PR-ESP: An Open-Source Platform for Design and Programming of Partially Reconfigurable SoCs

Biruk Seyoum, Davide Giri, Kuan-Lin Chiu, Bryce Natter and Luca Carloni
Department of Computer Science Columbia University New York, U.S.A
{biruk, davide_giri, chiu, luca}@cs.columbia.edu, bdn2113@columbia.edu

***Abstract*—Despite its presence for more than two decades and its proven benefits in expanding the space of system design, dynamic partial reconfiguration (DPR) is rarely integrated into frameworks and platforms that are used to design complex reconfigurable system-on-chip (SoC) architectures. This is due to the complexity of the DPR FPGA flow as well as the lack of architectural and software runtime support to enable and fully harness DPR. Moreover, as DPR designs involve additional design steps and constraints, they often have a higher FPGA compilation (RTL-to-bitstream) runtime compared to equivalent monolithic designs.**

In this work, we present PR-ESP, an open-source platform for a system-level design flow of partially reconfigurable FPGA-based SoC architectures targeting embedded applications that are deployed on resource-constrained FPGAs. Our approach is realized by combining SoC design methodologies and tools from the open-source ESP platform with a fully-automated DPR flow that features a novel size-driven technique for parallel FPGA compilation. We also developed a software runtime reconfiguration manager on top of Linux. Finally, we evaluated our proposed platform using the WAMI-App benchmark application on Xilinx VC707.

I. Introduction

Heterogeneous system-on-chip (SoC) architectures, which combine general-purpose processors with multiple domain-specific hardware accelerators, have become a dominant trend for implementing complex systems across a wide range of application domains. As the integration of diverse computing elements into a single chip has become very challenging over the years, several tools had been proposed both to reduce the complexity and raise the abstraction of the system-level design [1]–[3]. However, the majority of the proposed tools still target FPGAs only for rapid prototyping or functional verification purposes. Moreover, the *dynamic partial reconfiguration* (DPR) capability, which enables to modify only a portion of the circuit on the fly without requiring a full reconfiguration of the FPGA, is rarely exploited by these tools.

The challenge of designing heterogeneous partially reconfigurable SoCs is multifaceted. On one hand, the complexity that originates from integrating several independently-designed components requires a DPR-compliant flexible architecture and a companion methodology. On the other hand, integrating the partial reconfiguration flow into a SoC flow can create an implementation quagmire. The difficulty of the integration is even more exacerbated by the challenging DPR FPGA flow. To date, this flow is only semi-automated by the vendor tools, thus requiring an expert-level familiarity in low-level FPGA architecture to efficiently implement complex designs. For example, the allocation of partially reconfigurable accelerators to reconfigurable regions and the subsequent floorplanning for these regions still needs to be done manually. Furthermore, due to the additional design constraints and steps involved in DPR designs, the full compilation of a DPR design takes a much longer CPU runtime when compared to equivalent monolithic designs on commercial CAD tools. Finally, a DPR system requires the implementation of a software framework the provides an abstraction for a low-latency reconfiguration as well as runtime management of different DPR-related services.

While several approaches have been proposed to address one, or a combination of, these challenges [4]–[9], a holistic solution is still lacking. To fill this gap we present PR-ESP, an open-source DPR-based system-level design platform to build partially reconfigurable heterogeneous SoCs. To realize our platform, we adopted the heterogeneous tile-based distributed architecture of the Open-Source ESP platform [1], [10] as a baseline and introduced several changes to the architecture of its tiles to enable DPR support. The ESP platform was especially appealing to us because it simplifies the development and integration of loosely-coupled partially reconfigurable accelerators into complex SoC architectures. We also built a tool that, in addition to fully automating the DPR flow on Xilinx FPGAs, opportunistically parallelizes the FPGA physical implementation stage (place and route, *P&R*) to reduce the total FPGA compilation runtime. Finally, we developed a software stack containing a runtime manager with a lightweight Application Programming Interface (API) for ESP accelerators and driver modules for the hardware reconfiguration controller to support both Linux and baremetal applications.

The following are our key technical contributions:

- We conceived and realized a robust yet flexible automated system-level design flow for DPR. Our flow, which extends the current ESP FPGA design flow for monolithic (non DPR) designs, enables the generation of full and partial bitstreams for a complete SoC using a single `make` target.
- For the hardware support of our flow, we designed two new types of tiles that augment the native ESP tile-based architecture by providing several DPR-compliant features in a modular way, based on its socket-based approach. We designed a *reconfigurable tile* that accommodates a partially reconfigurable subset of an SoC design. We also modified the native ESP *auxiliary tile* by adding new features to enable and control the partial reconfiguration.
- We designed and integrated a size-driven P&R parallelism algorithm within our flow to reduce the total FPGA compilation time. We developed the algorithm by performing an extensive characterization of the Vivado tool with several designs and then built an approximate model that correlates the size of the design with the *P&R* runtime.
- At the software level, we supported our flow by augmenting the ESP software stack with a DPR runtime manager

978-3-9819263-7-8/DATE23/© 2023 EDAA
Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.

and a driver for Linux and baremetal applications.

