2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Scheduler for Inhomogeneous and Irregular CGRAs with Support for Complex Control Flow

Tajas Ruschke, Lukas Johannes Jung, Dennis Wolf and Christian Hochberger *Member, IEEE*
Department for Electrical Engineering and Information Technology
Computer Systems Group, TU Darmstadt
Email: {ruschke,jung,wolf,hochberger}@rs.tu-darmstadt.de

***Abstract*—Coarse Grained Reconfigurable Arrays (CGRA) are more area and energy efficient compared to FPGAs, if we consider applications that are dominated by arithmetical operations. Enabling the user to employ CGRAs requires tools to create suitable CGRA instances and to program them on a high abstraction level. In this contribution we briefly explain a CGRA archtiture generator and we focus on the scheduler that programs the generated CGRAs. This scheduler is able to work on inhomogeneous (concerning the operations in the processing elements) and irregular (concerning the connections between processing elements) compositions. Additionally, the scheduler is able to map complex control flow onto the CGRA, which means nested loops, even containing control flow in the loop body. This significantly increases the mappability of application kernels.**

I. Introduction

Programmable logic in the form of field programmable gate arrays (FPGAs) has become widely available and affordable. This fosters the idea to use FPGAs as customizable application accelerators. Particularly, SoC FPGAs like the Xilinx Zynq devices make this execution model attractive.

Unfortunately, engineering accelerators requires thorough background knowledge and experience in using the vendor specific toolset. Also, running the full toolflow for the implementation of one design can easily require several hours.

Although in the meantime tools are available that (semi-) automatically map application code to FPGA resources, this process still is not suitable for developers without FPGA background knowledge. Thus, a different kind of configurable architectures has been invented that can be configured on a higher abstraction level: Coarse Grained Reconfigurable Arrays (CGRAs). They consist of a set of processing elements (PEs) which can perform arithmetic and logic operations on word level. These PEs are connected by an interconnect that allows the transport of words between the PEs. Usually, this transport has a very low latency.

In the past, CGRAs have often been built in regular structure and with homogeneous operator selection. It turns out that it is possible to build inhomogeneous arrays (offering a different selection of operators in each PE) without loosing noteworthy performance [1]. Also, we can think of irregular structures minimizing the interconnect effort. It should be noted that currently we do not have a method to identify or construct such arrays. Rather, our current approach is based on experience and iteratively improving the CGRA compositions.

*This work was funded within the project UltraSynth by the German Federal Ministry for Education and Research with the funding ID 01 IS 15020. The responsibility for the content remains with the authors.

In this contribution, we briefly present our architecture generator that is able to compose an arbitrarily sized, inhomogeneous array of PEs forming a CGRAS. We will then focus on the scheduler that maps applications to a particular CARA composition. This scheduler is able to handle inhomogeneous and irregular arrays without any manual intervention. Also, the scheduler is able to map nested loops onto a CGRA, even if the loop body contains further control flow (nested if-else structures) or if the loop boundaries are not known at compile time. We demonstrate the functionality of generator and scheduler at the hand of an application example (ADPCM decoding). The main purpose of this example is to show that our toolset can automatically handle inhomogeneous and irregular CGRAs, not to make quantitative statements. The discussions in this paper are based on FPGA implementations of the generated CGRA compositions. In the context of highly varying application demands, it seems to be reasonable to use FPGAs also in real systems, since it allows to adapt the composition to the requirements of the target application domain. Nevertheless, the generated CGRA compositions can be implemented on ASICs with relatively small effort (yielding considerably higher clock frequencies and lower energy consumption).

The next Section discusses related work, while Section III presents our test environment. Section IV describes our specific architecture and the generator. In Section V, we explain the scheduler and its underlying operational principles. Afterwards, we evaluate the scheduler and the CGRA architecture using a complex application example. The last Section gives a short conclusion and an outlook onto future work.

II. Related Work

In the last few years many reconfigurable architectures were proposed as hardware accelerators. Initially, most approaches only mapped the data flow graph of basic blocks to the reconfigurable fabric. Thus all application kernels that need control flow could not be mapped.

Today, many approaches acknowledge the importance of control flow and the existence of nested loops. On a survey of publications many optimization fields can be recognized.

Firstly, there are methods to minimize the initiation interval for modulo scheduling to increase data throughput, e.g. SCC-based [2], EpiMap [3] and EMS [4]. All approaches are based on a DFG and do not consider the evaluation mechanism for control flow. PSB [5] achieves the lowest initiation interval by issuing selectively instructions only from the path taken by the branch at runtime. Path fusing and maximum clique heuristics are used. The approach holds eminent advantages for large outermost if-else structures, but is failing for more complex structures.

Secondly, there are modifications of modulo scheduling to improve the hardware utilization and execution performance, e.g.[6]. They use sophisticated techniques like affine loop transformations, but the approaches lack the possibility to map data dependent control flow to the CGRA.

Thirdly, there are optimizations for constructs including control flow such as if-else structures. BRMap [7] presents a dual issue predication. The approach achieves a small initiation interval. Main disadvantages of dual issue predication is the overhead in hardware due to necessity of loading two configurations per cycle. [8] suggests an architecture utilizing state-based full predication. Control flow is split and the resulting data flow graph is mapped using conventional mapping algorithms. However, only the innermost loop body is mapped.

Fourthly, there are control flow realizations from a more architectural point of view. In [9] not only basic blocks but so called Megablocks are mapped onto a reconfigurable processing unit (RPU). A Megablock is *one* most common path through a control flow graph. This technique increases the amount of code that can be accelerated on the RPU. If a branch is taken which was not added to the Megablock, the execution on the RPU has to be terminated and the GPP has to execute the subsequent code. The benefit of this approach for control-intensive applications is also limited.

