Partial Online-Synthesis for Mixed-Grained Reconfigurable Architectures

Artjom Grudnitsky, Lars Bauer, and Jörg Henkel
Karlsruhe Institute of Technology (KIT), Department of Computer Science, Chair for Embedded Systems
{grudnitsky, lars.bauer, henkel}@kit.edu

Processor architectures with Fine-Grained Reconfigurable Accelerators (FGRAs) allow for a high degree of adaptivity to address varying application requirements. When processing computation intensive kernels, multiple FGRAs may be used to execute a complex function. In order to exploit the adaptivity of a fine-grained reconfigurable fabric, a runtime system should decide when and which FGRAs to reconfigure with respect to application requirements. To enable this adaptivity, a flexible infrastructure is required that allows combining FGRAs to execute complex functions.

We propose a mixed-grained reconfigurable architecture composed from a Coarse-Grained Reconfigurable Infrastructure (CGRI) that connects the FGRAs. At runtime we synthesize CGRI configurations that depend on decisions of the runtime system, e.g. *which* FGRAs shall be reconfigured. Synthesis and place & route of the FGRAs are done at compile time for performance reasons. Combined, this results in a partial online synthesis for mixed-grained reconfigurable architectures, which allows maintaining a low runtime overhead while exploiting the inherent adaptivity of the reconfigurable fabric. In this work we focus on the crucial parts of synthesizing the configurations for the CGRI at runtime, propose algorithms, and compare their performance/overhead trade-offs for different application scenarios.

We are the first to exploit the increased adaptivity of FGRAs that are connected by a CGRI, by using our partial online synthesis. In comparison to a state-of-the-art reconfigurable architecture that synthesizes the configurations for the CGRI at compile time we obtain an average speedup of 1.79x.

I. Introduction & Related Work

Complex embedded applications like H.264 video encoding often consist of multiple computation intensive kernels. Their computation requirements can be addressed by employing a reconfigurable fabric to implement fine-grained reconfigurable accelerators (FGRAs) in an embedded field-programmable gate array (eFPGA) [1,2]. In the remainder of this paper we focus on reconfigurable architectures that comprise a standard (fixed) processor pipeline plus a reconfigurable fabric (eFPGA) to implement FGRAs.

A characteristic trait of reconfigurable fabrics is their *granularity* [3]. Fine-grained reconfiguration is based on lookup tables and is used to implement arbitrary logic like FGRAs. Architectures using FGRAs include MOLEN [4] or XiSystem [5]. The main disadvantage of fine-grained reconfiguration is the considerable amount of time to perform a reconfiguration that can take up to several milliseconds (0.6-0.7 ms for the FGRAs on our prototype).

Coarse-grained reconfigurable architectures such as Montium [6], ADRES [7], or XPP [8] use an array of processing elements (PEs) connected by a coarse-grained reconfigurable infrastructure (CGRI). The PEs perform operations (e.g. additions or multiplications) on (sub-)word level and the CGRI allows to connect several PEs to implement dataflow graphs (DFGs). The amount of configuration data to configure PEs and CGRI is rather small (e.g. with 5 bits a PE can be configured to perform 1 out of 32 different operations) and thus a new configuration can be established in a very short time (e.g. in a single cycle for Montium [6]). Unlike FGRA reconfiguration, switching the CGRI into a new configuration does not require loading a new bitstream (in fact, the CGRI can be implemented as an ASIC), thus resulting in the short configuration time. However, using coarse-grained reconfiguration to implement bit/byte operations or state machines etc. leads to an inefficient use of resources. Bit/byte operations (e.g. changing the sequence of bits in a word) can be implemented by using several PEs that perform shift- and logical operations. A fine-grained reconfigurable fabric can directly implement such operations in a single FGRA, thereby also providing an improved performance compared to a coarse-grained implementation, as the operation can usually be completed in one cycle in an FGRA. As both granularities have advantages and disadvantages [3], we employ a *mixed-grained* reconfigurable architecture, featuring fine-grained and coarse-grained reconfiguration to exploit their respective advantages. Figure 1 shows the mixed-grained target architecture of this work, based on [9]. FGRAs are reconfigured onto so-called *containers* and communication between the containers is established by the CGRI [10]. The CGRI is composed of an array of *connectors* (one per container) that are connected by *links*.