II. Essential Background

This section summarizes the DPR capabilities of Xilinx FPGAs and the main capabilities of the ESP project to help the reader appreciate our contributions, which are described in the following sections.

Dynamic partial reconfiguration (DPR), which has recently been re-branded as dynamic function exchange (DFX) by Xilinx [11], includes a *static part*, which is the subset of the design that is not subject to runtime reconfiguration, and one or more *reconfigurable partitions* (RPs), which host the partially reconfigurable modules. In the scope of this paper, these modules are typically loosely-coupled hardware accelerators [12].

The main design steps in the Xilinx DPR flow include *logic partitioning*, *floorplanning*, *synthesis*, and *implementation* (*P&R* and bitstream generation). In the partitioning step, partially reconfigurable accelerators are pre-allocated to specific RPs of the design. The floorplanning step involves the generation of physical placements for the RPs on the FPGA fabric. The placements for the RPs (`pblocks` in Xilinx terminology) must satisfy all the resource constraints of the hosted modules as well as the technological constraints imposed by the vendor. DPR floorplanning is not yet fully automated by Xilinx tools and must be performed manually.

ESP is an open-source research platform for heterogeneous SoC design and programming [1]. ESP combines a scalable architecture and a flexible design methodology [10]. The ESP architecture is structured as a tile grid. The tiles form a distributed system which is inherently scalable, modular and heterogeneous. The main types of tile are: processor, accelerator, and memory. For the processor tile, ESP currently allows a seamless choice between the 32-bit Leon3 SPARC core [13] and the 64-bit CVA6 (Ariane) RISC-V core [14]. An accelerator tile contains one or more loosely-coupled accelerators; these can be accelerators developed with the ESP methodology as well as third-party open-source accelerators like the NVDLA [15]. Each tile is encapsulated into a modular socket that interfaces it to a network-on-chip (NoC), which has a packet-switched 2D-mesh topology with multiple physical planes. ESP also provides a shared local memory (SLM) tile, which hosts an on-chip memory of configurable size. The ESP methodology guides the choice of the number, mix, and placement of tiles for a target SoC as well as the design of new SoC components, particularly accelerators.

III. Architectural Support For The PR-ESP Platform

In this section we describe how we augmented the ESP architecture to support DPR. These changes include the addition of a new type of reconfigurable tile and upgrades in the auxiliary tile to support runtime reconfiguration.

Reconfigurable Tile Architecture. The native ESP accelerator tile contains several architectural features that make it non-compliant with the rules and constraints of a DPR design. For example, the dynamic power management logic of the SoC, which contains clock modifying components, resides deep within the accelerator tile hierarchy. This feature is not

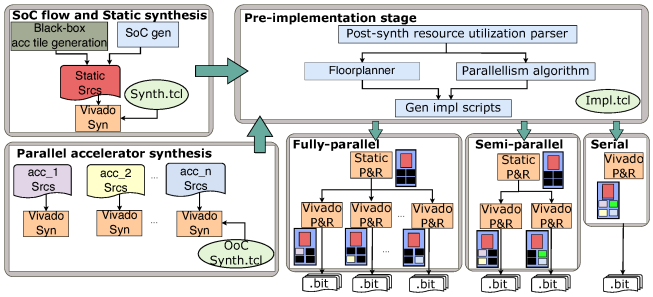

Fig. 1. Block diagram of the PR-ESP FPGA flow.

compliant with the Xilinx DPR rule that prohibits placing clock modifying logic inside reconfigurable regions. Furthermore, the accelerator tiles drive an output clock interface that feeds the main SoC clock, but this is prohibited by Xilinx's DPR guideline that does not allow such route-through paths inside partially reconfigurable partitions [16].