In [10] an array of functional units (DySER) is integrated in the execution stage of an OpenSPARC processor. Simple control flow structures like if/else can be realized in the functional unit array but most of the control flow is handled by the OpenSPARC processor. Moreover, in every loop iteration each local variable has to be written to and read from the computation-slices. Again, control-intensive applications do not benefit from this architecture. *Just in time* approaches [11] and [12] need a general purpose unit as well to handle control flow.

Similar to that, in [13] control flow structures are integrated into the Layers-CGRA using a token-based predication method. However, this method requires a specialization to each application domain and complex control flow structures like data dependent loop execution are not supported.

Recently, predication is dominant in discussions about (performance of) control flow mechanisms. [14] gives a overview of techniques based on the architecture FloRA described in [15]. A variety of predication techniques is used: full and partial predication and state-based predication. All have in common that control flow can be mapped efficiently concerning performance or energy consumption. On the downside, every technique implies substantial limitations depending on the application.

Most approaches only focus on regular mesh architectures. [11] uses a crossbar interconnect to reduce the effort for place and route. This approach gives great performance but it is expensive and not scalable for higher numbers of PEs.

All mentioned techniques lack the ability to compute kernels including nested loops as well as complex control flow.

Our proposed technique supports all following features:

1) Arbitrary PE interconnect
2) Non-static and data dependent control flow in both loop body and nested loops
3) Resource and routing aware scheduling

To the best of our knowledge no previous work implements all these features in one framework.

III. AMIDAR AS TEST ENVIRONMENT

The CGRA used in this paper was designed to be integrated in an AMIDAR processor [16] in the first place. We will evaluate the proposed scheduler using the CGRA in this setup, but it has to be noted that the combination of the scheduler and the CGRA can operate as a hardware accelerator for any processor. Only the data exchange between host and CGRA have to be adapted. Apart from that the CGRA is completely independent of AMIDAR.

In this version, the AMIDAR processor executes Java bytecode directly. The bytecodes are broken down into a set of tokens which are distributed to a set of functional units (FU). These tokens carry the information about the type of operation that shall be executed, the version information of the input data that shall be processed (called tag) and the destination of the result.

The AMIDAR hardware profiler [17] is able to detect code sequences that are executed frequently. The execution of these sequences will then be mapped to the CGRA using the proposed scheduler and the optimization techniques described in [18]. The synthesis process is shown in Fig. 1. The first two

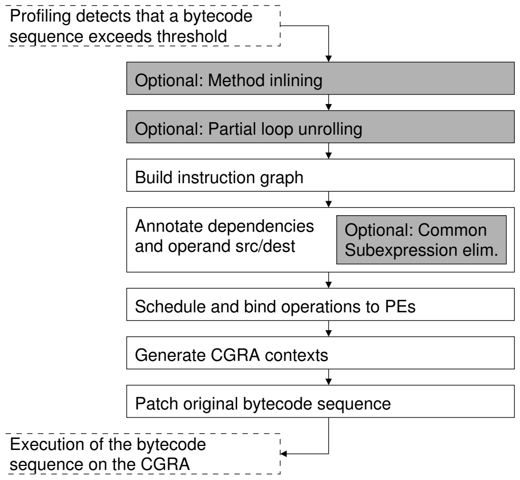
Profiling detects that a bytecode sequence exceeds threshold
Optional: Method inlining
Optional: Partial loop unrolling
Build instruction graph
Annotate dependencies and operand src/dest
Optional: Common Subexpression elim.
Schedule and bind operations to PEs
Generate CGRA contexts
Patch original bytecode sequence
Execution of the bytecode sequence on the CGRA

Fig. 1. Processing steps for the synthesis of an CGRA accelerator

steps in synthesis are optional. They perform method inlining and loop unrolling. From the resulting bytecode sequence an instruction graph is created. Afterwards, this graph is annotated with dependency information and information regarding sources and destinations of operands. This step can include common subexpression elimination. The resulting control and data flow graph (CDFG) is then subject to scheduling and binding. Each time the AMIDAR processor enters one of these code sequences, the processor forwards the execution to the CGRA and thus speeds up the execution of the application.

Data exchange between the AMIDAR processor and the CGRA works on two levels. Local variables are transferred to the CGRA via the internal data bus. After the execution, the local variables that may change their value during the execution are written back to the AMIDAR processor.

The heap memory stores arrays and object fields and is part of the AMIDAR processor. The CGRA can load required values via direct memory access (DMA). The exchange of local variables and the start of execution is controlled by the AMIDAR processor with the help of tokens. The CGRA handles memory accesses autonomously.

During CGRA execution the AMIDAR processor is idle as the CGRA handles all the control-flow autonomously.

IV. The CGRA concept

In this section the architecture of the CGRA and its generation is discussed. The generic architecture consists of an set of processing elements (PE) which are connected to each other. The structure of the interconnect, the number of PEs and their spectrum of operations can be highly inhomogeneous and is defined by the user. Therefore a generator is used to create the Verilog description for a *composition*.

A. Architecture

As the concept of CGRAs suggests, each PE is reconfigurable on word level. Therefore, one context memory is attached to each PE. The output of the memory drives all control signals of the PE and determines its configuration for each cycle. The context memory is addressed by a global context counter (CCNT) which is equivalent to a program counter. The CCNT is generated by a context control unit (CCU). In order to realize control flow it is necessary to perform branches. Hence, PEs have a status output which is routed to a Condition Box (C-Box). The C-Box generates a predication signal for store operations and a branch selection signal for conditional branches in the CCU. The C-Box is also controlled by a context memory. An overview is given in Fig. 2.

