Dynamically reconfigurable processors

Student: Seyed Saman Mohseni Sangtabi

Instructor: Prof. Hamid Sarbazi-Azad

July, 2021

Table of contents

Abstract

The ever-increasing demand for high-performance and real-time applications, especially on devices with power consumption limitations that also lack computational resources due to chip size limitations, leads us to a new class of architectures that tries to provide hardware performance while having software flexibility; dynamically reconfigurable architectures can be one of these architectures.

This paper will review some of the fundamentals of dynamically reconfigurable computing architectures and their programming models and compilation process. Finally, we will examine some of the real-world applications of these architectures and discuss their performance gains.

1. Introduction

Various new computationally intensive applications such as real-time video and image processing, data compression and encryption, deep learning, and AI applications with time constraints are emerging. On the other hand, we face power consumption constraints and chip area limitations. Such requirements and restrictions surpass the performance that general-purpose processors can provide, forcing the use of ad-hoc hardware-accelerated solutions, typically in the form of ASICs.

ASICs are very expensive to develop and lack flexibility; Still, they might not satisfy area constraints if there are many applications in a system requiring custom ASICs for efficient execution. Implementing numerous hardware accelerators might take up much space and be very expensive.

Dynamically reconfigurable architectures can solve the mentioned problems. In addition, they can have high energy efficiency and flexibility, making it possible to specify many application needs and thus achieve high performance while serving various applications, eliminating the need for numerous application-specific accelerated units.

We can classify reconfigurable architectures into two main classes, fine-grained (usually statically) reconfigurable and coarse-grained (usually dynamically) reconfigurable architectures.

Fine-grained reconfigurable architectures, such as FPGAs, usually allow the implementation of any logical functions; however, the fine-grained structure and the bit-level programmability of such devices result in large area and power overheads and slower performance compared to ASICs (20–35 times larger area, 7–14 times more power-consuming, and 3–4 times slower performance) [5]. Another issue with these devices is their more complex and challenging programming languages (HDL) than software-oriented ones.

Fine-grained reconfigurable architectures are typically not dynamically reconfigurable. If they are, there are restricted limitations. They usually are just partially dynamically reconfigurable; This is due to the extensive configuration data and, as a result, the slow configuration process of these architectures.

Coarse-grained reconfigurable architectures, on the other hand, have coarser computational blocks and simpler interconnections, resulting in reduced area and power overheads when compared to fine-grain reconfigurable architectures. The reduced configuration data in these devices makes dynamic runtime reconfigurability possible.

We will review both of these architectures as well as hybrid ones, which combine various reconfigurability levels, and discuss their advantages and drawbacks.

2. Fine-grained reconfigurable architectures

CPLDs and FPGAs are examples of fine-grained statically reconfigurable architectures. We will overview FPGAs' overall architecture and programming flow as examples and discuss some of its properties, use cases, and drawbacks.

FPGAs are flexible, reconfigurable devices that can be programmed to mimic almost any logical circuit. FPGAs are more affordable solutions when used in low to medium volume productions and provide faster time-to-market than ASICs; however, FPGAs are significantly larger, slower, and more power-hungry compared to ASICs, which is primarily because of the programmable routing interconnections of FPGAs, which comprises of almost 90% of the total area of FPGAs [7].

2.1. Architecture structure

Typically, FPGAs comprise configurable logic blocks (CLBs), I/O blocks, and programmable routing interconnections, connecting CLBs and I/O blocks. CLBs are configured to implement desired logical functions, and I/O blocks are responsible for off-chip connections. The overall structure of a primary FPGA is shown in fig. 1.

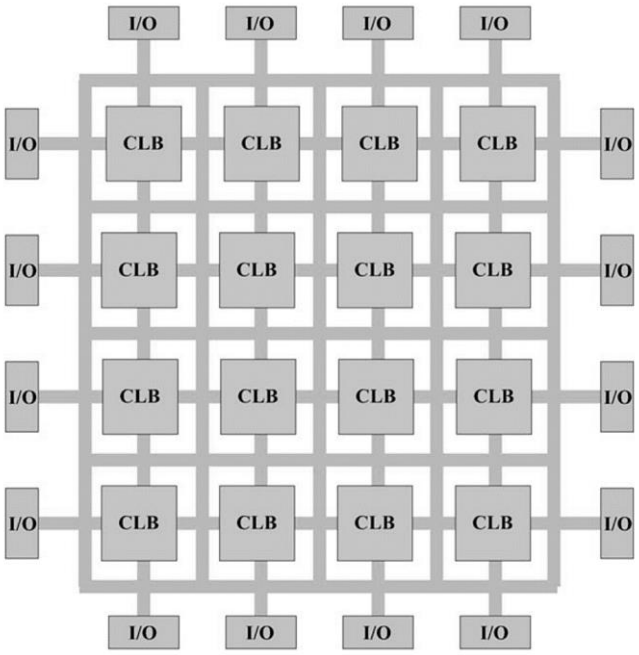

Fig. 1 The overall structure of a primary FPGA [7]

A CLB usually contains a cluster of basic logic elements (BLEs) connected through a local routing network, as shown in fig. 2. A simple BLE consists of a LUT and a Flip-Flop, as shown in fig. 3. Generally, a LUT with k inputs contains 2^k configuration bits (leftmost array of SRAM cells shown in fig. 3) which can be configured to implement any k-input

boolean function. The output of a LUT can also be selected to go through a Flip-Flop or not (by programming the rightmost SRAM cell).