To solve these problems and enable DPR within the tiles, we designed a new type of tile named *reconfigurable tile*, which hosts the runtime reconfigurable accelerators. This tile, whose architecture is shown in Fig. 2B, contains a reconfigurable wrapper (the brown box) with a predefined common interface for all reconfigurable accelerators supported by the SoC that includes *(i)* load/store ports for memory access, *(ii)* configuration ports for the memory-mapped registers, and *(iii)* an interrupt signal to notify task completion. While compliant with the ESP socket-based approach, the reconfigurable tile contains decoupling logic to detach the interfaces of the accelerator wrapper from the socket during runtime reconfiguration. The reconfiguration decoupling logic (the red box) is controlled by software via the memory-mapped configuration registers of the tile. During reconfiguration, the decoupler also disables the inputs to the NoC queues (the orange lines), which are located between the tile port of the NoC routers and the different proxy components. After a successful reconfiguration, the decoupler resets and re-enables the queues.

Reconfiguration Controller. To enable runtime partial reconfiguration, we augmented the *auxiliary tile* of ESP with a logic that instantiates the dynamic function exchange controller (DFXC) IP block from Xilinx [11] and the ICAP primitive (ICAPE2 or ICAPE3 depending on the FPGA family). The DFXC is configured at runtime using its memory-mapped registers via its AXI-Lite interface, which is seamlessly interfaced to the APB bus of the auxiliary tile using an AXI-Lite to APB adapter. The DFXC has an AXI4 master interface to fetch bitstreams from memory. An adapter inside the auxiliary tile translates the AXI transactions into NoC packets. At the end of a successful reconfiguration, the DFXC component sends an interrupt to the processor core in order to disable the decoupler in the reconfigurable tile and to start the operations of the new accelerator.

IV. The PR-ESP FPGA Flow

Although DPR was primarily intended to build adaptive systems, it has also been shown to be beneficial for reducing the time to generate the bitstream of an FPGA design [7]. Beyond the full automation of the SoC implementation, our DPR flow is designed to exploit this additional advantage.

Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.

Fig. 1 illustrates our DPR flow, where the RTL-to-bistream FPGA compilation of reconfigurable SoCs is fully automated for Xilinx FPGAs. The flow starts by parsing the input SoC configuration [1] to generate the RTL hierarchy of the full SoC. The parsing enables the separation of the sources for all the reconfigurable tiles from the static part. In this work, the static part is composed of all the instances of memory (MEM), processor (CPU), auxiliary (AUX), and shared local memory (SLM) tiles in the SoC. During synthesis, the reconfigurable accelerators inside the static part are replaced with instances of auto-generated black-box wrappers. Following the parsing step, the flow performs a parallel synthesis of the static part and all of the reconfigurable tiles. These syntheses are performed by running separate instances of the Xilinx Vivado tool guided by auto-generated synthesis scripts. To fully parallelize the synthesis, our flow exploits the out-of-context (OoC) synthesis mode [11] that is offered by Vivado.

The synthesis stage is followed by the pre-implementation stage, where the flow performs floorplanning (generates the placement *pblocks* for the reconfigurable tiles) and also decides the optimal level of parallelism that best reduces the total compilation runtime at the *P&R* stage of the design. Then, according to the type of parallelism, the flow orchestrates the synthesized checkpoints to fully automate the *P&R* to generate bitstreams.

DPR Floorplanning. We automated the DPR floorplanning targeting our Xilinx evaluation boards (VC707, VCU118, and VCU128) by adapting FLORA, an open-source DPR floorplanning tool [17].

Choosing Optimal *P&R* Parallelism. For both monolithic and DPR designs, *P&R* is the most time consuming design step. Overall, we would intuitively expect that the total design compilation time of most DPR designs can be reduced if the *P&R* of reconfigurable tiles is parallelized; i.e. the *P&R* of reconfigurable tiles is performed in parallel by using separate Vivado instances. However, deciding the type of parallelism that minimizes the cost of compilation time becomes a new challenge that requires a further understanding of the behavior of the CAD tool (in this case Vivado) in relation to the relative size and configuration of the static and reconfigurable parts of different DPR designs. To address this issue, we performed an exhaustive characterization of the Vivado tool. We built an empirical model that correlates the size of a DPR design against the total compilation time for *P&R* under different parallelism configurations. Then, we used the model to develop an algorithm that, depending on the post-synthesis resource utilization of the design, chooses between three types of *P&R* parallelism strategies: *(i)* serial, *(ii)* fully-parallel, and *(iii)* semi-parallel implementations. Next, we describe the three implementation strategies along with our model, followed by the characterization of Vivado. Finally, we provide a description of the algorithm to choose the appropriate strategy.

Given an SoC containing N reconfigurable tiles with $\tau \in \{1, N\}$ denoting the number of parallel *P&R* runs, we define three implementation strategies:

- *Serial*: this is the case when $\tau = 1$ and the implementation is performed without any parallelism using a single Vivado instance.