1) Computation: Fig. 3 depicts the internal structure of a PE. In each cycle, one operation can be processed by the ALU. Operands are loaded from the local register file (RF) or routed from a neighbouring PE, using the inputs i_1 ... i_n. For the transfer of data to neighbouring PEs each RF provides a distinct output out_l. Therefore, data can be passed without interfering with the internal computation of the ALU. The input of the RF is multiplexed so that data can be received from the surrounding system via input $live_{in}$ instead of a result from the ALU. The output $live_{out}$ is used to send local variables back when a kernel has been processed.

The ALU processes the operands according to the current context. Two types of results can be distinguished: In case of an arithmetic operation, the result is stored in the RF. In contrast, control flow operations evaluate a condition statement, which routes the signal (*status*) directly to the C-Box.

For kernels using large amounts of data, up to four PEs can feature a DMA interface. A handle and an offset or index

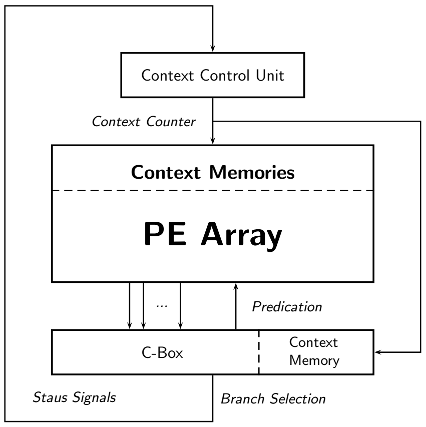

Fig. 2. Abstract overview of the CGRA structure

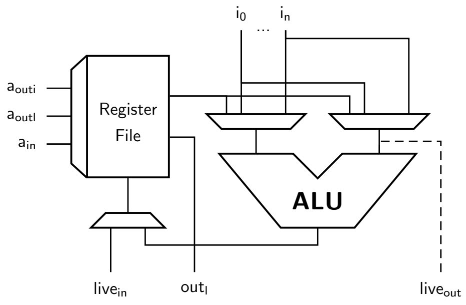

Fig. 3. Overview of a processing element

are required for memory accesses. This address information and the data are stored in separate entries of an RF. Therefore, RFs of PEs with DMA are extended by a third read port. It is needed to provide the offset or index while the output out_l provides the handle and $live_{out}$ provides the data. Furthermore, the multiplexer, which drives the inputs of the RF, provides an additional third input for incoming data from the memory.

2) Branches: The presented control flow concept uses speculation and predication. Branching is realized by the CCU performing jumps depending on a branch selection signal from the C-Box. The branch selection and predication signals are preprocessed statuses. An overview is given in Fig. 5. As mentioned, the CCU provides the CCNT which addresses all context memories. The CCNT is incremented every cycle. For branches a context memory stores an alternative CCNT (equivalent to the jump address) and the information whether the branch is conditional or unconditional. In case of an unconditional branch the CCNT adopts the provided address instead of the incremented CCNT. For conditional branches, the branch selection signal from the C-Box is used as a condition.

The C-Box (Fig. 4) processes the status bits $s_{1...n}$ from the PE array. The operation is equivalent to the logical operation in the code. Each logical operation is predefined by the scheduler and stored in the context memory of the C-Box. In order to avoid a complex network, the amount of processable incoming status bits is reduced to one per cycle. A predication or branch selection, that is processed out of multiple statuses, takes multiple cycles to evaluate. Consequently, a memory is used to store intermediate values. In case of nested loops, partial and intermediate logic functions or truth values which are stored in the condition memory are reused. Out_{ctrl} provides the branch selection signal and out_{PE} provides a predication for PEs.

```
if (x || y)
        path A;
else
        path B;
```

Listing 1.   If-else construct

$$A = x \lor y \quad (1)$$
$$B = \bar{x} \land \bar{y} \quad (2)$$

Listing. 1 illustrates an abstract code fragment of an if-else construct. The execution conditions of path A and B are shown in (1) and (2). Both conjunctions can be handled by the C-Box.

As two conditions have to be combined, the evaluation takes two cycles. In the first cycle the values of x and $\bar{x}$ are directly stored in the condition memory. Fig. 4 illustrates the functions in the second cycle. The values of x and $\bar{x}$ are loaded from the condition memory (blue and dashed blue line). Those values are combined with the incoming value of y and $\bar{y}$ (green and dashed green line) to the values of (1) (red line) and (2) (dashed red line).

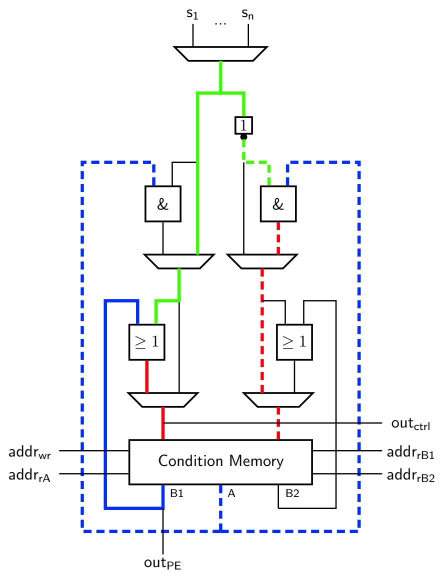

Fig. 4.   Schematic of the C-Box

With the presented network any expression can be computed. All branches are mapped onto the CGRA. Control flow dependent stores are predicated with the status evaluation of the respective branch. For this purpose, the write enable of an RF and memory write enable can optionally be dependent on the predication signal from the C-Box. Consequently, results of local variables of invalid branches are not stored in the RF and memory operations are inhibited.