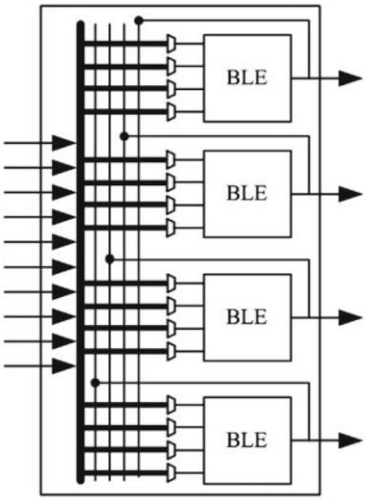

Fig. 2 A CLB having 4 BLEs [7]

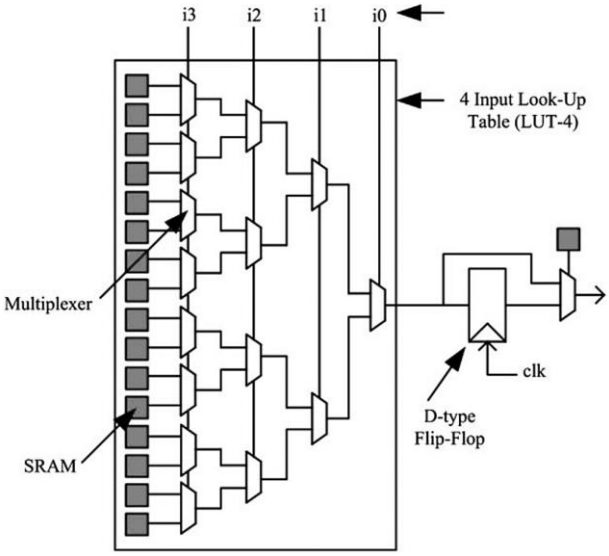

Fig. 3 A BLE having a 4-input LUT [7]

Logic blocks are connected to the programmable routing network through connection boxes (CB), as shown in fig. 4(b) and different parts of the routing network are interconnected using programmable connection boxes (CB), as shown in fig. 4(a).

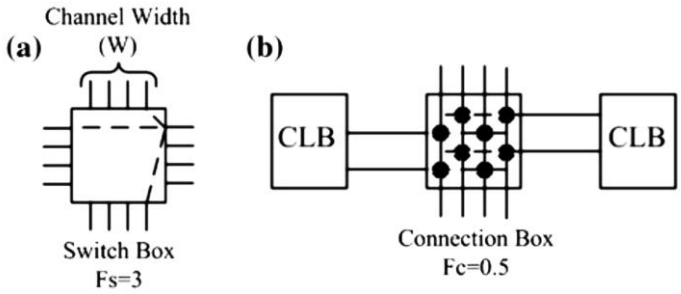

Fig. 4 SB and CB structure [7]

Besides the basic blocks mentioned, there are more complex and specific purpose blocks, called hard blocks, that are very efficient at implementing specific functions. Memory blocks, multipliers, adders, and various DSP blocks are typical examples of these blocks; such blocks can narrow the power consumption and area efficiency gap between FPGAs and ASICs.

2.2. Programming flow

FPGA applications are typically designed using a Hardware Description Language (HDL); The resulting design description is eventually converted to a bitstream used to program logic and routing blocks to form the desired logic function. This process can be divided

into logic synthesis, technology mapping, mapping, placement, and routing steps. We will briefly describe what each step (in its most basic form) does.

Logic synthesis transforms an HDL description into a hierarchical network of boolean gates and Flip-Flops, in the form of a Directed Acyclic Graph (DAG), where each node in the graph represents a gate, a flip-flop, or a primary input or output and each edge in the graph represents a connection between two circuit elements. Various technology-independent techniques are applied to optimize the boolean network.

Technology mapping is responsible for finding a network of cells (Given a cell library) that implements the Boolean network. In the FPGA technology mapping problem, the cell library is composed of k-input LUTs and flip-flops.

Mapping is the process of clustering and packing the logic blocks (k-input LUT and flip-flop pairs) hierarchically.

Placement is the process of determining which logic block within an FPGA is best to implement the corresponding logic block of the circuit based on optimization objectives. For example, it typically tries to place highly connected logic blocks closer together to minimize the required wiring or sometimes tries to balance the wiring density across the FPGA or to minimize the critical path delay. Graph-Partitioning-based algorithms and simulated annealing approaches are among the most common methods in placement algorithms.

Routing involves assigning nets to the routing resources so that no more than one net uses each routing resource. Abstracting the FPGA resources as a directed graph $G(V, E)$ where V represents the set of terminals of logic blocks, and E represents the set of connections between the terminals, the routing problem for a given net is to find a set of non-intersecting trees embedded in G that connects the source terminals of the net to each of their sink terminals. Such an embedding problem is a tough one and is usually solved using heuristic methods and iterative approaches that gradually converge to an acceptable solution.

2.3. Dynamic partial reconfiguration

As we saw earlier, the transition from hardware description to the configuration bitstream consists of many complex and time-consuming steps that are infeasible to execute during runtime and based on application needs. On the other hand, the relatively large configuration bitstream and the serial nature of the configuration process make the configuration process slow. As a result, most FPGAs are not dynamically reconfigurable, and if they are, they are merely partially dynamically reconfigurable with some limitations [8].

Dynamic partial reconfiguration (DRP) in FPGAs permits changing a portion of the device while the rest is still running; This allows the FPGA to adapt to changing hardware algorithms, improve fault tolerance and resource utilization, enhance performance, or reduce power consumption. DPR is also helpful in situations where devices cannot be disrupted for subsystem reconfiguration;

The difference-based partial reconfiguration and the module-based partial reconfiguration are two primary forms of dynamic reconfiguration on an FPGA. Difference-based partial reconfiguration can be applied for slight changes to the design; It is notably helpful when updating the content of Look-Up Table (LUT) equations or specialized memory blocks.

