Just-In-Time Composition of Reconfigurable Overlays

Rafael Zamacola
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Andrés Otero
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Alfonso Rodríguez
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Eduardo de la Torre
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Abstract

This paper describes a framework supporting the automatic composition of reconfigurable overlays laid on top of an FPGA to offload computing-intensive sections of a given application, from an embedded processor to a loosely coupled reconfigurable accelerator. Overlays provide an abstraction layer acting as an intermediate fabric between users' applications and the FPGA fabric. Among the existing flavors, the overlay template proposed in this work is based on a coarse-grain reconfigurable architecture featuring word-level operators, reducing long place-and-route times associated with FPGA designs. The proposed overlays are composed at run-time using a tile-based approach, in which pre-synthesized processing elements are stitched together following a 2D grid pattern and using dynamic and partial reconfiguration. The proposed reconfigurable architecture is accompanied by an automated toolchain that, relying on an LLVM intermediate representation, automatically converts the source code to a data-flow graph that is afterward mapped onto the overlay. A mapping example is provided in this paper to show the possibilities enabled by the framework, including loop mapping and loop unrolling support, features originally described in this work.

2012 ACM Subject Classification Hardware → High-level and register-transfer level synthesis

Keywords and phrases FPGA, Dynamic Partial Reconfiguration, Overlay, LLVM, Compilation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.2

Category Invited Talk

Funding This project has been funded by the Spanish Ministry for Science and Innovation under the project TALENT (ref. PID2020-116417RB-C42).

1 Introduction

The lack of accessibility of FPGAs to software programmers has been traditionally considered the main barrier to bringing them to mainstream computation, reducing their use to a limited group of hardware designers [1]. For this reason, there has been a growing interest during the last few years in finding alternatives to Hardware Description Languages (HDLs) to program the FPGAs. This limitation has led to the popularization of High-Level Synthesis (HLS) tools that use software-based languages as the entry point [13].

Currently, there are successful commercial HLS tools such as Vitis High-Level Synthesis [17] and Intel High-Level Synthesis Compiler [9] but also academic open-source HLS tools, such as [3]. Despite their many advantages, HLS tools still face some challenges to be of use by designers with little hardware design knowledge. First, to make efficient circuits, it is necessary to apply optimizations that require knowledge of the underlying hardware platform. Second, the integration of the generated accelerators with the rest of the system has to be

done manually with many HLS tools [8]. In addition, and while HLS tools partially solve the lack of accessibility, they do not solve the slow implementation cycles that characterize FPGA designs [15]. In particular, FPGA designs suffer from long synthesis and Place and Route (P&R) times, what is termed as the FPGA programmability wall [14].

This programmability wall is largely due to the fine granularity of the FPGA architectures that are configurable at bit level. That is why novel approaches for virtualizing the FPGA resources appear in the state-of-the-art. Among them are overlays, a pre-compiled FPGA circuit with a programmable architecture [4]. Overlays provide a higher abstraction layer of FPGA resources acting as an intermediate fabric between user applications and the reconfigurable fabric. Different architectures can be implemented as overlays, including soft processors, arrays of soft processors, and Coarse-Grained Reconfigurable Architectures (CGRA) [11]. CGRAs provide word-level operators and special-purpose interconnections. In CGRA-based overlays, different applications, usually represented as Data-flow Graphs (DFGs), can be mapped. This process is represented in Figure 1. Implementing CGRAs on top of FPGAs using overlays has several advantages over using the FPGA resources directly. First, they can reduce the P&R times by orders of magnitude. Second, they serve as an FPGA virtualization method to make designs portable across different devices. Lastly, they allow rapid reconfiguration to change between different applications.

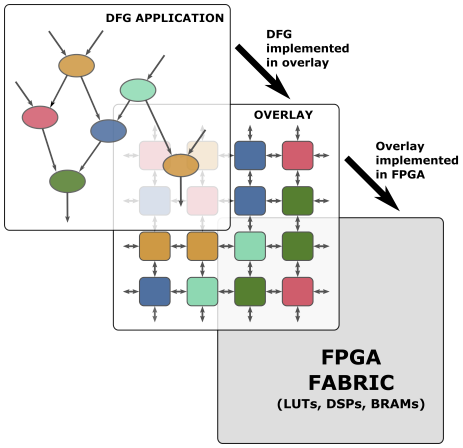

Figure 1 Overlays form an intermediate fabric between the FPGA and the applications.