3) Invocation and Communication: There are several steps required until the CGRA can start processing. We introduce the term *invocation* for the sequence of receiving local variables, executing a schedule and returning results (Fig. 6). The actual computation is called a *run*.

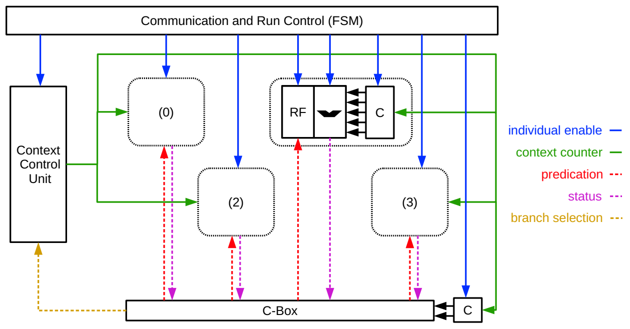

Fig. 5.   Overview of all control signals for an *composition* with 4 PEs

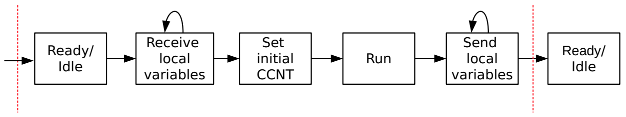

Fig. 6.   Characteristic sequence of an *invocation*

A finite state machine (Communication and Run Control) realizes the communication with AMIDAR and controls global enable signals (see Fig. 5). The first step of an *invocation* is to transfer all required local variables. The destination PE and RF slot for each local variable is determined by the scheduler. The RF address and PE identifier are transferred via tokens. The transfer (both receive and send) of local variables takes 2 cycles.

Since the context memories can potentially hold multiple schedules, it is necessary to transfer the initial CCNT of a schedule. When a schedule has finished running, the CCNT jumps to the last entry of the contexts and stays locked until it is reinitialized for a new *Run*.

B. Automated Generation

We call the infrastructure and spectrum of operations of a CGRA its *composition*. For irregular and inhomogeneous CGRAs one generic Verilog description is unreasonable regarding complexity. Therefore, we use a code-generator. As mentioned before the CGRA is embedded in AMIDAR for which a Java based simulator exists. The simulator is extended by a functional unit for the CGRA which implements a functionality to generate a Verilog representation. Fig. 7 shows the generation process.

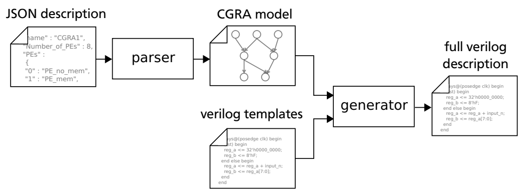

Fig. 7.   Sequence of a Verilog code-generation

```
{
"name" : "CGRA1",
"Number_of_PEs" : 4,
"PEs" :
        {
        "0" : "cgras/CGRA/WHICHEVER_PES.json",
        "1" : "cgras/CGRA/YOU.json",
        "2" : "cgras/CGRA/WANT_TO.json",
        "3" : "cgras/CGRA/INSTANTIATE.json"
        },
"Interconnect" : "cgras/CGRA/Intercon_4pe_vn.json",
"Context_memory_length" : 256,
"CBox_slots" : 32
}
```

Fig. 8. Example for a JSON description of a 4 PE *composition*

The first step of generation process is to parse a description based on a specification template in JavaScript Object Notation (JSON). Fig. 8 shows an example of a *composition* with 4 PEs. The description consists of parameters, references and operation lists. The following list includes all specifications.

- Number of PEs and a list with a path (reference) to a description for each PE
- A PE reference includes RF size and a list of available operations. Multiple different implementations of the same operation can be used (see Fig. 9). [1]
- Path to a description of the interconnect (mainly a list of available sources for each PE)
- Context size and size of the condition memory in the C-Box [2]

A CGRA Java object representation is created using the given specifications. This representation is required to emulate the CGRA in a simulation and is a model of the *composition*. The widths of control signals for elements such as multiplexers or addresses vary with the amount of neighbours or size of register files for instance. To minimize the width of control signals and consequently to minimize the width of each context, a bit-mask is created for each context.

The build hierarchy is equivalent to Fig. 5. The implementation can be divided into two categories. Firstly, there are variable structures. These refer to the modules PE, ALU and the top level module. Their implementation needs to be adapted with regard to the given *composition*. For instance, each operation is realized separately in the ALU. This cannot be done efficiently using parameters. Therefore, a template description for each module is loaded and a modular Verilog description is generated individually. Furthermore, the connection between context memories and PEs relies on the context bit-mask, using parameters for encoding. The interconnect is realized in the top level module using an array of wires. The inputs of a PE are connected by iterating over an input array in the model, that holds all possible source PEs.

```
{
        "name" : "PE_EXAMPLE",
        "Regfile_size" : 32,
        "IADD" : {"energy": 1.0,"duration": 1},
        "ISUB" : {"energy": 1.3,"duration": 1},
        "IMUL" : {"energy": 1.7,"duration": 4},
        "LADD" : {"energy": 1.4,"duration": 2},
        "IFGE" : {"energy": 1.1,"duration": 1},
        "IFLT" : {"energy": 1.1,"duration": 1},
        "NOP"  : {"energy": 0.7,"duration": 1}
}
```

Fig. 9. Example for a JSON description of a PE

Secondly, there are static structures. These refer to modules whose structural implementation does not change for different compositions. This applies to the CCU, context memory, RF and the C-Box. Structures like the multiplexer selecting one of the status signals s_1 to s_n in the C-Box or context size can be adapted using parameters, wherefore no template is needed.