Modular design concepts, on the other hand, are used to rearrange large sections of the design. Partial Reconfiguration Module (PRM) is a design module that can be swapped in and out of the device on the fly. The Partial Reconfiguration Region (PRR) is a section of the FPGA reserved for partially reconfigurable modules. Multiple PRRs can be set on the fly. Bus macros (BMs) are macros that have been pre-placed and pre-routed. All non-global signals between PRMs and static modules must be routed via BMs. The overall architecture of modular-based partial reconfiguration is illustrated in Fig. 5.

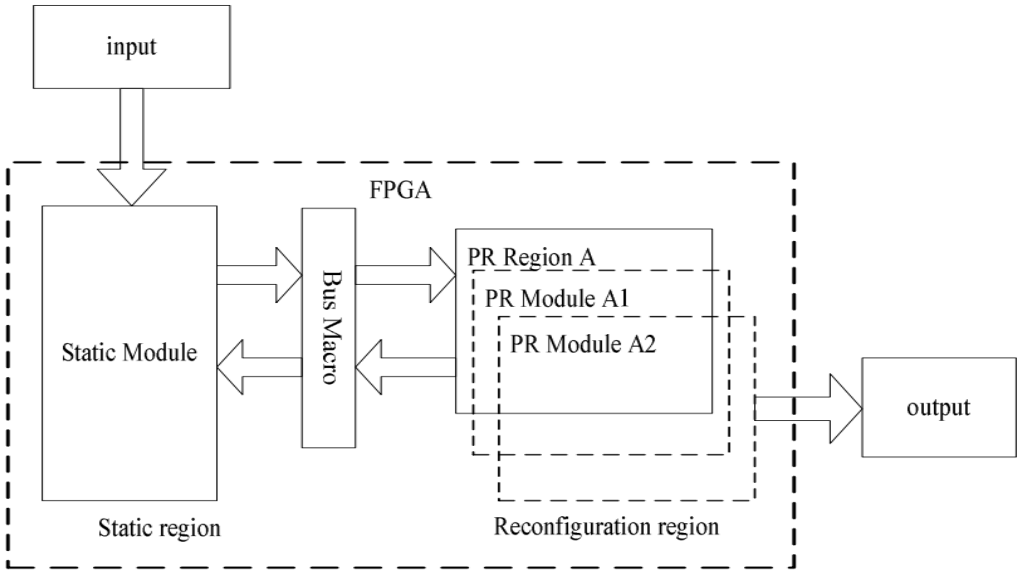

Fig. 5 The overall architecture of modular-based partial reconfiguration [8]

Because a reconfigurable module must meet specific properties and layout criteria, any FPGA design that intends to use partial reconfiguration must be planned and laid out with that in mind.

3. Coarse-grained reconfigurable architectures

Coarse-grained reconfigurable architectures (CGRAs) are dynamically reconfigurable fabrics that need fewer configuration data and shorter configuration time than fine-grained ones. CGRAs, compared to FPGAs, eliminate the overheads associated with the fine-grained architecture of FPGAs by replacing LUTs with coarser computational blocks and simplifying interconnect patterns. As a result, they can make more aggressive power/area trade-offs while sacrificing some of the FPGAs' flexibility; They aim to combine the runtime programmability of general-purpose microprocessors with the performance of spatial computation hardware platforms [2].

CGRAs are essentially spatial architectures with advantages in both compute-intensive and data-intensive applications, and they showed encouraging results when applied in neural networks, cryptography, multimedia, and signal processing accelerators; However, there are challenges in terms such as the architecture models, programming models, and compilation process, which we will briefly discuss next.

3.1. Architecture models

CGRAs are spatial computing architectures with time-multiplexed hardware resources; they have many parallel computing resources and, like ASICs, can be driven by the data flow. As a result, the performance of a CGRA is typically superior to that of a CPU. A CGRA's hardware resources, like those of an FPGA, can be reorganized by changing the configuration to suit the application. A CGRA, on the other hand, has much fewer configuration data and configuration time than an FPGA, allowing it to switch configuration while computing.

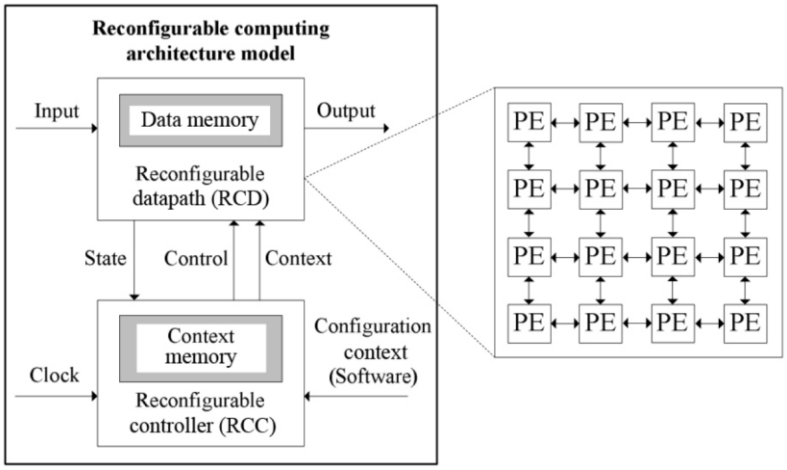

Fig. 6 Architecture model of a DRCA chip [1]

Fig. 6 depicts the architecture model of a Dynamically Reconfigurable Architecture (DRCA). The Reconfigurable datapath (RCD) is a two-dimensional Processing element

array (PEA) with a data memory. Each processing element (PE) function may differ depending on the application's field of interest, and the PE array's operation is data-driven. The Reconfigurable controller (RCC) is a programmable finite state machine (FSM) that reads the information of data flow, control flow, and configuration flow from the outside, which is so-called "software," and performs a state flow graph to control the execution of a task. Each state corresponds to a subtask. RCC controls the datapath to complete the configuration and execution processes of each subtask.