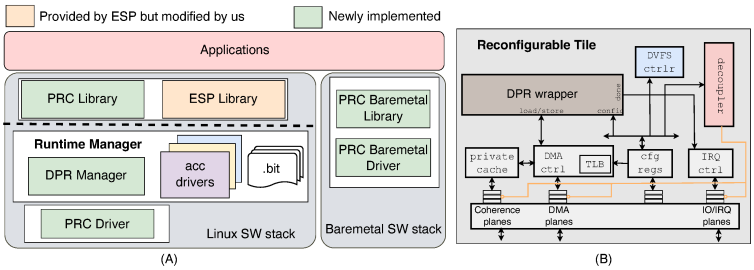

Fig. 2. (A) Our proposed software stack (B) The modified reconfigurable tile.

- *Fully parallel*: this is the case when $\tau = N$ and all the reconfigurable tiles are implemented in parallel by running separate instances of Vivado. This strategy contains an additional intermediate step. Before implementing the reconfigurable tiles, the synthesized static netlist is first placed and routed by instantiating place-holder hard-macros of empty reconfigurable tiles inside the black-boxes within the netlist. Since these empty netlists are prepared offline, they add no additional timing overhead. Then, the flow creates N instances of Vivado, each performing *P&R* on a single reconfigurable tile in-context with the pre-routed static part. In this case, the total compilation time can be expressed as, $T_{full} \approx t_{static} + max\{\Omega_i\}, \forall i = 1 \dots N$, where t_{static} and Ω_i denote the time to pre-route the static part and the execution time of the *P&R* for the i^{th} reconfigurable tile respectively.
- *Semi-parallel*: this strategy is for DPR designs where a fully-parallel implementation is an overkill and the serial implementation is too slow. In this case, two or more reconfigurable accelerators are opportunistically grouped and implemented together in each Vivado instance. Similar to the fully parallel one, this strategy also includes an intermediate static-only *P&R* step.

Vivado Characterization. As our characterization was aimed at exploring the effect of the size-driven parallelism on the total compilation time, we investigated the 3-way correlation between the size (measured in number of LUTs) of the static part (κ), the average size of accelerators (α_{av}), and the ratio of sum total of all the reconfigurable accelerators to the static part (γ). For an SoC with N reconfigurable tiles, these values can be defined as follows:

$$\kappa = \frac{lut_{static}}{LUT_{tot}}, \quad \alpha_{av} = \frac{\sum_{i=1}^{N} lut_i}{N \cdot LUT_{tot}}, \quad \gamma = \frac{\sum_{i=1}^{N} lut_i}{lut_{static}} \tag{1}$$

where lut_i and lut_{static} denote the number of LUTs in the i^{th} reconfigurable tile and the static part of the SoC, respectively. Next, we define the five possible classes DPR designs can be grouped into depending on their size (resource utilization):

Group 1: $\kappa \gg \alpha_{av}$, designs that belong in this category have a static part which is much larger than each of the individual reconfigurable tiles. Depending on the ratio of the total size of all the reconfigurable tiles to the static part, i.e., the value of γ, designs in this group can be divided into three classes described below:

Class 1.1: when $\gamma < 1$ the size of the static part is also larger than the size of all reconfigurable tiles combined.

Class1.2: when $\gamma > 1$ the total size of the reconfigurable tiles exceeds the size of the static part.

Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.

Class 1.3: when $\gamma \approx 1$ the size of the static part is approximately equal to the sum of all the reconfigurable tiles.

Group 2: $\kappa \ll \alpha_{av}$ or $\kappa \approx \alpha_{av}$, this category represents designs where the size of the static part is either equal to or much less than each of the reconfigurable tiles. Also in this group, there are two additional classes depending on the size of γ.
Class 2.1: when $\gamma > 1$, then a design contains one or multiple reconfigurable tiles whose size are larger than the static region.
Class 2.2: the case $\gamma \approx 1$ is true under the condition where the design contains only a single reconfigurable tile.

Note that, for designs that belong in the second group, $\gamma < 1$ denotes an impossible condition, meaning if the size of a static region is smaller than the average reconfigurable part, then it is impossible for the ratio of the total reconfigurable area to the static area to be smaller than one.