V. SCHEDULER

In this section the scheduler for the CGRA architecture is described. The basic flow of information is shown in Fig. 10. The following sections explain some required preliminary steps and the scheduling in detail.

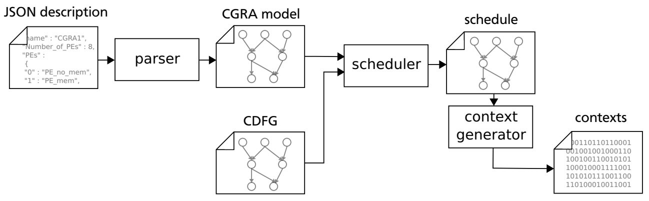

Fig. 10. Sequence of CGRA context generation

A. CDFG & Terminology

A control and data flow graph (CDFG) is used as an intermediate representation for scheduling. The CDFG is generated from candidate instruction sequences, which have been identified by the profiler[17]. These sequences are analyzed and control and data flow dependencies are extracted. This includes nested loops and conditional statements.

The CDFG is a set of nodes and edges. Nodes with no incoming edge are referred to as root nodes. A node is considered handled if its operation has completed execution before the current time step. Pending nodes are being executed in the current time step. If all predecessors of a node are already handled, this node is considered a candidate node. Since all data dependencies are fulfilled, the node may be scheduled in the current time step.

B. Predication and Speculation

The scheduler uses a mixture of speculation and predication to increase the level of parallelism. Speculation is referred to as computing operations whose results may not be needed,

[1]Currently only integer and control flow operations are supported, excluding division.

[2]The size of the memory used in the C-Box limits the maximum number of parallel branches.

because a different branch was taken. Predication in contrast means that the branch decision is computed first and then only those operations are carried out that are actually needed for the result.

No phi nodes are used to join values afterwards. Instead, predicated writes (pWRITE) are used to implement a similar behaviour. Each variable or intermediate result that is reused in another iteration or in other branches is assigned to a RF slot in one particular PE. Whenever different results for the same variable are created in different branches the right one is chosen by predicating the write operation on the PEs RF slot and all other results are automatically dismissed.

C. Nested loops

A loop graph is used to determine if a node belongs to a loop. Additionally, a set of controlling nodes (nodes producing the loop condition) tells in which cases the loop execution is terminated. These conditions are evaluated using the C-Box. A conditional jump (done by conditionally manipulating the CCNT) is used to cancel loop execution inside the CGRA. Fig. 11 shows an example CDFG.

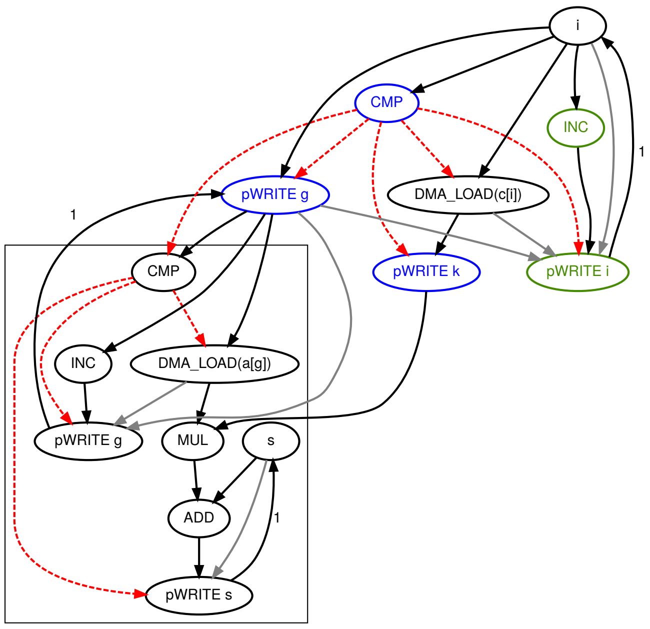

Fig. 11. CDFG with nested loops

Black and grey edges correspond to data flow and control flow respectively. Edges with a weight depict loop carried dependencies. Dashed red edges indicate that the comparison result must be false for the operation to be executed. The nodes inside the black box belong to the inner loop.

Several different cases have to be considered during scheduling. If the list of candidate nodes contains a node from an inner loop, it has to be checked if another operation of the current loop has already been scheduled in the current time step. The node is delayed if that is the case. Otherwise, it must be verified that all preceding nodes of every node in the inner loop have finished executing. If this is not the case, again, this node is delayed.

For Fig. 11 this means that all blue nodes have to be finished at least one time step before any of the nodes inside the box are scheduled.

Vice versa, if the list of candidates contains a node which belongs to an outer loop, it must be tested that all other nodes of the current loop have already been planned and finished executing. The green nodes are an example which have to be scheduled either before or after the inner loop.

D. Load, store and Constants

Loads and stores describe accesses to local variables in the host system. In case a value of a local variable changes during a CGRA *Run*, it must be written back to the host after execution. During CGRA execution changes of the value are scheduled as pWRITEs. Reads and writes refer to register accesses on the CGRA. However, values that are read but not written are treated as *pseudo-constants*.

In order to correctly handle local variables that may be written and read multiple times, the following simple heuristic is used to assign a local variable to one PE: The scheduler assigns a variable to a PE which is able to provide the value to the first PE requiring it. A write must ultimately be done on its assigned PE. This simplifies transfer and sharing of local variables. If other PEs need the value of a local variable, it is copied to these PEs. Each copy may be reused as long as no write to the variable occurred in the meantime. The scheduler keeps track of this.

