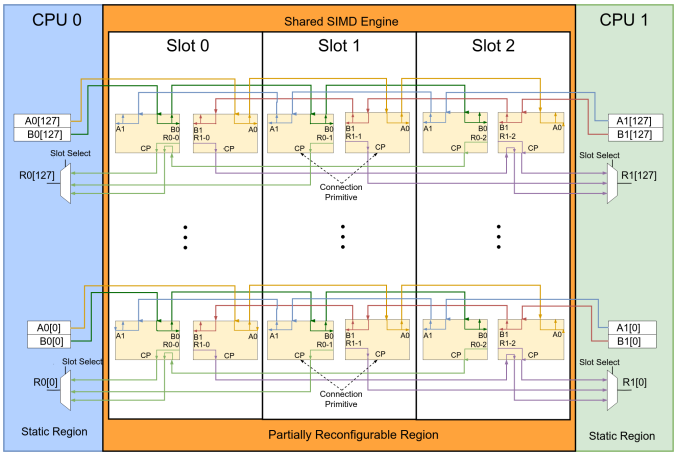

Fig. 4: A detailed diagram of the wiring implemented to support resource sharing for the SIMD engine. Please note that each signal in the interface is heavily constrained to use a specific wiring resource in the FPGA fabric.

TABLE II: Static subsystem resource utilization.

System Component	FPGA Primitive		
	LUT	DSP	BRAM
Soft Dual-Processor RISC-V System	6917	8	36
RISC-V CPU 0	2728	4	10
RISC-V CPU 1	2728	4	10
Common Infrastructure	1461	0	16

changes when the static subsystem is resynthesized. Moreover, (2) a PR region *cannot be shared* by multiple modules at the same time. Finally, (3) modules *cannot be easily relocated* to a different slot position.

It is worth mentioning that our system is quite challenging to route as just the operands and results take over $2\times(2+3)\times128 = 1280$ individual interfacing wires that cross between the static subsystem region and the reconfigurable region which is located in a relatively small area with strict requirements on the interfacing wires.

To overcome those issues, we used the GoAhead tool [14] to build the static and the dynamic subsystems. GoAhead leverages the concept of hard connection macros and blocker macros to produce physical implementation constraints that produce a predefined structured routing for interfacing with reconfigurable modules.

GoAhead is used to generate TCL scripts that are used by the Vivado flow to implement an interface that is *vertically aligned, has low communication overhead, and remains consistent even if the static system is resynthesized.*

The GoAhead tool also generates a VHDL template that contains the corresponding module interface. This VHDL entity is integrated into the system source code at design time

and it is synthesized along the rest of the static subsystem.

The static/dynamic interface is physically implemented through connection primitives using the wire resources available in the FPGA fabric as illustrated in Figure 4. The connection primitives can be seen as placeholders for reconfigurable SIMD instruction modules. Figure 3 illustrates our PR design flow.

Figure 5 shows the floorplan of the implemented dual-processor system. As this figure shows, the system is divided into a static region and a PR region. Note that CPU 0 is placed at the left-hand side of the PR region. Similarly, CPU 1 is placed at the right-hand side of the PR region.

The static/dynamic interface is implemented in such a way that the operands and results are connected botton-up (when considering LSB to MSB direction). This incorporates the fact that arithmetic operators are physically implemented botom-up (i.e. the carry chain runs botom-up on the used Xilinx Zynq FPGA). Consequently, four operand/result bit vectors are connected per CLB row of the FPGA. Note that the reconfigurable slots do not span the full height of a clock region (which is not necessary in our implementation), as required by the Xilinx PR design flow [13].

The smallest atomic unit that can be written to the FPGA is a *frame* which provides configuration information of all primitives and routing resources within the height of a clock region. Sub-column reconfiguration is possible by generating a configuration bitstream that keeps the information of the static subsystem and only updates the configuration data of the individual slot to be configured.

Corresponding bitstreams can be generated at the bitstream level using the tool BitMan [15] at run-time. This method uses the property that by overwriting a configuration multiple times with the same content, this will not cause any glitches. Alternatively, bitstreams can be generated offline using netlist operations in GoAhead.