3.2. Programming models

So far, there is no unified programming model for CGRAs, which is the main reason why CGRAs have not been widely adopted yet [1]. Designing a programming model for a CGRA is much more complicated than designing one for a CPU because it must be capable of scheduling and coordinating the resources of a two-dimensional reconfigurable array. For a CGRA programming model to be both practical and efficient, it must provide a high abstraction level (to hide hardware details from the programmer) while expressing features that exploit the potential performance of the complex target hardware.

3.3. Compilation process

Completing dynamically reconfigurable architecture programs is challenging because the compiler must orchestrate many programmable resources; The main task is to generate control codes for the controller and configuration information for the datapath.

Generally speaking, the compilation includes code transformation and optimization, task partition, task scheduling, mapping, and configuration generation, but the details of these steps are architecture-dependent and may vary between different architectures.

We can classify CGRA compilation into two classes, static and dynamic. Static compilation maps input kernels onto the entire spatial architecture before runtime, whereas dynamic compilation generates valid mappings during runtime. The instruction-based method and the configuration-based method are the two types of dynamic compilation methods. The former converts CPU instruction flow into CGRA configurations during runtime, whereas the latter converts static configurations into runtime configurations. Dynamic compilation can usually provide better resource utilization at the cost of compilation overhead during execution.

3.4. Architecture virtualization

Variable parameters of the underlying hardware and the divergence in the CGRA structures make compilation challenging and device-specific. As illustrated in Fig. 7, virtualization provides a unified CGRA model, named virtualized CGRA, which comprises standardized interfaces, communication protocols, and execution abstraction; Based on this model, first, the compiler compiles input applications into virtual configurations that fit the target

CGRAs, then, the virtual configurations are optimized and interpreted online or offline for a specialized CGRA. The system scheduler uses the generated configuration binaries to determine runtime task placement and eviction by monitoring resource utilization and system states.

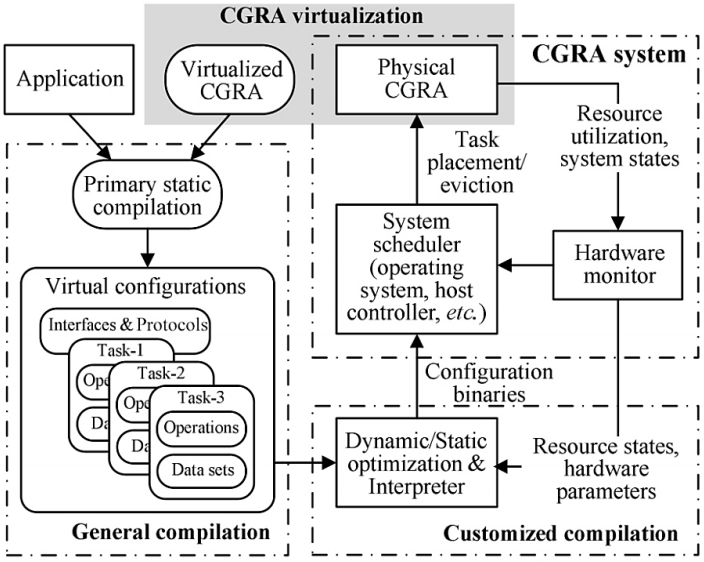

Fig. 7 CGRA virtualization [1]

4. Hybrid architectures

In practice, most reconfigurable architectures combine different parts such as General Propose Processing (GPP) cores, fine-grained reconfigurable FPGA cores, coarse-grained dynamically reconfigurable cores, and ASIC parts. We will see examples of such architectures in the next section.

5. Real-world examples

Next, we will see some of the real-world examples to clarify some of the discussed concepts.

5.1. Example of high-level Synthesis flow of a DRP

Fig. 8(a) shows the basic building block of the target DRP [3], called a "tile," and comprises many processing elements (PEs) with memory units arranged on it. Based on configuration codes stored in each PE, the operations to perform and the wires to use between the PEs and other resources, such as on-chip memories and external ports, are chosen. The

sequencer, also known as a state transition controller (STC), selects the configuration in one clock cycle. Each PE has an 8-bit arithmetic logic unit (ALU) shown in Fig. 8(b), a data manipulation unit (DMU) for both 8-bit shift/mask operations and 1-bit logic operations, an 8-bit register file unit (RFU), an 8-bit flip-flop unit (FFU), and wire switches. The DRP can have an arbitrary number of tiles.

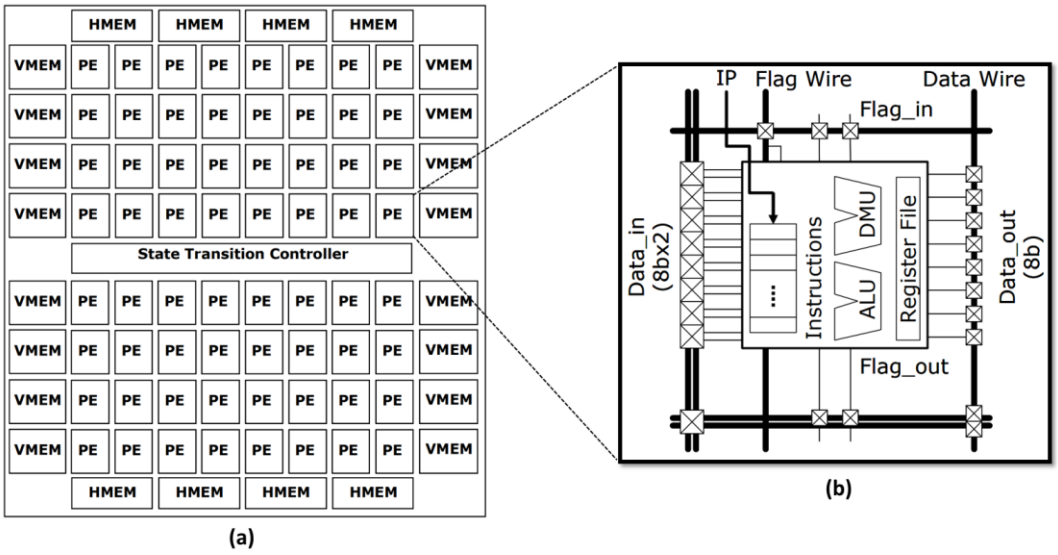

Fig. 8 The target DRP architecture. (a) Structure of tile in the DRP. (b) The PE architecture [3]