To perform the characterization, we designed 4 SoCs, one for each of the first four classes described above, targeting Xilinx VC707 FPGA. Since designs that belong to Class 2.2 can only be implemented in a serial mode, there is no need to investigate further. The first design, `SoC_1`, which was designed to fit in Class 1.1, has a 4x5 tile configuration containing 16 instances of a reconfigurable MAC accelerator that was generated by using the ESP *Vivado HLS* accelerator flow. The second design, `SoC_2`, which is of Class 1.2, has a 3x3 tile configuration and contains four reconfigurable accelerators: *(i)* 2-d convolutional (Conv2d), *(ii)* matrix multiply (GEMM), *(*iii) Fast Fourier Transform (FFT), and *(iv)* vector sorting (sort). These accelerators are designed in SystemC and synthesized by using the *Cadence Stratus HLS* tool. The third SoC, `SoC_3`, that belongs to Class 1.3, is a variant of `SoC_2` containing only the Conv-2d, GEMM, and sort accelerators. The static part of all three SoCs is composed of a single instance of the MEM, AUX, and a CPU tile with an instance of a Leon3 core. The last SoC, `SoC_4`, which belongs to Class 2.1, is created by modifying the CPU tile in `SoC_2` to move it from the static part into the reconfigurable part. In this case, our goal was not making the CPU partially reconfigurable but reducing the size of the static part.

We performed the characterization by using several implementation runs on all four SoCs under different levels of parallelism and recorded the total design compilation time. The characterization took hundreds of hours and was performed using Vivado 2019.2 on an Intel Core-i7 machine with 16 cores running at 3.6GHz and 64GB DRAM memory. We run each test case serially to maximize the accuracy of the characterization. Although we run multiple instances of Vivado on the machine, due to the inherent sequential nature of the *P&R* algorithms, Vivado actually uses a limited number of the cores [18].

Table II reports the resource consumption of the accelerators, the CPU tile, and the static part with and without the processor. Table III provides a summary of the characterization for the four SoCs under different levels of implementation parallelism. The boldface values on the table denote the implementation with the shortest compilation time. Based on these results and a long experience working on DPR designs using the Vivado tool, we devised an algorithm that chooses an implementation strategy that improves the total design runtime. The algorithm, which is based on the strategies defined in Table I, mainly relies on the resource profile of the design to make a choice. For example, for designs where the static part is much larger than both the average reconfigurable accelerator and the sum of all reconfigurable accelerators (Class 1.1), a serial implementation is favorable. But under the previous condition, if the static part is less than the sum of all reconfigurable accelerators (Class 1.2), then a fully-parallel or semi-parallel implementation is likely to reduce the timing cost. The two entries of Table I that are left unfilled correspond to impossible conditions discussed above.

V. Software Support For the PR-ESP Platform

ESP already provides a library API and auto-generated Linux and baremetal device drivers to invoke accelerators. However, the software stack lacks the necessary abstraction to enable a runtime swapping of accelerators and their respective drivers. We augmented the software stack by implementing *(i)* a Linux kernel level runtime manager that handles the scheduling and synchronization of reconfiguration requests as well as the swapping of accelerator drivers during reconfiguration, *(ii)* Linux and bare-metal drivers to handle the decoupling of tiles and FPGA reconfiguration via the PRC and ICAP modules, and *(iii)* a user-space API to expose DPR services to applications. Fig 2A shows the modified software stack.

Before the start of application execution, partial bitstreams, which are *mmaped* in the user-space in the DDR, are copied into the kernel memory. This enables the runtime manager to create a reference between the bitstreams, their physical addresses, the tiles they will be loaded into, and their respective drivers. The runtime manager uses the built-in kernel *workqueue* to manage multiple reconfiguration requests. Reconfiguration requests are queued up and executed as soon as the PRC is ready. However, before being inserted into the queue, the manager forces the calling thread to wait for the accelerator in the tile to complete its execution. During reconfiguration, it locks access to the device so that other threads trying to access it must wait until the reconfiguration is complete (interrupt is received from the PRC) and the new driver is loaded. The loading of drivers is realized by modifying the ESP library that registers and un-registers drivers.

VI. Experimental Evaluation

We present the case study of an embedded SoC application that was designed and implemented using our approach. The SoC is composed of a set of image-processing accelerators for the open-source Wide Area Motion Imagery (WAMI) benchmark suite [19]. Fig. 3 depicts the data flow of the accelerators in the SoC, which include *Debayer*, *Grayscale*, *Lucas-Kanade*, and *Change-Detection* kernels. We decomposed the *Lucas-Kanade* accelerator into multiple accelerators to further parallelize its execution. We first profiled each accelerator for its LUT consumption and execution time by using a 2x2 SoC with a single accelerator tile and targeting a Xilinx VC707 board. The values obtained from the profiling are annotated next to each accelerator in Fig 3. We then used the benchmark application to evaluate compilation runtime of the DPR flow of PR-ESP as well as the performance (execution time and energy

Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THE SIZE-DRIVEN IMPLEMENTATION STRATEGIES IN PR-ESP.