B. PRR (SIMD) Subsystem Implementation

The SIMD engine is implemented in the PRR region located amongst the two system CPUs (see Figure 5). The size of the PR region is set to 2082 LUTs. As described in Section III, the SIMD engine implements a slot-based architecture where each slot is 2 CLB columns wide. Therefore, each slot is comprised of 694 LUTs.

The PRR region is reconfigured through the Internal Configuration Access Port (ICAP) supported by the Zynq device [13].

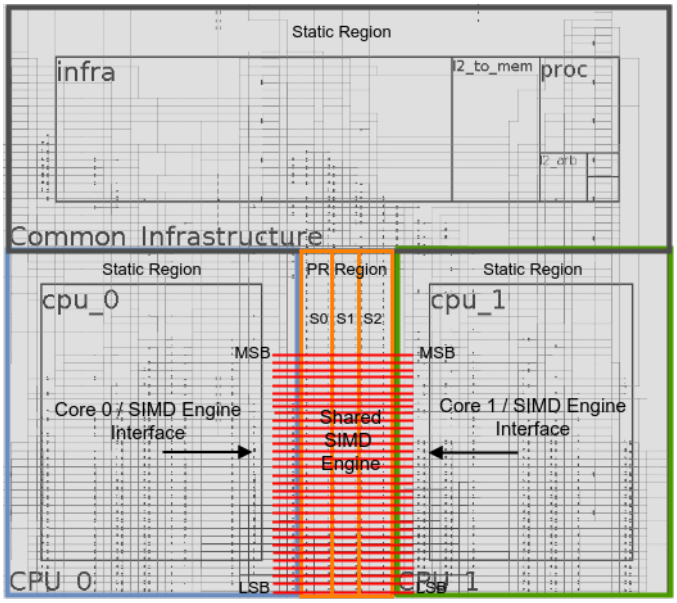

Fig. 5: Dual-Processor RISC-V system floorplan featuring a reconfigurable shared 128-Bit SIMD engine. The static/dynamic interfacing wires are highlighted in red.

ICAP provides a maximum throughput of 400 MB/s. Consequently, the theoretical best reconfiguration time for each PR execution unit slot is 295 μs.

However, to hide SIMD module reconfiguration overhead (which can negatively impact overall system performance), we consider integrating SIMD module configuration prefetching similarly as described by Hauck in [16]. With this technique, prefetching calls can be inserted into the existing application code so that SIMD module configurations are loaded in advance to efficiently hide configuration latency.

V. Case Study

In the following paragraphs we describe a case study, divided in two parts, that showcases the benefits of our proposed system.

1) Part 1: In this part, we consider the case where each system CPU executes its own program, using its corresponding memory space. As an example, we examine three application kernels where vector instructions substantially reduce program execution time as described in Section III: (1) a kernel for mixing (MIX) N 16-bit PCM audio signals for a signal processing program, (2) a kernel for computing the Sum of Absolute Differences (SAD) for a motion estimation program, and (3) a kernel for computing 8-bit Add-Compare-Select (ACS) values for a Viterbi decoder algorithm. We obtained speedup and resource utilization numbers for each kernel.

Speedup gains are obtained by using the RISC-V toolchain and a cycle accurate simulator [17]. First, we compile each program to obtain binaries. Then, those binaries are disassembled to examine the generated assembly code. Additionally, the binaries are simulated to extract program execution information. Finally, based on the gathered information, speedup gains are derived for using SIMD instructions instead of the existing scalar RISC-V instructions.

Resource utilization is obtained from the Vivado tool after implementing the selected SIMD instructions in RTL. These SIMD instructions have been constrained to the area allocated to a single 2 CLB-wide slot (i.e. 694 LUTs) as described in Section IV.

Implementation results are summarized in Table III. The studied kernel benchmarks are presented in the first column. The second column shows the application-specific SIMD instructions implemented for each kernel (note that we use the prefix *"V"* to indicate a vector instruction and a postfix *"8/16"* to indicate a byte or a halfword operation).

Column 3 shows SIMD instruction latency. Column 4 shows the resource footprint corresponding to each SIMD instruction in terms of LUTs. Finally, Column 5 shows the kernel speedup gained by using the implemented SIMD instructions over existing scalar operations.