Fig. 9 depicts the compilation flow of the compiler developed for this architecture. The compiler inputs C, or behavioral design language (BDL), and outputs downloadable configuration code; It extracts instances of parallelism by generating a control data-flow graph (CDFG) that splits up the description of each step based on given constraints.

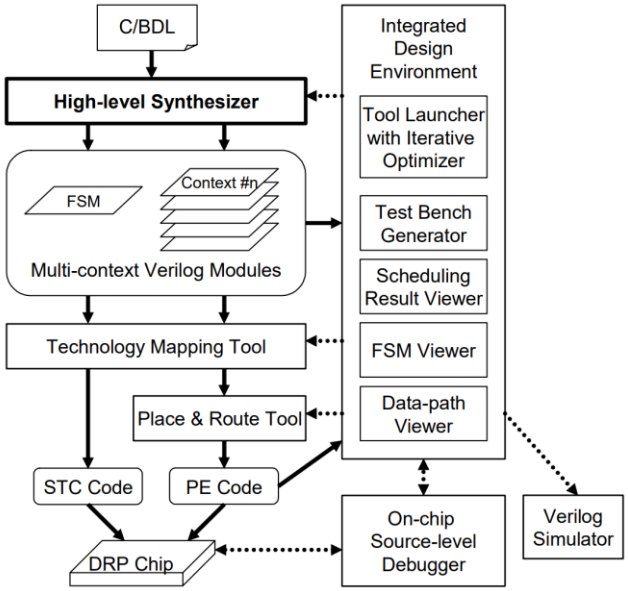

Fig. 9 Compilation flow and design environment for the DRP [3]

Fig. 10 illustrates how each step of the CDFG is mapped to a configuration context. The leftmost half of this figure shows the communication path between the data-flow graph and the DRP resources.

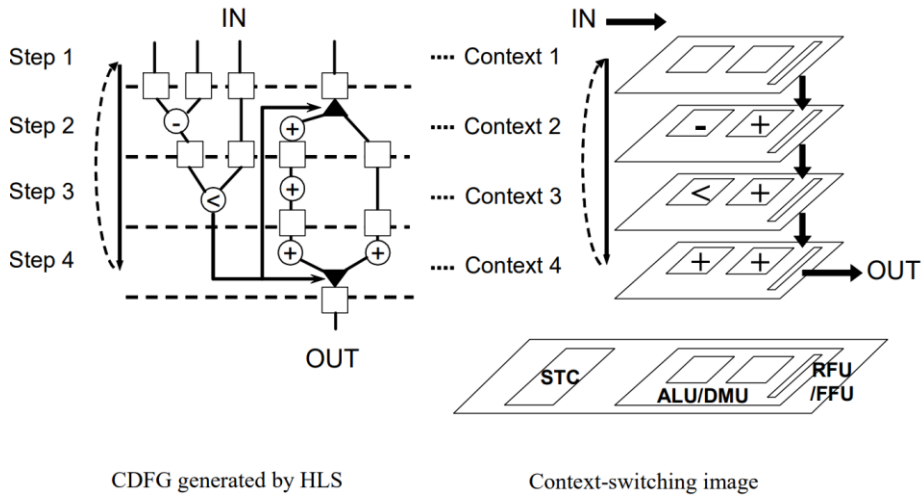

Fig. 10 Relationships between CDFG steps and contexts [3]

Finally, the compiler outputs STC code and PE code; The STS code is used by the state transition controller to runtime configure the PEs' functionality and interconnections, and the PE code is distributed among PEs and executed as instructions.

5.2. HReA: A general propose DRCA example

HReA [6] stands for Hybrid-grained Reconfigurable Architecture, and it was developed to process 13-Dwarfs. The proposed dynamically reconfigurable fabric is made up of four 4x4 multi-functional Processing Element (PE) arrays, with a hybrid-grained structure that combines a 32-bit and a 1-bit data stream to handle numerous computing granularities in 13-Dwarfs. A dwarf is an algorithmic method for capturing typical processing and communication patterns in a group of important applications. Using kernels from the 13-Dwarfs to evaluate system performance and efficiency is regarded as a proper assessment of computing architectures designed for general computing.

Processing Element Arrays (PEA), PEA microcontroller, and master microcontroller are the three main functional parts of the proposed HReA architecture, depicted in Fig. 11. A Direct Memory Access Controller (DMAC), embedded SRAM (ESRAM), and other common peripherals such as an interrupt controller (INTC), timer, UART, and system controller are also included in this architecture. On-chip caching, including configuration cache and data cache, are also employed to reduce the required off-chip memory bandwidth.