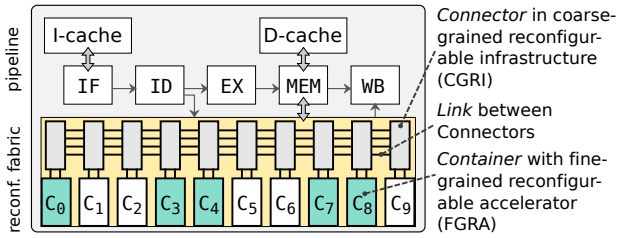

Fig. 1: Architecture of a runtime-reconfigurable processor

Mixed-grained architectures are also used by the RISPP [11] and KAHRISMA [12] projects. In addition to the FGRA and CGRI, KAHRISMA also employs coarse-grained reconfigurable PEs. Both projects propose runtime systems that perform parts of the reconfiguration decisions [11,13]. Their focus is on high-level runtime decisions such as the amount of FGRAs that shall be reconfigured and the sequence in which they shall be reconfigured onto the fabric. Methods to deal with disadvantages of fine-grained reconfiguration, i.e. long reconfiguration time, have been investigated in both projects, e.g. in [11]. This is especially important for applications whose behavior heavily depends on input data (e.g. an H.264 video encoder has different computational demands when encoding a slow changing video frame sequence in comparison to a hectic scene with many movements). However, they do not address the details of *'where'* to load FGRAs on the fabric (into which container) and *'how'* to connect them, i.e. they do not determine the configuration of the CGRI. This configuration is required to process complex kernels that demand more than one FGRA and it cannot be created at compile time as it depends on runtime decisions (demonstrated in Section II-D). The already existing high-level runtime systems of both systems are orthogonal to the low-level fabric configuration decisions addressed in this work, i.e. this work can be integrated into the runtime systems of the aforementioned projects.

Approaches on placement of tasks on reconfigurable fabrics have been explored. [14] presents an offline approach using ILP and [15] presents an online approach for 2D placement, which however does not regard communication-induced constraints between placed modules. An online synthesis approach for a fine-grained reconfigurable fabric is proposed in [16], where FGRAs are synthesized at runtime. However,

978-3-9810801-8-6/DATE12/©2012 EDAA

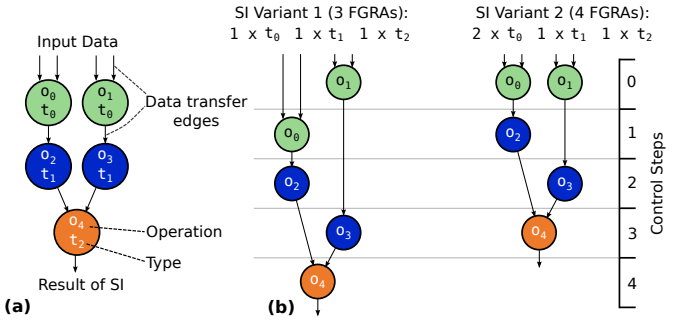

Fig. 2: **(a)** Dataflow Graph (DFG) of a Special Instruction (SI) **(b)** Different combinations of available FGRAs (i.e. FGRAs that are reconfigured into a container at a certain time) result in SI variants that differ in their execution latency

the high overhead of synthesizing FGRAs reduces the performance in case of frequently changing application requirements (e.g. input-data dependent functions in a video encoder) as they would lead to frequent re-synthesis. Therefore, we synthesize the FGRAs at compile time and focus on synthesizing the configuration of the CGRI at runtime. The authors of [17] target a coarse-grained reconfigurable fabric and perform online synthesis of the CGRI. However, as they do not use FGRAs (and thus do not benefit from their performance), they do not address placement of FGRAs and binding of operations to FGRAs at runtime (details explained in next section).