	$\gamma < 1$	$\gamma \approx 1$	$\gamma > 1$
$\kappa \approx \alpha_{av}$	-	serial	fully-parallel
$\kappa \gg \alpha_{av}$	serial	semi-parallel	semi/fully-parallel
$\kappa \ll \alpha_{av}$	-	serial	fully-parallel

TABLE II
THE RESOURCE UTILIZATION OF THE ACCELERATORS.

	MAC	Conv-2d	GEMM	FFT	Sort	CPU	Static	Static (w/o CPU)
LUTs	2450	36741	30617	33690	20468	41544	82267	39254

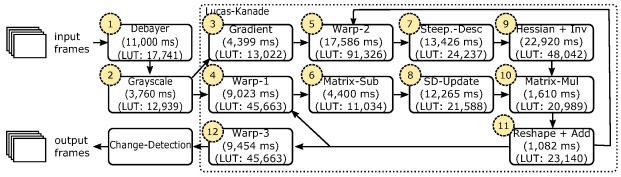

Fig. 3. Data flow model of the WAMI-App.

efficiency) of the partially reconfigurable SoCs implemented using the PR-ESP platform.

Evaluation of the PR-ESP DPR Flow. For this experiment, we created four SoCs, each with a 3x3 tile configuration hosting a combination of the accelerators from the WAMI application. The composition of the accelerators is aimed at creating systems whose LUT consumption profile coincides with the different classes described in Section IV. The combination of accelerators inside the SoCs is provided in Table IV (second column). Also in this case the CPU tile of SoC_D is configured as partially reconfigurable to reduce the size of the static area.

We first performed *P&R* on all four SoCs in different parallelism configurations to evaluate if the parallelism strategies chosen by our algorithm indeed produced the best results. For all the semi-parallel implementations we set $\tau = 2$. The boldface columns on Table IV denote the strategy chosen by PR-ESP for that particular SoC. As shown on the table, for each class of design, the parallelism strategy chosen by PR-ESP resulted in the fastest *P&R* runtime. An interesting point to note from these results is that, while it is true that parallelization reduces the compilation time for most designs as suggested in [7], our results demonstrate that, designs that belong in Class 1.1 benefit from a serial implementation rather than a parallel one. This point is confirmed again in the comparisons against monolithic implementations.

We also compared the design runtime of the full implementation (synthesis and *P&R*) of the SoCs in PR-ESP against their equivalent implementations in Xilinx's standard DPR flow, which is always performed in a single instance of Vivado. Table V reports the results of the comparison. The condition in which our full flow performs the best are for designs that belong to Classes 1.2 and 2.1. Indeed, in our comparison, PR-ESP improved the total implementation time of SoC_A by 46 minutes (19%) and SoC_D by 54 minutes (24%). This improvement is mainly due to the larger size of these classes of designs (large static part that is exceeded by an even larger total reconfigurable tiles), which makes the standard DPR implementation very difficult and time consuming. For such designs, the PR-ESP flow improves the compilation time by opting for a fully-parallel implementation. We also evaluated SoC_C, which represents Class 1.3. In this case, our flow adopts a semi-parallel implementation strategy using two Vivado instances and implements the design with 2 semi-parallel runs. Also, in this case our design is slightly better than the monolithic flow (by 4.4%). Finally, for designs where serial implementation is the best strategy (Class 1.1), our flow does as good as, or slightly worse than, the standard implementation. In fact, for SoC_B, which was implemented in this mode, our flow was slower by only 5 minutes (2.5%). But when SoC_B was implemented using either fully-parallel and semi-parallel modes, the performances were 6.5% and 12.3% worse than the monolithic implementation, respectively. Indeed, as mentioned above, such a fine-grained classification of DPR designs with the assignment of class-specific parallelism is key aspect that distinguishes PR-ESP from [7].

TABLE III
RESULTS FROM THE CHARACTERIZATION OF VIVADO UNDER DIFFERENT LEVELS OF PARALLELISM. TIME MEASURED IN MINUTES.