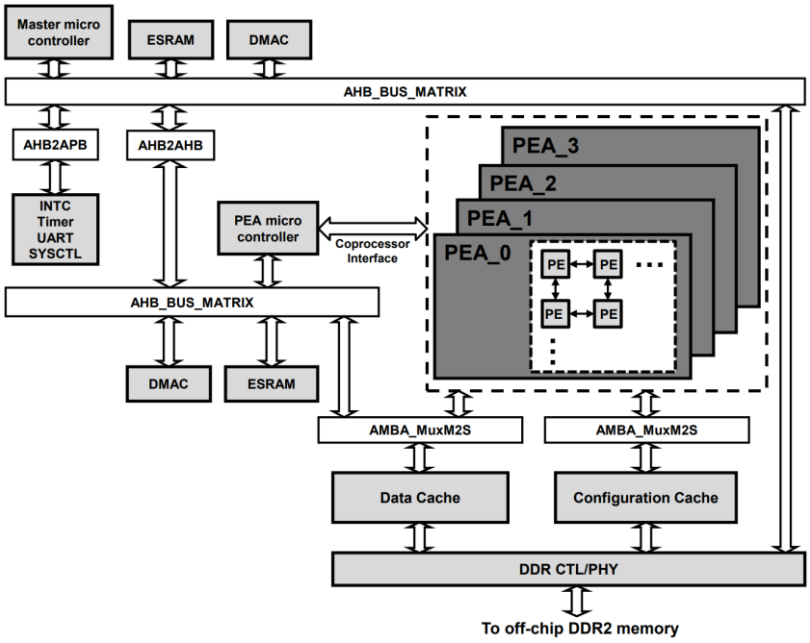

Fig. 11 HReA system architecture [6]

As depicted in Fig. 12, PEA contains a 4×4 hybrid-grained PE array, where each PE can be dynamically configured to execute arithmetic and logic operations under the control of configuration context. It contains auxiliary components, including host interface, PEA controller, configuration controller, and a data controller, to prepare control signal, configuration, and operand data for the PE's array. The core part of PEA is responsible for fetching, processing, storing, and exporting data flow driven by control flow and configuration flow.

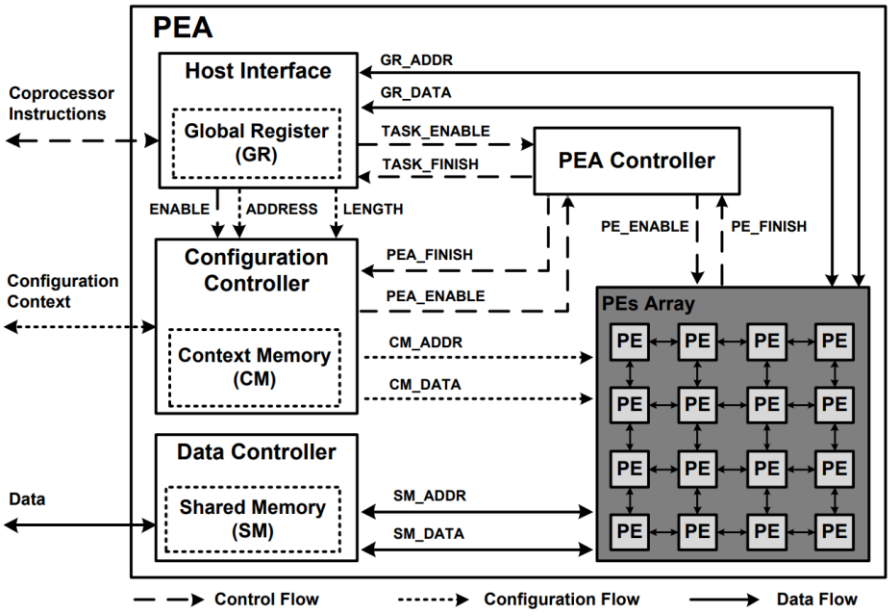

Fig. 12 P EA architecture [6]

The architecture of a PE is depicted in Fig. 13(a); the main computational element of each PE is a 32-bit Arithmetic Logic Unit (ALU), which can integrate up to 15 different operators. The ALU's calculation results can be saved in the inner register file for short-term storage or loaded into shared memory using the Load Store Unit (LSU). Each PE can be connected to any PEs in the adjacent positions. The interconnections between PEs can be dynamically reconfigured based on routers' configuration context.

As shown in Fig. 13(b), a hybrid-grained data path structure is designed to accelerate both compute-intensive and control-intensive applications. The ALU output can be chosen based on the results of the 1-Bit data path. As a result, the PEA can use the 1 Bit data path to execute controlling statements (such as "if-else" and loops). Fig. 14 shows a simple "if-else" as an example; three PEs constituting a two-stage pipeline are employed. PE1 is assigned to obtain a 1-bit statement of conditional judgment in the first stage of the pipeline, while PE2 and PE3 complete 32-bit expressions in "if-else" branches, respectively. At the next pipeline stage, PE2 chooses one result from expression1 and expression2 according to the 1-bit judgment condition.

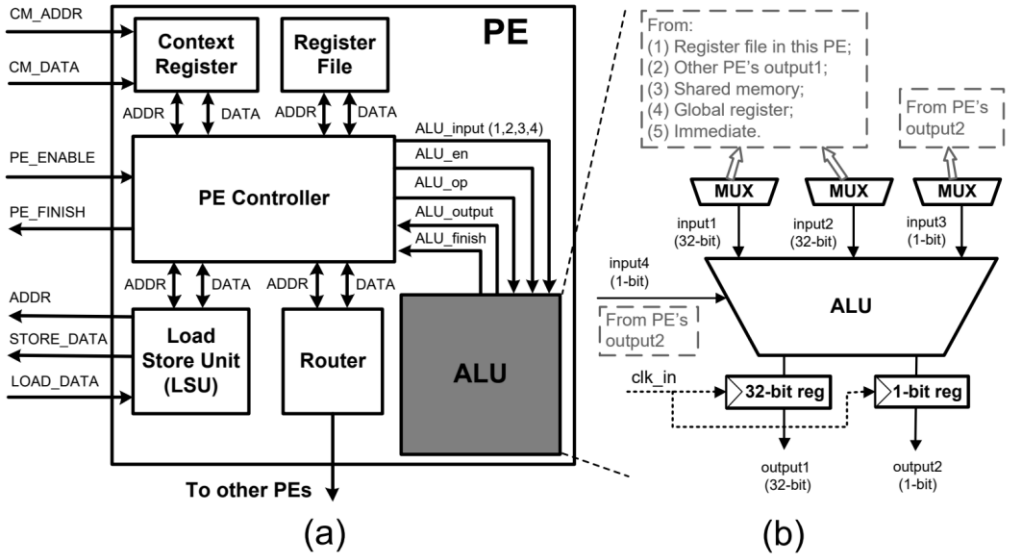

Fig. 13 PE and ALU architectures. (a) PE architecture, (b) ALU architecture [6]