The paper organization is as follows: in Section II we provide a brief overview of how the FGRAs are invoked by an application and how the CGRI is used to connect them. After that, we explain the synthesis problem that needs to be solved, state the novel contribution of our partial online-synthesis, and provide a motivational case study why it cannot be performed at compile time. The concepts of *hazards* and how we address them is explained in Section III before presenting and discussing our solutions for placement and binding in Sections IV and V, respectively. Experimental results and overhead analysis are presented and discussed in Section VI and Section VII concludes the paper.

II. Definitions, Problem Statement, and Novel Contribution

A. Special Instructions

Special Instructions (SIs) are extensions to the instruction set architecture of a standard processor pipeline (see Figure 1). They serve as the interface between applications and the reconfigurable fabric and access their inputs and outputs from main memory or the register file. Computation intensive kernels are common candidates for acceleration via SIs. The application programmer invokes SIs in the application (e.g. using inline assembly code) and the runtime system of the reconfigurable processor controls the reconfigurations of the FGRAs. The actions taken by the runtime system to create a datapath of FGRAs on the fabric from an SI invocation, are described in the remainder of this chapter.

An SI is described by an acyclic data-flow graph (DFG) as shown in Figure 2a. The DFG consists of $|V|$ *operations* $o_i, 0 \leq i < |V|$ of $T \leq |V|$ different *types* $t_k = t(o_i), 0 \leq k < T$ and directed *data transfers* $d_{i,j}$ between nodes o_i and o_j. There shall be $|A|$ FGRAs $a_m, 0 \leq m < |A|$ available to the system. Each operation o_i is implemented by an FGRA a_m of the same type $t(o_i) = t(a_m)$. The complexity of an operation exceeds the complexity of a simple arithmetic or logical function. An FGRA can perform transformation, coding, or filter operations on its input data. The amount of cycles to complete an operation o_i is annotated in its type $t(o_i)$ (e.g. operations that fetch data from main memory require at least 2 cycles). For example, to execute a 2D discrete cosine transformation (DCT), a horizontal as well as a vertical transformation is required. The SI input data is sent to FGRAs for transformations (performing horizontal DCT), that output is sent to another type of FGRAs that rearrange the data (because vertical and horizontal DCT have different access patterns), and that output is send back to the FGRAs for transformation (performing vertical DCT) before storing the results.

An SI exists in different *SI variants* that differ in the number of FGRAs that are used to implement the DFG. Figure 2b shows an example for two different SI variants for the DFG in Figure 2a. The quantity of FGRAs of a certain type t_k that are used to implement an SI variant is denoted as $q(t_k) \in \mathbb{N}$. It affects the number of *control steps* that are required to execute the SI variant (and its execution latency in cycles) as shown in Figure 2b.

B. Partial Online-Synthesis

In this paper, we focus on synthesizing the configuration data for the CGRI. Synthesizing the FGRAs and creating the SI DFGs is done at compile time. The steps that are required to synthesize the DFG into a CGRI configuration are: Allocation, Scheduling, Placement, and Binding as described in the following.

The *allocation* decides the quantity $q(t_k)$ of FGRAs of a certain type that shall be reconfigured onto the fine-grained reconfigurable fabric. This determines which SI variant is used to implement an SI (see Figure 2). Both RISPP [11] and KAHRISMA [12] propose different algorithms for this problem. Our placement and binding techniques are independent of the allocation scheme, thus in this work we use the RISPP allocation strategy [11]. The *Scheduling* of the DFG for a given allocation assigns operation o_i to *control step* $cs_l = cs(o_i)$ such that (i) in each control step not more FGRAs are required than assigned by the allocation (see Eq. 1), (ii) the data dependencies between the operations are considered (see Eq. 2), and (iii) the number of control steps (SI execution latency) is minimal. SI Scheduling is done offline using high-level synthesis scheduling algorithms such as *LIST* or *Force-directed Scheduling* [18].