An appealing option for overlay configuration is to leverage FPGAs' Dynamic Partial Reconfiguration (DPR) mechanism. DPR allows reconfiguring an area of the FPGA while the rest of the system remains working and unaltered. One of the primary purposes of DPR is to time-multiplex the reconfigurable resources, allocating only the accelerators used at any given time, thus reducing the required area on the FPGA. As a counterpoint, users applying DPR have to face many challenges related to low-level access to the device configuration memory. However, changes introduced in commercial tools during the last years changed DPR perception so that it is no longer seen as a complicated technique only used by experts [16]. The interest in using FPGAs in data centers for cloud computing [2] has undoubtedly contributed to making the use of Dynamic and Partial Reconfiguration more widespread.

This paper gathers a set of research activities aiming to provide a design framework and several architectures that leverage DPR to compose reconfigurable overlays on the fly, starting from software descriptions using high-level languages. It is based on a custom hardware

composition technique to generate new hardware accelerators by combining pre-implemented logic modules while the system is being executed. This way, by stitching these modules together, it is possible to generate new accelerators without implementing them from scratch, reducing the implementation time. This proposal is enabled by IMPRESS [21], a design tool to implement highly flexible reconfigurable systems combining multiple granularity levels. Accelerators are described starting from software-based descriptions, which enables the offloading of intensive computing tasks from the embedded processor to the FPGA. The accompanying toolchain automatically converts the source code to a data-flow graph that is afterward mapped onto the overlay. The proposal targets heterogeneous systems such as the Xilinx Zynq-700 SoPCs that integrates an FPGA alongside a hard-core processor. A mapping example has been included in the paper to show the possibilities enabled by the framework, including loop mapping and loop unrolling support, features that are originally described in this work.

The rest of this paper is organized as follows. In Section 2, a discussion on the composition of hardware accelerators using a 2D arrangement of pre-synthesized reconfigurable processing elements is provided. Section 3 describes the architecture of the multi-grain reconfigurable overlay, while the accompanying software tools for mapping applications on the overlay are provided in section 4. In section 5, an application example is provided, while conclusions and future work are provided in section 6.

2 Tile-based Composition of Hardware Accelerators

DPR has been used in the state-of-the-art in combination with overlays to increase their flexibility by changing the PEs at run-time, adapting them to any given application [12]. However, previous approaches were restricted by the limitations that impose former commercial DPR flows, such as Xilinx [18] and Intel DPR flows [10]. Differently, in this work, the advanced configuration possibilities enabled by the academic tool IMPRESS [21] have been explored. Using these advanced features, the overlay composition described in this work follows a tile-based approach that generates new accelerators by stitching together PEs in a 2D grid pattern. This reconfiguration style is called medium-grain reconfiguration in IMPRESS. The implementation of the system starts by defining a dynamically reconfigurable region reserved for acceleration composition. The communication of the reconfigurable region with the rest of the system (including the attached processors) is defined at design time and remains fixed for all the reconfigurable accelerators. The implementation of individual PEs is carried out independently from the main system. The PEs include custom interfaces to connect to adjacent PEs or the rest of the system if the PE is allocated in the reconfigurable region border. Accelerator composition is carried out by reconfiguring the PEs in different subregions, forming a 2D regular architecture. Figure 2 illustrates the tile-based composition technique. The configuration of each PE can be parameterized by instantiating individual reconfigurable components whose behavior can be adapted reconfiguring FPGA Look-Up Tables (LUTs) at high speed. This is referred to as fine-grain reconfiguration, according to the IMPRESS terminology [19].