Constants and pseudo-constants may be copied to multiple different PEs, depending on which PE needs these values. Since their value does not change, there is no need to store it back to the host system afterwards.

Memory accesses are realized using DMA. These operations are always predicated (both load and store) to prevent stalls due to cache misses.

E. Fusing Nodes

The term *fusing* is used to express that read and pWRITE operations may be executed together with their successor or predecessor respectively.

Each read is fused into every succeeding node. Whenever a value needs to be accessed, the scheduler handles the routing constraints and makes this value accessible to the succeeding node.

A pWRITE may be fused into the node producing the value if there are no restricting control dependencies and the operation is scheduled on the same PE as the variable. If any control flow predecessor inhibits fusing, a pWRITE is not fused and scheduled as soon as its preceding nodes are done executing.

F. Scheduling

The scheduler is based on a list scheduler in order to quickly provide schedules and the longest path weight is currently used as the priority criterion. Algorithm 1 shows the basic principle of the scheduler. The idea of this work is to extend the functionality to handle routing constraints and data locality.

A simple list scheduler picks nodes from the candidates in the order provided by the priority criterion. It then tries to find a suitable PE and schedules the operation. Simple list scheduling assumes that all data and results are accessible by any PE. Since this assumption does not hold for all CGRA compositions, the scheduler prioritizes neighbouring PEs in order to minimize overhead in data exchange. Additionally, during the plan candidates step data locality and routing constraints are resolved by copying required values to other PEs.

```
Data: CDFG, CGRA composition and priorities
Result: Schedule and contexts
begin
    t = 0;
    C = get root nodes;
    while unplanned nodes left do
        C = fuse nodes(C);
        sC = get sorted candidates(C, priorities);
        foreach candidate ∈ sC do
            check loop compatibility(candidate);
            sPE = get sorted set of PEs(candidate);
            foreach PE ∈ sPE do
                if incompatible(candidate, PE) then
                    continue;
                end
                if busy(PE) then
                    continue;
                end
                plan candidate(candidate, PE, t);
                update Attraction;
                C = remove candidate(candidate, C);
                C = add successors(candidate, C);
                break;
            end
        end
        t++;
    end
    context generation;
end
```

Algorithm 1: Scheduling algorithm

The *fuse nodes*-step has to be repeated every time step, because control dependencies may prohibit fusing.

Each node of the CDFG belongs to one specific loop. This relationship is taken into account during the *check loop compatibility*-step. Further explanation on this and an example are given in part V-C.

The *plan candidate*-step is more complex. In case operands are not accessible on the current PE their values must be transferred to this resource. This is done by copying required values if they are not directly accessible. These values are copied before the current time step if it is possible (i.e. resources are available) to prevent extension of the schedule. Otherwise, the current node is delayed until all operands are accessible.

G. Data Locality and Routing Constraints

In order to sort PEs in a meaningful way, an attraction criterion is introduced. When a node is scheduled on a PE, the attraction value for each succeeding node is increased for every PE which can access this register file (the PE itself or a neighbouring PE).

PEs are ordered by decreasing value of the attraction criterion. If two or more PEs have the same value, the PE with more connections is prioritized. This is done to simplify the resolution of routing constraints during the *plan candidate*-step.

As mentioned, during the *plan candidate* step data locality and routing constraints are taken into account. For each operand the scheduler checks if it is accessible. In case it is not accessible, the time span after finishing this operand and the current time is analyzed and the value is copied if the required resources have empty time steps.

To find the shortest path between two PEs the Floyd algorithm is used[19].

H. C-Box

During scheduling the C-Box is treated as any other resource. In the current design only one write and one read may occur at any given time. If any condition producing node is about to be planned, the scheduler checks whether any other C-Box accesses take place during the current time step.

If any conditional node is scheduled, the C-Box is used to get the corresponding condition bit. For nested branches and loops the stored condition bit is a conjunction of the outer and current condition. Speculation is implicitly implemented by using these conditions to enable or disable writes only for operations that belong to the same variable and not intermediate results.

The condition bit may be the result of multiple AND- and OR-operations. However, only one condition may be added each time step. This way the combination of input signals can always be achieved by using one stored condition, the current condition and their inverses.

I. Context generation

After a valid schedule that fits onto the CGRA is found, the contexts are produced. For both RF and C-Box allocation the left edge algorithm is used. To determine variable lifetimes the loops have to be taken into account. A value that is read in an inner loop needs an extended lifetime until the end of that loop. The same holds for the lifetimes of condition bits.

VI. Evaluation

A. ADPCM Decode as Application Example

We will evaluate our work with an ADPCM decoder, which was also used in [20]. The code consists of a large while loop and contains several nested loops. Some of them are executed under certain conditions, dependent on the input data, while some nested loops contain conditional code in the loop body. Fig. 12 shows the control flow of the ADPCM decoder. The black dots mark branch or merge points respectively. Grey boxes represent loops while the solid and dashed lines denote conditional branches. With the help of the C-Box it is possible to map the whole decoder on a CGRA. When the ADPCM

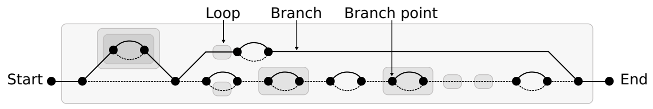

Fig. 12. Control flow of the ADPCM decoder

decoder is executed purely on AMIDAR, it takes 926 k cycles.

Note that the main goal of this evaluation is to show the functionality of our framework rather than comparing the performance to other approaches or finding an optimal CGRA composition for the given application. Thus, arbitrary CGRA compositions were chosen to show the ability of our framework to handle inhomogeneous and irregular CGRAs.