$$\forall cs_l \, \forall t_k : \left| \sum_{o_i, t(o_i)=t_k, cs(o_i)=cs_l} 1 \right| \leq q(t_k) \tag{1}$$

$$\forall d_{i,j} : cs(o_i) < cs(o_j) \tag{2}$$

Placement of an FGRA a_m decides into which container (see Figure 1) $c_p = c(a_m)$ it shall be reconfigured. This depends on the allocation and thus needs to be performed at runtime as well. The placement has to ensure that different FGRAs (potentially of the same type) are placed in different containers (see Eq. 3).

$$\forall a_m, a_n : m \neq n \Rightarrow c(a_m) \neq c(a_n) \tag{3}$$

Finally, *Binding* needs to bind components of the DFG (operations o_i and data transfers $d_{i,j}$) to the hardware (containers and links). As it depends on the placement it also needs to be performed at runtime. The binding $b(o_i) = c_p$ assigns operation o_i to a container c_p such that the types match (see Eq. 4) and two operations in the same control step are not assigned to the same container (see Eq. 5). The connectors (see Figure 1) provide a small amount of local memory (8 words in our prototype) for results of operations. An operation o_i that executes on container c_p stores its results to the local memory of the connector that is connected to c_p. The binding needs to decide to which address (in a connector's memory) a result shall be written without overwriting results that are still needed in a later control step. Therefore, during compile-time, a lookup table is created that describes for all operations o_i the control step $cs(o_i)$ in which they create a result and after which control step $cs_l = max\{cs(o_j) : \exists d_{i,j}\}$ it is no longer required. This table is used for binding operations to containers at runtime to annotate memory addresses as occupied or as free.

$$b(o_i) = c_p = c(a_m) \Rightarrow t(o_i) = t(a_m) \tag{4}$$

$$\forall o_i, o_j, cs(o_i) = cs(o_j) : i \neq j \Rightarrow b(o_i) \neq b(o_j) \tag{5}$$

When executing an operation o_j that demands input data from another operation o_i the data transfer $d_{i,j}$ is performed in control

step $cs(o_j)$ and needs to be bound to a *communication segment*, i.e. a connected set of links. The architecture shown in Figure 1 uses four links l_0-l_3 between neighbored containers. The communication segment to establish $d_{i,j}$ uses a particular link l_s for the interval $[b(o_i), b(o_j)]$. That means that the link l_s between the connector at container $b(o_i)$ and $b(o_i)+1$ is occupied (assuming that $b(o_i) < b(o_j)$) and l_s between $b(o_i)+1$ and $b(o_i)+2$ is occupied etc. However, the links l_s before $b(o_i)$ and after $b(o_j)$ are still free. One link l_s can be used to realize multiple data transfers $d_{i,j}, d_{x,y}, i \neq x$ or $j \neq y$ in the same control step if their intervals do not overlap (see Eq. 6). If they overlap, then different links $l_r, l_s, r \neq s$ need to be used or $d_{i,j}$ and $d_{x,y}$ need to be performed in different control steps.

$$[b(o_i), b(o_j)] \cap [b(o_x), b(o_y)] = \oslash \tag{6}$$

After binding operations to containers, the decisions which memory address shall be used for a result and which links shall be used for a data transfer is trivial. When multiple memory addresses are free, then it does not affect the execution latency of the SI variant which one is used (however, binding of operations to containers has to assure that sufficient free memory addresses are available). When multiple links are free between two communicating containers, then it is not important which one is used (however, if no link is available the system has to find a solution, as explained in Section III). Therefore, according to the binding we focus on binding operations to containers in the following and use *First Fit* strategies for binding results to memory and data transfers to links.