The main advantage of the proposed approach is the high flexibility that it offers. As there is only one reconfigurable region that can allocate multiple PEs, the size of the PEs is not fixed and can vary between accelerators. The communication interface between PEs is defined individually in each PE, allowing different PE interconnections tailored to a given accelerator. Moreover, the number of PEs is adapted to the accelerator requirements leaving the rest of the reconfigurable resources free to allocate other accelerators. This means that the

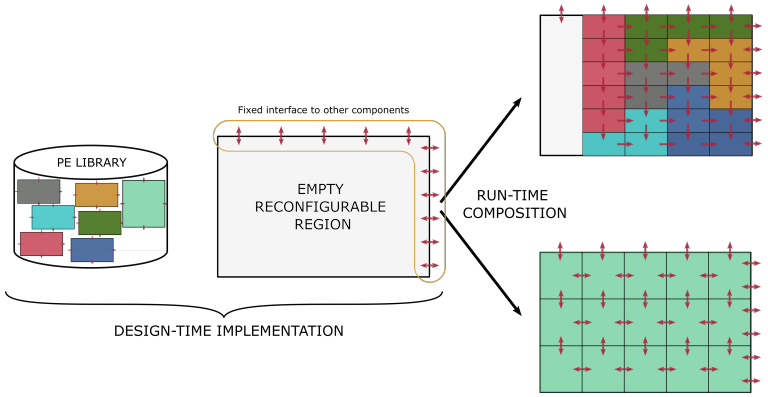

Figure 2 Tile based composition approach. At design time (left) it is necessary to implement a library with different PEs and an empty reconfigurable region that only defines the interconnection with the rest of the system. At run time (right) different accelerators can by composed on the fly by stitching together PEs from the library.

overlay is dynamically scalable. Finally, the individual PEs can be configured instantiating specific components (i.e., with fine-grain reconfiguration, see [21] for further details) that can be reconfigured quickly without having to reconfigure the whole PE and without needing a direct connection to the rest of the system, reducing the communication interconnections between the PEs and the rest of the system.

The combination of medium-grain reconfiguration to compose the overlay from scratch for each application, with the fine-grain reconfiguration to modify specific components within the overlay, is known as multi-grain reconfiguration. This is a unique feature of the overlays proposed in this work.

3 Reconfigurable Overlay Architecture

As explained in the introduction, the term overlay references CGRA-based overlays composed of an array of PEs surrounded by a configurable interconnect in this work. The CGRA used in this work is built on top of the baseline architecture described in [4].

Figure 3 shows the architecture for one PE. It is divided into different components. Blue boxes represent different Data-Driven Pipeline Units (DDPUs) that control data-flow. Yellow boxes are the routing elements that move the data through the PE. The synchronization components that ensure that data can fan out to multiple destinations are marked in brown boxes. Finally, the computing operations on the input data are performed on the FU, represented as a green box.

Each PE input/output is connected through a DDPU that controls data-flow while pipelining it. The DDPUs transfer data using the synchronous elastic protocol (SELF) [7]. This protocol uses a handshake distributed control with a pair of signals. The *valid* signal indicates whether the DDPU has data or is empty (i.e., it has a pipeline bubble), and the *accept* signal indicates whether the DDPU is stalled or can receive new data. The elastic pipeline of the overlay allows the input data to arrive at different times to the FU inputs. Therefore, it is necessary to include a synchronization component that ensures that data is forwarded to the FU only when both inputs have valid data. This is the goal of the *join* and *fork* components.

This work uses the baseline PE architecture proposed in [4] that supports the following 32-bit integer operators: *add*, *subtract*, *multiplication*, *shift left*, *shift right*, and *bitwise* logical operators. However, it has been modified, adding three extra features to support scalability,

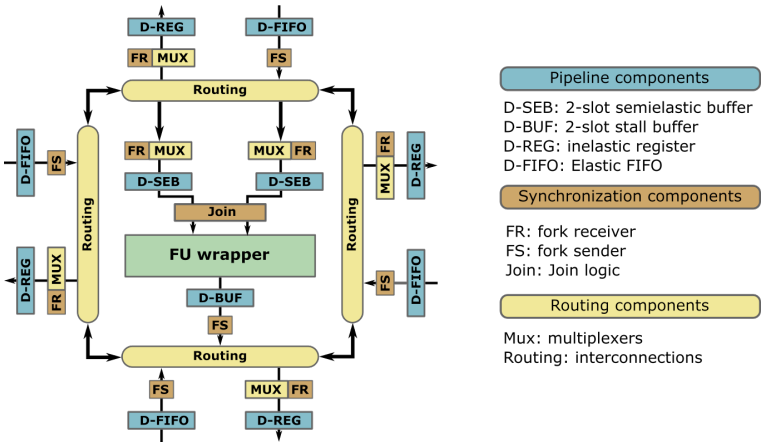

Figure 3 Baseline PE architecture. The PE is divided into four different components. In blue are the Data-Driven Pipeline Unit (DDPU), in yellow the routing elements, in brown the synchronization components and in green the FU. Figure extracted from [4].