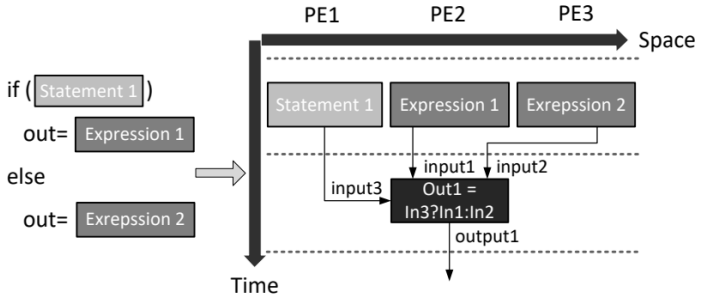

Fig. 14 Execution of a "if-else" condition on a PEA [6]

Fig. 15 depicts the execution of kernel code on HReA. The kernel's core loop consumes the majority of the computational complexity and can be accelerated on PEAs. PEA is capable of parallel processing in both space and time dimensions. Multiple PEs form a pipeline to speed up computation when computing multiple sequential stages in each iteration.

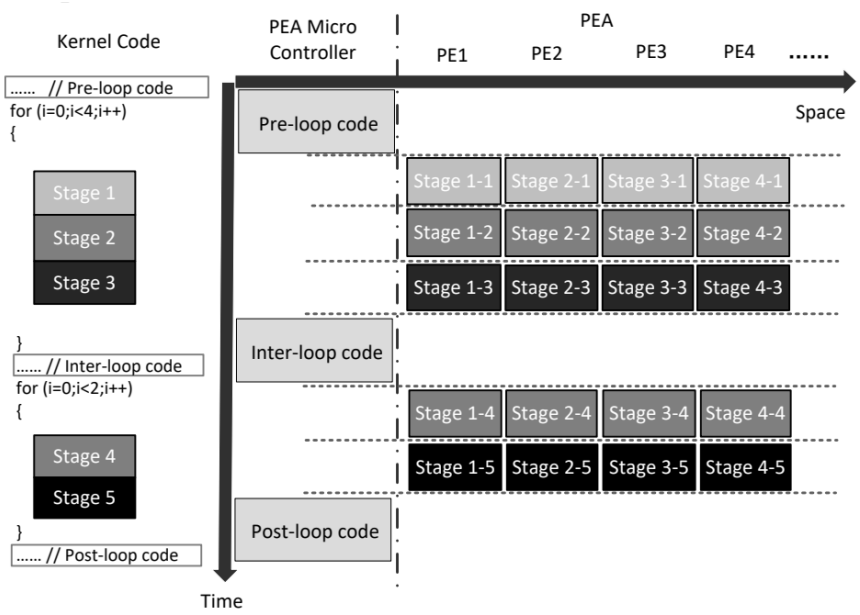

Fig. 15 HReA execution flow [6]

HReA achieves much higher energy efficiency improvements compared to Atom N550 and Cortex A15, i.e., 33.13x and 15.17x, respectively, when normalized to the same technology.

5.3. Hybrid architecture examples

The overall structure of a heterogeneous reconfigurable digital signal processor, proposed by Rossi et al. [2], is shown in Fig. 16 as an example. The device can handle a wide range of tasks thanks to a well-balanced mix of heterogeneous reconfigurable fabrics linked by a flexible and efficient communication infrastructure based on a 64-bit Network On Chip. A fine-grain embedded FPGA, a mid-grain configurable processor, and a coarse grain reconfigurable array are included in the SoC. The SoC supervisor is an ARM processor with a resident operating system that manages communication, synchronization, and reconfiguration mechanisms. The programmer can use the ARM processor to manage the high-level synchronization and global data of complex signal processing applications while allocating the most critical computational kernels to the most appropriate reconfigurable engines using this computational model.

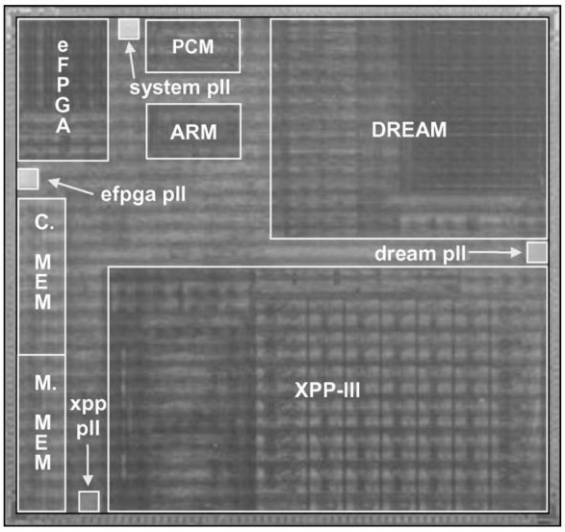

Fig. 16 The overall chip structure [2]

We can mention the dynamically reconfigurable AI accelerator proposed by Fujii et al. [4] as another hybrid example where dynamically reconfigurable fabrics are integrated with hard-wired logic. The architecture is a dynamically reconfigurable processor (DRP) for accelerating deep neural networks (DNNs) in embedded microprocessor systems. A DRP unit and a multiply-and-accumulate (MAC) unit are tightly integrated into an STP-3 AI core to achieve high versatility, high performance, and low latency DNN processing. As shown in Fig. 17, the STP-3 engine is composed of versatile DRP and DMA units. AI-MAC, as depicted in fig. 18, is composed of 16 64-MAC channels. Four of them (MAC group) share a single 64b input/output FIFO interface to DRP. The DRP transmits both input and output data of MAC units. DRP pulls data from AI-MAC one by one, post-processes them, and pushes back the output. Programmability is very high in this system because series of dataflow except MAC operations are controlled as a DRP program written in C language.