After placement and binding are completed for an SI variant, the detailed information which operation shall execute on which container, which data transfer shall use which link etc. directly determines a specific configuration for the CGRI for each cycle of the SI execution. This configuration is stored in an on-chip configuration memory. In our prototype, we use 1024-bit configuration data for the CGRI and need between 40 and 50 KByte per FGRA (due to the size of the FGRA configurations, they have to be stored in the off-chip DRAM memory). When an SI is invoked, the CGRI is reconfigured cycle-by-cycle to control the SI execution.

C. Novel Contribution

The challenges for placement and binding are that their decisions affect the execution latency of SI variants. In the best case, each control step is executed in one cycle, but depending on placement and binding, a control steps might require multiple cycles (details and examples in Section III), thus delaying the SI variant execution latency. As the SIs are designed to accelerate computation intensive kernels, delaying their execution directly affects the application's performance. Therefore, the **novel contributions** of our partial online synthesis address the crucial details of

1) *Placing FGRAs*: we propose and evaluate novel *Cluster-* and *Connectivity* placement algorithms.
2) *Binding Operations*: we propose three different online binding algorithms and evaluate their performance and overhead in various scenarios.

D. Motivational Example

The number of FGRAs of a certain type that shall be reconfigured is decided at runtime to adapt to different application requirements, as described in Section I. The number of different *SI schedules* (i.e. determining which FGRA is used in which cycle) is limited and thus it is possible to create the schedules offline and store them in memory for the runtime system. However, the number of different FGRA placements is huge, as it depends on (i) which FGRAs are already configured, (ii) which may be replaced, and (iii) which shall be reconfigured. When n containers are available and an SI demands m different FGRA types with $q(a_k)$ FGRAs per type, then the number of different placement possibilities (after writing down the binomial coefficients, bringing them into their factorial form and reducing the terms) is shown in Eq. 7.

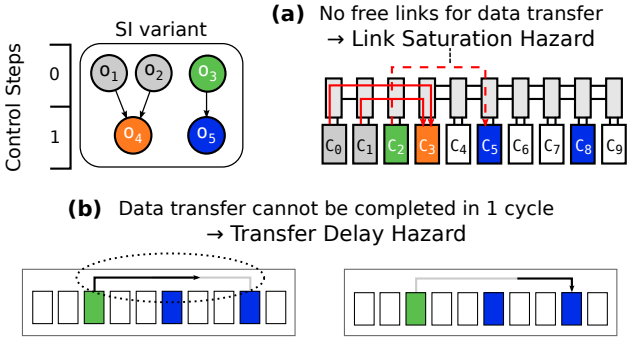

Fig. 3: Examples for Link Saturation Hazard (LSH) and Transfer Delay Hazard (TDH)

$$\frac{n!}{\prod_{k=0}^{m-1}(q(a_k)!) * \left(n - \sum_{k=0}^{m-1} q(a_k)\right)!} \tag{7}$$

To give an example, our prototype has 10 containers and uses 1024 bit per cycle to configure the CGRI. Some SIs use up to four different FGRA types, e.g. the *Sum of Absolute Transformed Differences* (SATD) from an H.264 video encoder. When the quantities for the four FGRA types are (1,1,1,1), then there are 5,040 different placements. For (1,1,2,2) and (1,2,2,2) there are already 37,800 and 75,600 different placements, respectively. The SATD execution latency for these three SI variants are 23, 21, and 19 cycle. Thus, 14.2, 96.9, and 175.3 MB are required to store the CGRI configuration data for all placements of the three SI variants. When 12 containers are available then 33.1, 426.3, and 1157.3 MB are required, respectively. Additionally, the configuration data for further SI variants and further SIs needs to be stored. That clearly shows that preparing and storing all placement combinations at compile time for all SI variants of all applications that shall execute is practically infeasible and has to be synthesized at runtime instead.

III. Solving Saturation- and Delay Hazards

While ideally a data transfer between two FGRAs should be completed in one cycle, it may take longer depending on placement and binding. Before presenting our algorithms for placement and binding that aim at reducing these delays, we explain the *Link Saturation Hazard* (LSH) and the *Transfer Delays Hazard* (TDH) that may cause these delays and we explain how they are handled in our approach.