improve routability, and support feedback loops. The first difference is that multiplexers and the parameters that configure the data paths are implemented with fine-grain reconfigurable components whose behavior can be modified by reconfiguring their LUTs. Moreover, the FU has been implemented using the fine-grain FU available in IMPRESS to implement add, subtraction, and bitwise operations by replacing only LUT constants in the FU. Second, constants must be mapped as inputs connected to the processor in the original PE. In contrast, in this proposal, a fine-grain reconfigurable parameter is inserted in each PE to map constants directly into the PE during the configuration phase. This approach facilitates the routing phase and reduces the number of inputs. Finally, the PE has been modified to implement feedback loops that allow routing the output of the FU to one of its inputs. This is necessary to implement typical operations, such as accumulators that reuse data from their previous iteration. When mapping complex nested loops, whole arrays of elements may be reused in subsequent iterations of outer loops. A FIFO has been included to store these feedback elements before reusing them.

To offload computation from the processor to the overlay, an infrastructure capable of transferring data between the two devices efficiently has been provided. The overlay has been implemented in a Xilinx Zynq-7000 SoC. Zynq-7000 SoCs provide high-performance AXI ports that can be used to transfer bursts of data between the FPGA and the processor memory. In particular, in the proposal described in this paper, we have used the Accelerator Coherency Port (ACP) port that ensures cache coherency in data transfers. The overlay wrapper includes a multiport memory that interfaces to external PEs of the overlay using input/output nodes that compute the address to access the correct data locations. The reader is referred to [20] for further details on the system integration.

4 Software Support for the Composition of Overlay Accelerators

The framework includes a supporting tool for mapping user applications onto the reconfigurable overlay. The entry language is C/C++, making it accessible to users without a hardware background. Only loop-based code sections are supported to be offloaded since they offer the highest ratio between the time associated with data transfers and the computation time. In particular, the proposed framework supports: (1) single loops, (2) nested loop block (with

up to three levels), (3) sequential loop blocks with data dependencies, where different loops are executed one after another, and (4) sequential loop blocks surrounded by an outer loop, as shown in Listing 1.

Listing 1 Sequential loops surrounded by outer loop.

```
for (i = 0; i < LOOP_IT_1 ; i++) {
    for (j = 0; j < LOOP_IT_2 ; j++) {
        for (k = 0; k < LOOP_IT_3 ; k++) {
            // DFG A
        }
    }
    for (l = 0; l < LOOP_IT_4 ; l++) {
        // DFG B
    }
}
```

However, there are still limitations to the loops that can be implemented in the current version of the framework. First, loops should not include any control statements such as if/else statements. Second, within a single loop block, each iteration must not carry dependencies with any other to allow data pipelining. The only exception is using variables that perform a computation using their previous iteration results, which happens, for instance, with accumulators that sum all the elements of an array. When computing sequential loop blocks surrounded by an outer loop, it is possible to have some variables that reuse the last n iteration results. This feature is supported by storing all the n elements inside a feedback FIFO available in the FU.

The next subsection provides an overview of the process followed by the tool to transform the source code into the configuration bitstream for the overlay. Then, specific transformations introduced in the framework to support feedback loops and loop unrolling are described.

4.1 Automatic Bitstream Generation Overview

The process starts by identifying the appropriate code sections to be accelerated, which the user must handle manually. Afterward, two steps are automatically carried out for each of the selected code sections. First, the source code is converted to a DFG, and then the DFG is mapped onto the overlay. These steps have been fully automated leveraging the CGRA-ME framework [6]. CGRA-ME is a unified framework encompassing a generic architecture description, architecture modeling, application mapping, and physical implementation. The primary goal of CGRA-ME is to provide a platform to investigate different CGRA architectures, algorithms, and applications. In this work, the original CGRA-ME has been extended with new features to adapt it to the proposed scalable overlay, adding support for transparently offloading applications from the processor and extending the support for complex loops (CGRA-ME only supports single loops).