B. Homogeneous CGRA Structures

At first we will evaluate the proposed scheduler for homogeneous and regular CGRA compositions shown in Fig. 13. Supported operations are 32 bit logic operations, addition, subtraction and multiplication. The multiplication was realized as a two clock cycle block multiplication. All compositions have a context size of 256 and use an RF size of 128.

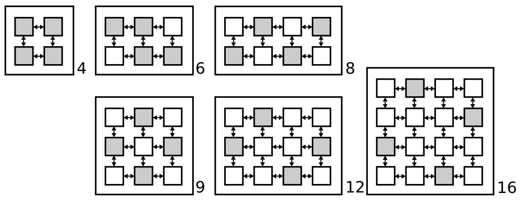

Fig. 13. Evaluated CGRAs with mesh architecture. Grey PEs have DMA

Table II shows the execution times for the ADPCM decoder with an input vector of 416 samples. The time is given in clock cycles for both AMIDAR and the CGRAs evaluated in Part A. A maximum unroll factor of 2 for inner loops was used. The table also shows the synthesis results when mapping the CGRA on a Virtex 7 XC7VX690 FPGA. We used the default settings for the synthesis and implementation in Vivado.

It can be seen that the CGRA with 9 PEs performs best and is 7.3 times faster than the AMIDAR processor[3]. Further investigation leads to the conclusion that the execution time does not only depend on the number of PEs but also on the interconnect and the choice of PEs with DMA as we will see in the following section.

Two effects overlap: More PEs can speed up the application as more instructions can be executed concurrently. Vice versa, more PEs also result in longer paths between source and sink of data such that more time is consumed while transferring data. The length of the paths is highly dependent on the interconnect and the position of PEs with DMA.

Table I shows that the same holds for the number of contexts and register file entries used, when mapping the ADPCM decoder to all CGRAs. It can also be seen that the number of contexts does not directly correspond to the execution time as these contexts also include the inner loops whose execution time is highly relevant for the total execution time.

[3]Note that AMIDAR requires about twice as many clock cycles as an Intel® Core^{TM}2 Duo [1]

TABLE I. MEMORY UTILIZATION OF THE ADPCM DECODER SCHEDULES FOR ALL CGRAS

	4 PEs	6 PEs	8 PEs	9 PEs	12 PEs	16 PEs
Used Contexts	200	191	189	175	173	168
Max. RF entries	66	69	62	51	44	49

The generated Verilog descriptions are synthesizeable and result in low utilization and moderate timing. The utilization increases with the number of PEs approximately in a linear fashion. Since there is currently no support for floating point arithmetic, the utilization is fairly low, as expected. The utilization benefits from the efficient use of BRAMs for the context memories and the use of bit-masks for the contexts.

The clock frequency is moderate, but the design holds significant potential for optimization. An RF width of 128, which is required due to limitations of the current scheduler, leads to significant slowdown. An alternative composition of 4PE using 32 entries shows an increase of 7.2 % (111.1 MHz) in clock frequency. Despite a lean Verilog description, the templates are still unoptimized. Moreover, the use of homogeneous structures leads to a dense and inefficient data path.

1) Influence of Multiplier Implementation: Table III shows the execution for the ADPCM decoder when using CGRAs with multipliers which take only one cycle to perform a multiplication. As expected for most CGRAs the number of cycles decreases compared to the block multiplier implementation where the multiplication takes two cycles. The only exception is the CGRA with 16 PEs. Table III also shows the maximum frequencies when mapping those CGRAs on Virtex 7 XC7VX690 FPGA. Taking these results into account, the execution times in seconds can be calculated for all CGRAs with both multiplier implementations. The results can be seen in Table IV. Due to higher clock frequencies for CGRAs with block multipliers, the execution time is shorter in that case.

TABLE III. EXECUTION TIMES FOR DIFFERENT CGRAS WITH SINGLE CYCLE MULTIPLIERS IN CLOCK CYCLES

	4 PEs	6 PEs	8 PEs	9 PEs	12 PEs	16 PEs
Cycles	147.0k	131.4k	134.9k	125.6k	133.1k	143.1k
Frequency in MHz	86.9	84.0	81.3	79.7	79.0	76.3

TABLE IV. ADPCM DECODE EXECUTION TIMES IN MILLISECONDS

	4 PEs	6 PEs	8 PEs	9 PEs	12 PEs	16 PEs
Single cycle multiplier	1.69	1.56	1.66	1.58	1.68	1.88
Dual cycle multiplier	1.48	1.36	1.40	1.35	1.54	1.61

C. Inhomogeneous and Irregular CGRA Structures

Fig. 14 shows CGRA compositions with irregular interconnect. The CGRAs A to E have the same operational spectrum as the CGRAs evaluated in Part B whereas in CGRA F only the black PEs support multiplication. The results are also shown in Table II.