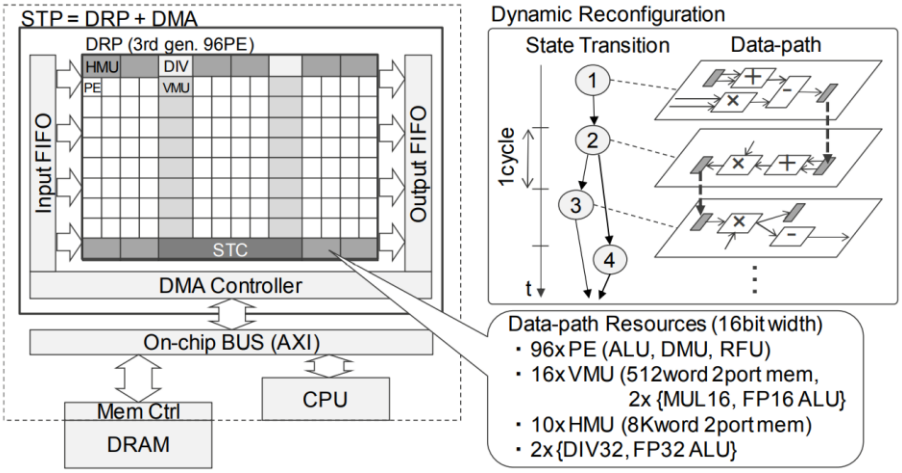

Fig. 17 STP Engine Concept and STP3 Design [4]

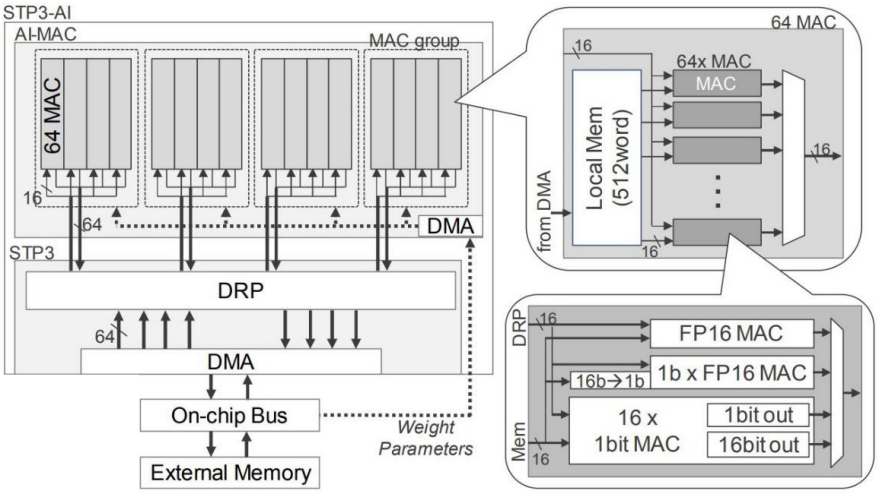

Fig. 18 AI-MAC Architecture [4]

6. Conclusions

Dynamically reconfigurable computing architectures, which show high energy efficiency and flexibility, are promising emerging computing architectures with the potential to be used in a broad range of applications. However, they are still at the early stages of development and are not yet ready to replace general-propose processors. Lack of a unified programming model and Architecture, alongside the much-complicated compilation process, might be the main reason they are not widely adopted yet.

References

[1]. S. Wei and Y. Lu, "The Principle and Progress of Dynamically Reconfigurable Computing Technologies, " Chinese Journal of Electronics, Volume 29, Issue 4, pp. 595 – 607, 2020.

[2]. D. Rossi, F. Campi, S. Spolzino, S. Pucillo and R. Guerrieri, "A Heterogeneous Digital Signal Processor for Dynamically Reconfigurable Computing," in IEEE Journal of Solid-State Circuits, vol. 45, no. 8, pp. 1615-1626, 2010.

[3]. T. Toi et al., "High-level synthesis challenges and solutions for a dynamically reconfigurable processor, " In Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06). Association for Computing Machinery, New York, NY, USA, pp. 702–708, 2016.

[4]. T. Fujii et al., "New Generation Dynamically Reconfigurable Processor Technology for Accelerating Embedded AI Applications," 2018 IEEE Symposium on VLSI Circuits, pp. 41-42, 2018.

[5]. I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203-215, Feb. 2007, doi: 10.1109/TCAD.2006.884574.

[6]. L. Liu, Z. Li, C. Yang, C. Deng, S. Yin and S. Wei, "HReA: An Energy-Efficient Embedded Dynamically Reconfigurable Fabric for 13-Dwarfs Processing," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 3, pp. 381-385, 2018.

[7]. U. Farooq, Z. Marrakchi, and H. Mehrez, "Tree-based heterogeneous FPGA architectures: application specific exploration and optimization," Springer Science & Business Media, 2012.

[8]. W. Lie and W. Feng-yan, "Dynamic Partial Reconfiguration in FPGAs," 2009 Third International Symposium on Intelligent Information Technology Application, pp. 445-448, 2009.