Unlike the original CGRA-ME framework, where the user needed to tag the loops that were going to be mapped onto the overlay, in the proposed framework is necessary to wrap the loops inside a new software function. This strategy allows to quickly offload the application to the overlay by substituting the original function to another that directly composes the overlay, configures it, and then manages the input/output data transactions.

CGRA-ME relies on the LLVM compiler infrastructure to transform a C/C++ loop into a DFG. The LLVM infrastructure can be divided into front-end tools that transform high-level source languages to the LLVM Intermediate Representation (IR), middle-end optimization

passes that analyze and transform the IR, and back-end tools that compile the IR to machine code. The LLVM infrastructure includes a collection of classes and methods that can be used to generate custom LLVM passes to inspect and transform the IR as required by each specific overlay. Based on this infrastructure, CGRA-ME relies on *clang* to compile a C/C++ application to the IR and a custom loop-based LLVM pass to analyze all the IR instructions inside loops to generate a DFG. A new pass function has been developed in this work to extend the original CGRA-ME functionalities.

The new features added to the LLVM pass are described next. First, the pass type has changed from a loop pass to a function pass that allows iterating over each Basic Block (BB) of the function. A BB is a sequence of instructions followed by one branching instruction at the end. Therefore, when a program enters a basic block, it is executed until it reaches its end and jumps to another BB. The first thing the proposed LLVM pass does is to identify all the instructions of each BB to classify it in one of the following categories: (1) function entry point, (2) loop header, (3) innermost loop, (4) loop exit, and (5) function exit. The function entry and exit points are discarded, and the rest are stored in a BB array to analyze each BB's structure later.

Given the list of BBs, the LLVM pass checks the number of innermost loops to determine the number of DFGs. If there are two or more DFGs, the LLVM pass analyzes the BBs to catch any outer loops surrounding the inner loops. The condition for an outer loop is that the first and last BBs are the loop header and loop exit, respectively, and the exit has a branch instruction that can jump to the loop header BB. If there is an outer loop, the iteration variable is analyzed to get its name and the loop iteration limits (i.e., *initial value*, *step*, and *last value*). Then, all the remaining BBs are analyzed to get the number of nested loops and their loop iteration limits for each DFG. After this process, the structure of each BB is analyzed, and it is possible to start obtaining the DFGs of the innermost loops.

Then, the LLVM pass converts each instruction into a node of the DFG, while the dependencies between instructions are represented as edges. Another difference concerning the original LLVM pass is how *load/store* instructions are managed. Originally, a *load/store* instruction was represented as a special node that received the offset from the base address as an argument. Therefore, the DFG included the instructions necessary to compute these offsets. The proposed overlay architecture includes specific configurable input/output nodes that can autonomously compute the offset address to access the memory in each iteration. This work makes it possible to map the *load/store* instruction into these nodes without adding specific nodes to the DFG. It is now necessary to analyze the precedent instructions to obtain the base address and the stride for each nested loop to configure the input/output nodes of the architecture. The current LLVM pass can identify the instruction patterns that are used to compute the following indices: *[i]*, *[offset]*, *[i+offset]*, *[arg]*, *[arg+offset]*, where *i* is the iteration variable of one loop and *arg* is another input. The current proposal supports arrays with up to three dimensions. Once the LLVM pass has generated the DFG, it removes all the leaf nodes that do not generate any output value.

The next step is to map it onto the overlay. This proposal uses the architecture-agnostic Integer Linear Programming (ILP) P&R algorithm [5] provided in the CGRA-ME framework. This algorithm takes a description of the overlay architecture and the DFG and generates an optimal mapping. To use the mapper, it has been necessary to describe the overlay using the C++ Application Programming Interface (API) provided in CGRA-ME. The tool generates two output files. The first file is the overlay bitstream, a binary file that describes the configuration of the overlay using four configuration words per PE to specify its location, the FU used, and the value of the PE parameters and multiplexers. The second file includes

a function that replaces the original code function with an equivalent one that offloads the computation to the overlay. The new function performs three steps. First, it reads the overlay bitstream to compose the overlay using medium-grain reconfiguration and configures it using fine-grain reconfiguration. Second, it configures the overlay's input/output nodes obtained from the attributes of the DFG. These two steps are only necessary the first time the application is offloaded. Finally, the function transfers the input data to the overlay, waits until it finishes its computations, and then returns the overlay results to the processor memory.