It can be seen that the fastest composition is CGRA D while CGRA B performs worst because little interconnect is available. The highly inhomogeneous and irregular CGRA F is only marginally slower in terms of clock cycles than CGRA

TABLE II. Execution times for different CGRAs in clock cycles

	4 PEs	6 PEs	8 PEs	9 PEs	12 PEs	16 PEs	8 PEs A	8 PEs B	8 PEs C	8 PEs D	8 PEs E	8 PEs F
Execution time / cycles	152.3k	135.3k	137.5k	126.6k	135.3k	140.1k	147.6k	157.7k	133.9k	133.8k	150.4k	134.4k
Frequency (MHz)	103.6	99.5	98.0	93.6	88.1	86.9	94.8	93.6	100.4	96.0	94.3	93.5
LUT - logic (% util.)	1.01	1.49	1.89	2.22	2.80	3.61	1.92	1.87	1.91	1.88	1.90	1.80
LUT - memory (% util.)	0.61	0.81	1.01	1.11	1.41	1.82	0.91	0.91	1.01	1.01	1.01	1.01
DSP (% util.)	0.33	0.50	0.67	0.75	1.00	1.33	0.67	0.67	0.67	0.67	0.67	0.17
BRAM (% util.)	0.34	0.48	0.61	0.68	0.88	1.16	0.61	0.61	0.61	0.61	0.61	0.61

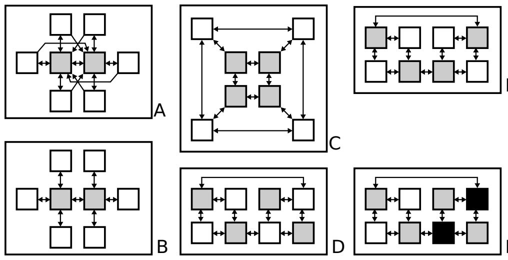

Fig. 14. Irregular and inhomogeneous compositions. Grey PEs have DMA

D but the utilization of DSPs decreases by 75 %. It can be seen that supporting irregular and inhomogeneous structures can potentially save area on the chip and most likely energy.

Apart from that other forms of inhomogeneity are possible where for example only one PE supports floating point arithmetic and all others only support integer arithmetic.

For the ADPCM decoder the scheduling and context generation takes at most 3.1 s on a Intel® Core^{TM} i7-6700 with 3.4 GHz.

VII. Conclusion

In this contribution we have shown a tool set for the generation and programming of CGRAs. The generated CGRA instances are capable to embed control flow. Thus, the scope of mappable kernels is highly increased. The generated instances can all be programmed with our scheduler. Generator and scheduler can operate on inhomogeneous and irregular CGRA compositions. This opens up great potential to save resources and energy by tailoring the CGRA to a specific application domain. We have demonstrated the mapping abilities of the scheduler with a complex sample application. We have achieved a speedup of more than 7 compared to the AMIDAR baseline. No modification of the application was required to reach this acceleration. We have seen other applications with even higher speedup, but we chose the ADPCM decoder since it better demonstrates the ability to map nested loops and control flow onto the CGRA.

Currently, we are improving the library of elements from which the PEs are composed. Also, several optimizations regarding the introduction of further pipeline stages in the PEs are investigated.

In the future, we want to improve the scheduler to employ modulo scheduling. Also, we want to develop a tool that automatically analyzes a set of problems from an application domain and generates a matching CGRA composition.

References

[1] S. Döbrich and C. Hochberger, "Exploring online synthesis for CGRAs with specialized operator sets," *International Journal of Reconfigurable Computing*, vol. 2011, p. 10, 2011.

[2] W. Kim, D. Yoo, H. Park, and M. Ahn, "Scc based modulo scheduling for coarse-grained reconfigurable processors," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 321–328.

[3] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1284–1291.

[4] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 166–176.

[5] S. RajendranRadhika, A. Shrivastava, and M. Hamzeh, "Path selection based acceleration of conditionals in cgras," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 121–126.

[6] S. Yin, D. Liu, L. Liu, S. Wei, and Y. Guo, "Joint affine transformation and loop pipelining for mapping nested loop on cgras," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 115–120.

[7] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on cgras," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, June 2014, pp. 1–6.

[8] K. Han, K. Choi, and J. Lee, "Compiling control-intensive loops for cgras with state-based full predication," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1579–1582.

[9] N. Paulino, J. a. C. Ferreira, J. a. Bispo, and J. a. M. P. Cardoso, "Transparent acceleration of program execution using reconfigurable hardware," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1066–1071.

[10] C.-H. Ho, V. Govindaraju, T. Nowatzki, Z. Marzec, P. Agarwal, C. Frericks, R. Cofell, J. Benson, and K. Sankaralingam, "Performance evaluation of a dyser FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation," *Energy (mJ)*, vol. 5, no. 10, p. 15, 2015.

[11] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, "A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, July 2013, pp. 188–195.

[12] L. Chen, J. Tarango, T. Mitra, and P. Brisk, "A just-in-time customizable processor," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, Nov 2013, pp. 524–531.

[13] Z. E. Rákossy, A. A. Aponte, T. Noll, R. Leupers, G. Ascheid, and A. Chattopadhyay, "Design and synthesis of reconfigurable control-flow structures for cgra," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, Dec 2015, pp. 1–7.

[14] K. Han, J. Ahn, and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on cgra," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 2, pp. 8:1–8:25, May 2013.

[15] D. Lee, M. Jo, K. Han, and K. Choi, "Flora: Coarse-grained reconfigurable architecture with floating-point operation capability," in *Field-*

Programmable Technology, 2009. FPT 2009. International Conference on, Dec 2009, pp. 376–379.

[16] S. Gatzka and C. Hochberger, “The AMIDAR class of reconfigurable processors,” *The Journal of Supercomputing*, vol. 32, no. 2, pp. 163–181, 2005.

[17] ——, “Hardware based online profiling in AMIDAR processors,” in *IPDPS*, 2005, p. 144b.

[18] L. J. Jung and C. Hochberger, “Feasibility of high level compiler optimizations in online synthesis,” in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, Dec 2015, pp. 1–7.

[19] R. W. Floyd, “Algorithm 97: Shortest path,” *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[20] C. Hochberger, L. J. Jung, A. Engel, and A. Koch, “Synthilation: JIT-Compilation of Microinstruction Sequences in AMIDAR Processors,” in *DASIP*, 2014, pp. 193–198.