Let us consider the *VADD16* instruction, which was generated to accelerate the MIX kernel. It implements relatively lightweight addition operations, and consequently, 2 instances of this instruction can be simultaneously implemented in one individual slot at a combined cost of 513 LUTs. This also applies to the *VADD8* and *VSUB8* instructions implemented for the ACS kernel. With this, each CPU can execute the same instruction in a non mutually-exclusive manner.

In contrast, note that the *VSAD8* instruction that is implemented for the SAD kernel uses 1452 LUTs. In this case, to implement the *VSAD8* instruction, there could be two possible solutions: (1) one would be to use other available slots. Alternatively, (2) the design technique known as *operation folding* [11] can be applied.

With this technique, commonly used in the DSP domain, a vector operation normally executed by a *vector execution unit* in *1 CPU cycle*, is split into N sub-operations which are sequentially executed by a *scalar execution unit* in *N CPU cycles*. In this case, the *VSAD8* instruction can be split into 4 sub-operations and sequentially executed by a scalar *SAD8* instruction which has a cost of 364 LUTs.

Note that the reported cost is for a single instance of the scalar *SAD8* instruction. Consequently, the system CPUs could only use this instruction in a mutually-exclusive fashion (when using a single slot). Note that while a latency penalty is paid with this approach, it allows the remaining slots to be used to implement other instructions.

TABLE III: Characteristics of the implemented SIMD instructions.

Kernel Benchmark	SIMD Instruction	Instruction Latency	Resource Footprint	Kernel Speedup
MIX	VADD16	5.7 ns	513 LUTs	8×
ACS	VADD8	5.7 ns	513 LUTs	16×
	VSUB8	5.7 ns	513 LUTs	
	VSEL8	4.5 ns	129 LUTs	
SAD	VSAD8	25.6 ns	1452 LUTs	4×
SORT	VSORT8	5.3 ns	312 LUTs	452 ×
	VMBOX	6.1 ns	128 LUTs	

2) Part 2: In this part, we consider the case where an application is partitioned into the two system CPUs. The first half of the application's data is assigned to the private data memory space of CPU 0, and the second half is assigned to the private data memory space of CPU 1. As an example, consider a sorting kernel (SORT) where an array of pixels (8-bit values) has to be sorted by hue. In this case, half of the array data is loaded to the private data memory space of CPU 0, and the other half of the array is loaded to the private data memory space of CPU 1.

Each processor sorts its corresponding array subset and then both results are merged, similarly as described by Zurek et al. in [18], and by Wakasugi et al. in [19]. Instead of using existing scalar RISC-V instructions, a vector sorting instruction is integrated into the SIMD engine so that each processor can sort 16 8-bit values at a time. According to our analysis, sorting 16 8-bit values in one CPU running at 100 MHz, using scalar instructions, takes 32 μs.

In contrast, by using our custom *VSORT8* instruction, the same 16 8-bit values can be sorted in just one CPU cycle. This represents a theoretical kernel speedup of 452×. The area cost of the *VSORT8* instruction is low, for this reason, two instances of the same instruction had been integrated in the same slot at a cost of only 312 LUTs.

In this way, each CPU can execute the same sorting instruction in a non mutually-exclusive fashion. Table III summarizes these numbers. Moreover, since each kernel is executed in parallel by the two system CPUs, it is expected that the overall sorting application performance is additionally boosted about 2×.

Additionally, note that the architecture of our proposed system opens up the opportunity of putting in place a low-overhead mechanism for *core-to-core communication*. We implemented this mechanism as a special mailbox instruction *VMBOX*. This instruction can be used to move 128-bits of data from one processor to the other in one CPU cycle without putting pressure to the memory subsystem. This is useful, for example, on the case described above where the sorted data array from one of the CPUs is moved to the other processor using this instruction so that both sorted data arrays can be finally merged. The characteristics of the *VMOV* instruction are presented in Table III.

VI. Conclusions

In this paper we presented a RISC-V multi-processor system that provides a partially run-time reconfigurable 128-bit SIMD engine. Our design merges the multi-processor paradigm, and the SIMD and partial run-time reconfiguration techniques to build a system that delivers more performance, efficiency, and flexibility than a comparable baseline static system.