Next, the specific modifications introduced to deal with feedback loops and loop unrolling are described.

4.2 Supporting Feedback Loops

The initial version of the DFG is generated directly by transforming every LLVM instruction into nodes according to the sequential instruction pipelining typical in microprocessors. Every iteration, the value to use is loaded, the computation is performed, and the result is stored in memory. Therefore, this first DFG does not show feedback loops explicitly. However, we know that a feedback loop occurs whenever the input value that is loaded has been stored in previous iterations. The LLVM pass has been modified as described below to identify these situations.

The identification process starts by analyzing all the output nodes to see if any nested loop has a stride value of zero, as this would indicate that the output node is writing to the same address every iteration. This situation suggests that another input node is probably reusing this value. Then, for all the output nodes that have been identified as potential feedback loops, the LLVM pass checks if there is an input node with the same attributes as the output node. When an input and output node share the same parameters and have an edge to the same computation node, this confirms the presence of a feedback loop in the node. Once a feedback loop has been identified, it is necessary to find the set of attributes to configure the corresponding FU. Feedback nodes require four attributes to be fully defined. The first two are basic attributes that are used to indicate the presence of a feedback loop (*unitary_loop*) and to indicate which of the two operands of the FU is used to route the feedback loop. The third attribute *iterations_reset* indicates the number of iterations that have to elapse to reset the accumulated value. Finally, the last attribute *loop_size* indicates the number of data elements that are reused. A *loop_size=0* indicates that the FU reuses the last result to compute the new value, while a *loop_size* larger than one indicates that the results of the FU need to be stored in the feedback FIFO before being reused.

4.3 Automatic loop unrolling

The CGRA-ME framework has also been modified to support loop unrolling. This feature consists in exploiting loop parallelism by simultaneously executing several copies of the DFG in the overlay. Loop unrolling can be effectively combined with the overlay scalability to map the same application, with different loop unrolling factors, onto overlays with different sizes. Therefore, allowing to trade-off the overlay performance with the number of resources used. Currently, loop unrolling is implemented only for applications with just one DFG, and, in case it has feedback loops, the *loop_size* attribute must be zero (i.e., loops that reuse their last result in their subsequent computation). Applying unrolling makes it necessary to modify the original DFG before calling the P&R tool. First, the DFG is replicated l times, where l is the loop unrolling factor. Then, it is necessary to change the attributes of the input/output nodes to distribute data accesses among all the replicated nodes equally.

Each replicated node needs an offset that is one stride apart from each other so that they access the first l consecutive elements. The offset of the *k-th* replicated node is `offset_0 = stride_0 * k`. The stride attribute also has to change to `stride_0 = stride_0 * l`, and it is common between all the replicated nodes. Finally, the total number of iterations has to be updated to `iterations_0 = (iterations_0 / l) + (((iterations_0 % l) < k) ? 1 : 0)` where the right side of the expression uses the ternary operator `?:` to indicate that it is necessary to add one iteration to the first m replicated nodes, where m is the remainder of the *iterations_0 / l* division. When the total number of iterations is not divisible by the loop unrolling factor, it can generate problems as the different replicated nodes will generate a different number of partial results. This circumstance has been solved by modifying the PE architecture, asserting the valid signal in the last iteration, and changing the maximum iterations in the output node to one.

When the DFG contains feedback loops, loop unrolling requires a few extra steps. In these cases, the loop is replicated to l feedback nodes that generate partial results that have to be computed with each other to obtain the final result. Then, it is necessary to remove the output nodes of the feedback loops and add extra computing nodes until getting the final result that is then forwarded to an output node.

5 Transformation Example

As an example of how the mapping process works, we are going to analyze the accumulate application shown in Listing 2. Figure 4 shows a comparison between the DFG obtained by the original CGRA-ME LLVM pass and the proposed one. Even for an application with unidimensional arrays like this one, the total number of nodes in the DFG is reduced from 18 to 9 nodes, allowing the application to be mapped onto smaller overlays. This reduction is due to the changes on how the input operands are read in this proposal, compared to the original CGRA-ME. The attributes of the input/output nodes are shown in the DFG text representation shown in Listing 3.