	α_{av}	κ	γ	design runtime	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$	$\tau = 16$
SoC_1	0.8	27	0.48	t_{static}	89	75	75	75	75	75
				$\lceil\Omega\rceil$	-	35	30	22	19	18
				T_{tot}	**89**	110	105	97	94	93
SoC_2	10.1	27.2	1.47	t_{static}	181	94	94	94	-	-
				$\lceil\Omega\rceil$	-	79	72	58	-	-
				T_{tot}	181	173	166	**152**	-	-
SoC_3	9.6	27.1	1.07	t_{static}	158	86	86	86	-	-
				$\lceil\Omega\rceil$	-	48	52	51	-	-
				T_{tot}	158	**134**	137	-	-	-
SoC_4	10.8	11.5	4.1	t_{static}	163	42	42	42	42	-
				$\lceil\Omega\rceil$	-	88	63	58	52	-
				T_{tot}	163	130	105	100	**94**	-

Evaluation of embedded SoCs. In the second part of the experimental evaluation, we implemented the WAMI application in three different SoCs, named SoC_X, SoC_Y, and SoC_Z, with two, three, and four reconfigurable tiles in each SoCs respectively. The static part of the SoCs was composed of a single CPU, MEM, and AUX tiles. Since the WAMI workload can be mapped to the SoC in multiple ways, depending on the number of reconfigurable tiles, we manually partitioned the accelerators to reconfigurable tiles in a way that most likely maximizes the performance. Table VI lists the allocation of accelerators to reconfigurable tiles for each SoC. The table also provides the respective sizes of the partial bitstreams (pbs) generated for each accelerator. PR-ESP is configured to generate partial bitstreams using Vivado's compression mode to reduce the memory access latency during reconfiguration. We also developed a multi-threaded Linux software, with one thread per reconfigurable tile, to control the execution flow of accelerators. All SoCs process individual frames without pipelining and are implemented targeting a Xilinx VC707 FPGA connected to a 1GB shared DRAM memory at 78MHz.

Fig 4 provides the results of the comparison of the total execution time per frame and the energy efficiency, measured in Joule/Frame. As shown in Fig 4, SoC_X has the best energy efficiency compared to the other two (1.65x w.r.t. SoC_Y and 2.77x w.r.t. SoC_Z) but it does relatively worse in terms of total execution time (2.6x and 3.6x worse w.r.t. SoC_Y and SoC_Z). It also has a higher non-interleaved reconfiguration due to the fewer number of reconfigurable tiles. On the contrary, SoC_Z processes the input image in the shortest time but with the worst energy efficiency out of the three. All in all, SoC_Y has a good balance between energy efficiency, total execution time, and a minimal reconfiguration overhead.

Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
SUMMARY OF THE EVALUATION OF THE *P&R* PARALLELLISM IN PR-ESP. TIME IS MEASURED IN MINUTES

	SoC accs (indexes from Fig 3)	α_{av}	κ	γ	P&R runtime	fully-par	semi-par	serial
SoC_A	{4, 8, 10, 9} (class 1.2)	9.2	29.1	1.26	t_{static}	**98**	98	-
					$\lceil\Omega\rceil$	**52**	88	-
					$T_{P\&R}$	**150**	186	192
SoC_B	{2, 3, 11, 1} (class 1.1)	4.5	28.3	0.6	t_{static}	95	95	-
					$\lceil\Omega\rceil$	48	61	-
					$T_{P\&R}$	143	156	**135**
SoC_C	{7, 11, 8, 2} (class 1.3)	5.5	28.2	0.97	t_{static}	88	**88**	-
					$\lceil\Omega\rceil$	71	**64**	-
					$T_{P\&R}$	159	**152**	167
SoC_D	{4, 5, 9, 2} (class 2.1)	23.5	12.2	2.4	t_{static}	**48**	48	-
					$\lceil\Omega\rceil$	**71**	83	-
					$T_{P\&R}$	**119**	131	142

TABLE VI
THE PARTITIONING OF ACCELERATORS INSIDE THE THREE SOCS.

	SoC_X		SoC_Y		SoC_Z	
Reconf. Tile	WAMI accs	pbs (KB)	WAMI accs	pbs (KB)	WAMI accs	pbs (KB)
RT_1	{1, 4, 9, 10, 8}	328	{1, 3, 7, 12}	283	{1, 6, 12}	305
RT_2	{2, 3, 6, 7, 11}	245	{2, 6, 8}	247	{2, 5, 11}	359
RT_3	-	-	{4, 9, 10}	378	{4, 10, 7}	317
RT_4	-	-	-	-	{3, 8, 9}	397

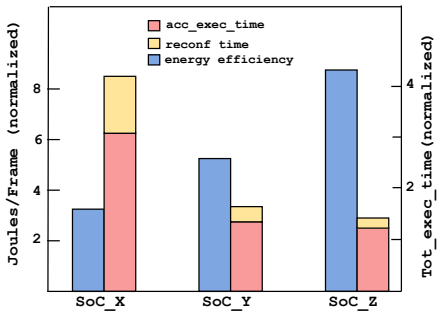

Fig. 4. Total execution time and energy efficiency of the WAMI SoC implementations.