Additionally, we described the architecture of the proposed system which includes the design of the static and the dynamic subsystems and their interfaces. In particular, for the dynamic subsystem, we described a slot-based execution unit used to dynamically integrate SIMD instructions at run-time to obtain substantial performance boosts. Furthermore, we presented in detail the PR design flow and the additional components used to implement our system.

Finally, we presented a case study to showcase our proposed design. We demonstrated that by exploiting the PRR shared SIMD engine, custom SIMD instructions can deliver speedups of 4-452$\times$ for 4 time-consuming kernels. All this was achieved at low resource footprints of just 129-1452 LUTs.

As future work, we envision extending the existing SIMD instruction set with additional custom vector instructions to accelerate a wider range of applications. For example custom vector instructions to process arbitrary-precision operands can be designed to accelerate machine learning applications that show high data-level parallelism.

Furthermore, future work could also include scaling the existing dual-processor system. In this case, a number of design aspects can be considered. For example, (1) allocating more resources to the dynamic subsystem and expanding the 128-bit interface to a wider vector, (2) adding more CPUs to the existing system, (3) increasing both the static/dynamic interface as well as the number of CPUs in the system.

Optionally, we are considering extending the existing RISC-V toolchain with auto-vectorization options as a way to facilitate the development of applications that can exploit the provided SIMD engine.

Acknowledgment

This work is kindly supported by the Mexican National Council for Science and Technology (CONACyT) under grant 381920, and through the Defence Science and Technology Laboratory (DSTL), UK, under grant DSTLX-10000092266. Furthermore, we thank the Xilinx University Program for providing us with tools and hardware.

References

[1] "Introducing NEON. Development Article," www.arm.com.
[2] "Cortex-A9 MPCore. Technical Reference Manual," www.arm.com.
[3] Y. Lee *et al.*, "A 45nm 1.3 GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators," in *European Solid State Circuits Conference*. IEEE, 2014.
[4] J. Yu, G. Lemieux, and C. Eagleston, "Vector Processing as a Soft-Core CPU Accelerator," in *FPGA*. ACM, 2008.
[5] F. Anjam, S. Wong, and F. Nadeem, "A Shared Reconfigurable VLIW Multiprocessor System," in *IPDPSW*, 2010.
[6] A. Cazzaniga *et al.*, "On the Development of a Runtime Reconfigurable Multicore System-On-Chip," in *DSD*, 2012.
[7] T. D. Nguyen and A. Kumar, "PR-HMPSoC: A Versatile Partially Reconfigurable Heterogeneous Multiprocessor System-on-Chip for Dynamic FPGA-Based Embedded Systems," in *FPL*, 2014, pp. 1–6.
[8] B. Janßen, P. Zimprich, and M. Hübner, "A Dynamic Partial Reconfigurable Overlay Concept for PYNQ," in *FPL*, 2017.
[9] E. Matthews and L. Shannon, "TAIGA: A New RISC-V Soft-Processor Framework Enabling High Performance CPU Architectural Features," in *FPL*, 2017.
[10] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume I: User-Level ISA," www.riscv.org.
[11] K. K. Parhi *et al.*, "Synthesis of Control Circuits in Folded Pipelined DSP Architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, 1992.
[12] M. Santarini, "Zynq-7000 EPP Sets Stage for New Era of Innovations," *Xcell Journal*, vol. 75, 2011.
[13] "Vivado Design Suite User Guide. Partial Reconfiguration," www.xilinx.com.
[14] C. Beckhoff *et al.*, "GoAhead: A Partial Reconfiguration Framework," in *FCCM*, 2012.
[15] K. D. Pham *et al.*, "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*. IEEE, 2017.
[16] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors," in *FPGA*. ACM, 1998.
[17] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, 2011.
[18] D. Żurek *et al.*, "The Comparison of Parallel Sorting Algorithms Implemented on Different Hardware Platforms," *Computer Science*, 2013.
[19] Y. Wakasugi *et al.*, "MipsCoreDuo: A Multifunction Dual-Core Processor," in *ISPACS*, 2009.