As an example, the attributes of the input *a[i+1]* are listed below (labelled as input0 in the Listing). The *[argNo=0][argType=reference]* attributes alongside the *[offset=1]* argument are used to obtain the base address of the input by specifying the function variable and its offset. The *[stride_0=1][iterations_0=1000]* attributes indicate that there are 1000 iterations and the address increases by one for each iteration. *[inner_loops=1][DFG_position=0]* are more relevant for applications with more than one DFG to indicate the number of direct nested loops (i.e., attribute *inner_loops*) and which DFG is executed first (i.e., attribute *DFG_position*).

Listing 2 Accumulation Application.

```
# define LOOP_SIZE_1 1000
void accumulate (int *a, int *b, int *c, int *sum) {
   int i;
   for (i = 0; i < LOOP_SIZE_1 ; i++) {
      c[i] *= a[i+1] + b[i -1];
      *sum += c[i];
   }
}
```

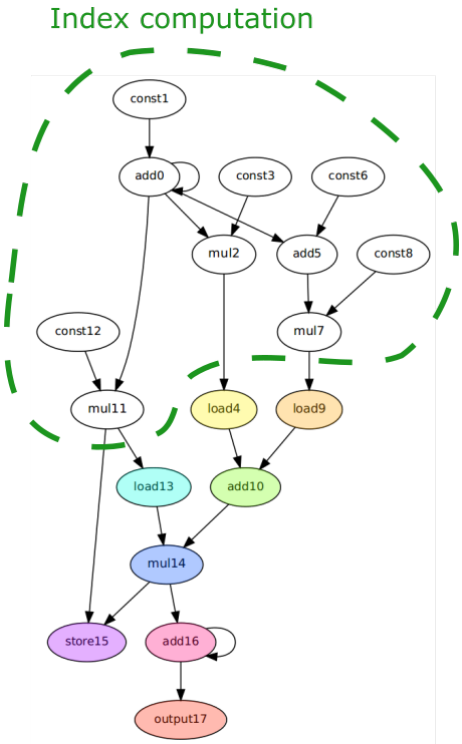

Figure 4 Original accumulator DFG generated by the CGRA-ME framework versus the proposed accumulator DFG. The proposed DFG discards all the nodes for computing the index and replaces them for attributes (e.g., initial address and stride) to configure the inputs/outputs nodes.

Listing 3 Attributes of the accumulate basic blocks.

```
digraph G {
input0[opcode=input][argNo=0][argType=reference][offset=1][inner_loops=1]
   [DFG_position=0] [stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input1 [opcode=input][argNo=1][argType=reference][offset=-1][inner_loops=1]
   [DFG_position=0][stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
add2[opcode=add][unitary_loop=0];
input3[opcode=input][argNo=2][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
mul4[opcode=mul][unitary_loop=0];
output5[opcode=output][argNo=2][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=1] [iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input6[opcode=input][argNo=3][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=0][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
add7[opcode=add][unitary_loop=1][iterations_reset=1000][loop_size=0]
   [loop_operand_pos=0];
output8 [opcode=output] [argNo=3] [argType=reference] [offset=0][inner_loops=1]
   [DFG_position=0][stride_0=0][iterations_0=1][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input0->add2[operand=1];
input1->add2[operand=0];
add2->mul4[operand=1];
input3->mul4[operand=0];
mul4->output5[operand=0];
mul4->add7[operand=1];
input6->add7[operand=0];
add7->output8[operand=0];
add7->add7[operand=0];
}
```

Figure 5 shows the mapping of the accumulate application DFG from Figure 4 onto a 2x3 overlay.

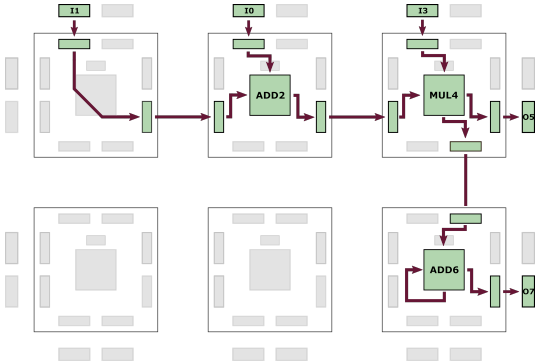

Figure 5 DFG to overlay mapping.