VII. RELATED WORKS

The efficient implementation of partially reconfigurable SoCs continues to be challenging despite the growing research on several aspects of DPR designs. This include efforts to automate the DPR FPGA flow [4]–[6], as well as software frameworks to abstract several DPR related services [8], [9]. Recently some works [7], [18] had also been proposed to enable parallel compilations of DPR designs to reduce the daunting FPGA design runtime. But most of the proposed approaches focus on one or a few combinations of the challenges. Furthermore, most of the proposed works rarely take advantage of porting DPR into well matured SoC integration tools [1], [2], that enable to integrate several independently-designed components. To address the DPR-related design challenges in a comprehensive way, we propose PR-ESP, an open-source platform to design partially reconfigurable SoCs. Our platform adopts the ESP platform but augments the architecture, methodology, and introduces a novel size-driven parallel FPGA compilation technique to reduce design runtime.

VIII. CONCLUSION

We presented an open-source platform for a system-level design flow of partially reconfigurable SoC architectures targeting FPGA implementations. Our approach combines ESP, an agile open-source SoC design platform, with our custom DPR automation tool for Xilinx FPGAs. We augmented both the architecture and the methodology of ESP with capabilities that allow it to support dynamic partial reconfiguration in a modular and scalable manner. Furthermore, we introduced a robust yet flexible FPGA flow that fully automates design implementation.

TABLE V
COMPARISON OF COMPILATION TIME OF THE PR-ESP IMPLEMENTATION AGAINST MONOLITHIC IMPLEMENTATIONS. TIME IS MEASURED IN MINUTES.

	PR-ESP					monolithic		
	Synth	t_{static}	$max\{\Omega\}$	T_{tot}	τ	Synth	P&R	T_{tot}
SoC_A	47	98	52	197	4 fully-par	91	152	243
SoC_B	54	135	-	189	1 serial	60	124	184
SoC_C	42	88	64	194	2 semi-par	74	129	203
SoC_D	49	48	71	168	6 fully-par	81	141	222

Acknowledgments. This work was supported in part by DARPA (C#: FA8650-18-2-7862) and in part by the NSF (A#: 1764000). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory and DARPA or the U.S. Government.

REFERENCES

[1] P. Mantovani *et al.*, "Agile SoC development with Open ESP," in *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, 2020.
[2] A. Amid *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.
[3] C. Heinz *et al.*, "The TaPaSCo open-source toolflow," *Journal of Signal Processing Systems*, vol. 93, no. 5, pp. 545–563, 2021.
[4] B. Seyoum *et al.*, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in FPGA SoC," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 481–490.
[5] K. Vipin *et al.*, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," in *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1–8.
[6] C. Beckhoff *et al.*, "Go ahead: A partial reconfiguration framework," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 37–44.
[7] Y. Xiao *et al.*, "Reducing FPGA compile time with separate compilation for FPGA building blocks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 153–161.
[8] M. Pagani *et al.*, "A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017, pp. 96–101.
[9] A. Bucknall *et al.*, "Build automation and runtime abstraction for partial reconfiguration on xilinx zynq ultrascale+," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 215–220.
[10] L. P. Carloni, "The case for embedded scalable platforms," in *Proc. of the Design Automation Conf. (DAC)*, 2016, pp. 1–6.
[11] Xilinx, "Vivado Design Suite User Guide: Dynamic Function eXchange," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf, 2020.
[12] E. G. Cota *et al.*, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. of the Design Automation Conf. (DAC)*, Jun. 2015, pp. 202:1–202:6.
[13] Cobham Gaisler, "Leon3 processor," www.gaisler.com/index.php/products/processors/leon3.
[14] F. Zaruba *et al.*, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology," *IEEE Trans. on VLSI Systems*, 2019.
[15] "NVDLA Deep Learning Accelerator," https://github.com/nvdla/.
[16] Xilinx, "Vivado Design Suite User Guide: Partial Reconfiguration," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf, 2018.
[17] B. Seyoum *et al.*, "FLORA: floorplan optimizer for reconfigurable areas in FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358202
[18] L. Guo *et al.*, "Rapidstream: Parallel physical implementation of fpga hls designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 1–12.
[19] K. Barker *et al.*, "PERFECT (power efficiency revolution for embedded computing technologies) benchmark suite manual," *Pacific Northwest National Laboratory and Georgia Tech Research Institute*, 2013.

Authorized licensed use limited to: Columbia University Libraries. Downloaded on September 03,2023 at 13:42:52 UTC from IEEE Xplore. Restrictions apply.