Finally, Regarding loop unrolling, an example can be seen in Figure 6 that replicates the accumulate DFG two times. The replicated DFG has added a new node (e.g., add17) to add the partial values generated by add7 and add16.

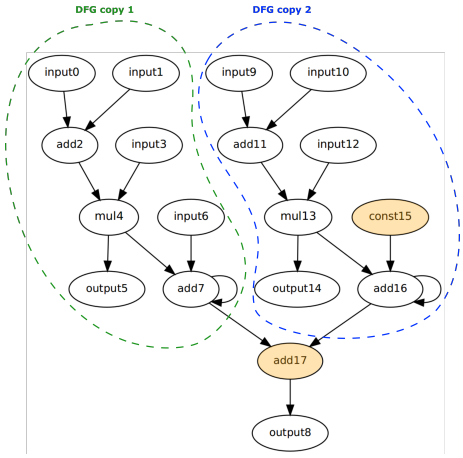

Figure 6 Accumulate DFG with a loop unrolling factor of two.

To evaluate the proposed overlay, we have used different applications that allow testing the performance of the proposed infrastructure and the behavioral correctness of the multi-grain reconfigurable method to build the overlay. The reader is referred to [20] for experimental results and performance metrics carried out with benchmark applications.

6 Conclusions and Future Work

This paper describes the effort carried out at the Centro de Electrónica Industrial of the Universidad Politécnica de Madrid related to the automatic generation of overlay accelerators. Proposed overlays take the form of a coarse-grain reconfigurable array, allowing the offloading of computing-intensive sections of code from the processor to the accelerator. The process followed to map a piece of code to the overlay is described in this work with one example.

This is an ongoing research work, which is now evolving towards integration with the RISC-V ecosystem. The overlay is being integrated with the RISC-V processor closer to the CPU, reducing the overhead associated with data transfers between the processor and the overlay. Moreover, future research plans include the development of run-time strategies for automatically deciding which parts of the code are offloaded at each time instant. Besides, the overlay is being applied to accelerate biomedical applications in the embedded computing domain.

References

1 David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses: The programmability of fpgas must improve if they are to be part of mainstream computing. *Queue*, 11(2):40–52, 2013.

2 Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

3 Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, 2013.

4 Davor Capalija. *Architecture, Mapping Algorithms and Physical Design of Mesh-of-Functional-Units FPGA Overlays for Pipelined Execution of Data Flow Graphs*. PhD thesis, University of Toronto (Canada), 2017.

5 S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

6 S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pages 184–189. IEEE, 2017.

7 Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Self: Specification and design of synchronous elastic circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*. Citeseer, 2006.

8 Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

9 Intel. Intel® high level synthesis compiler (user guide). URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/archives/ug-hls-18-0.pdf`.

10 Intel. Intel® partial reconfigurationuser guide, 2019. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-pr.pdf`.

11 Xiangwei Li and Douglas L Maskell. Time-multiplexed fpga overlay architectures: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(5):1–19, 2019.

12 Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–178, 2016.

13 Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

14 Mrunal Patel, Shenghsun Cho, Michael Ferdman, and Peter Milder. Runtime-programmable pipelines for model checkers on fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 51–58. IEEE, 2019.

15 Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

16 Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

17 Xilinx. Vitis high-level synthesis user guide (ug1399). URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf`.

18 Xilinx. Xilinx, inc., dynamic function exchange, 2021. URL: `https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf`.

19 Rafael Zamacola, Alberto García Martínez, Javier Mora, Andrés Otero, and Eduardo de la Torre. Automated tool and runtime support for fine-grain reconfiguration in highly flexible reconfigurable systems. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 307–307, 2019. `doi:10.1109/FCCM.2019.00048`.

20 Rafael Zamacola, Andres Otero, and Eduardo de la Torre. Multi-grain reconfigurable and scalable overlays for hardware accelerator composition. *Journal of Systems Architecture*, 121:102302, 2021.

21 Rafael Zamacola, Andrés Otero, Alberto García, and Eduardo De La Torre. An integrated approach and tool support for the design of fpga-based multi-grain reconfigurable systems. *IEEE Access*, 8:202133–202152, 2020. `doi:10.1109/ACCESS.2020.3036541`.