An example for an LSH is shown in Figure 3a (only two links per connector are used for simplicity). It occurs when the binder tries to reserve a link for $d_{2,4}$ in control step $cs(o_4) = cs_1$ but there is no free link between $b(o_2) = c_2$ and $b(o_4) = c_5$, i.e. all links between these containers are already used for other data transfers in cs_1. The binder resolves this hazard by splitting cs_1 into two cycles $cs_{1,0}$ and $cs_{1,1}$ and binding o_4 to $cs_{1,1}$. This method is applied iteratively until all LSHs have been resolved, i.e. if more LSHs exist in $cs_{1,0}$, then the corresponding operations are also bound to $cs_{1,1}$ and if afterwards LSHs exist in $cs_{1,1}$, then another cycle $cs_{1,2}$ is introduced.

A TDH occurs when for a data transfer $d_{i,j}$ the *distance* of the containers $b(o_i)$ and $b(o_j)$ (i.e. $|b(o_i) - b(o_j)|$) is too long to be traversed in one cycle (see Figure 3b). In such a case, the control step is split into two cycles, as described earlier. The maximum *distance* D that can be traversed in one cycle depends on the technology and operation frequency and we evaluate different values for D in the results. The required number of cycles to complete a data transfer $d_{i,j}$ is $\lceil |b(o_i) - b(o_j)| / D \rceil$.

The occurrences of LSHs and TDHs depend on the placement and binding decisions. When FGRAs are placed near to each other, then no TDH occurs when they need to communicate. In the following we

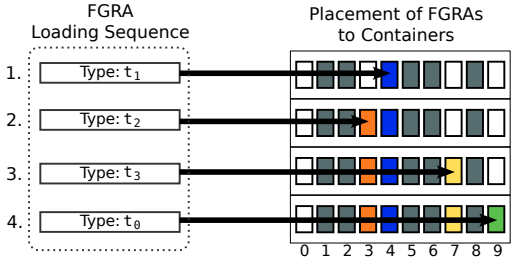

Fig. 4: Example for placement of FGRAs

present our placement and binding algorithms that aim at reducing these hazards.

IV. Placing FGRAs

The placement of an FGRA to a container can affect the execution latency of all SIs that use this FGRA (see Section III). The goal of our placement algorithm is to minimize the cycle loss due to Transfer Delay Hazards, i.e. it needs to consider the data transfers between operations during an SI execution. At runtime, the high-level runtime system determines the amount of FGRAs that shall be reconfigured to accelerate a computation intensive kernel (see Section I) and provides the following information to the placement algorithm: (i) the current configuration of all containers C, (ii) a set of SI variants S that are used in that kernel, (iii) an FGRA type t_R that is configured but that is not required in the kernel and thus can be replaced, and (iv) an FGRA type t_P that needs to be reconfigured and shall be placed.

We propose *Cluster Placement*, which aims at placing all FGRAs of an SI variant $s \in S$ closely together. Let A be the set of FGRAs that are required to implement an SI variant s. A_L shall denote the leftmost container and A_R the rightmost container that contains an FGRA $\in A$, i.e. $A_L = min\{c(a_m) : a_m \in A\}$ and $A_R = max\{c(a_m) : a_m \in A\}$. Out of all possible data transfers that could be performed by s, no one is longer than the *distance* between A_L and A_R, i.e. $|A_L - A_R|$ which we call the *expansiveness* of SI variant s. For example, Figure 4 shows a sequence of FGRAs that shall be placed to implement an SI variant. A_L is c_3 and A_R is c_9, thus the expansiveness of this SI variant is 6. The Cluster Placement iterates over all SI variants s and all container candidates c_p that match the replacement type t_R or are empty and examines the expansiveness of s if the FGRA would be placed to c_p. The container candidate that results in the smallest expansiveness is selected. If multiple candidates result in the same expansiveness, then the first candidate with that value is selected. The complexity of the Cluster Placement is $\mathcal{O}\left(|containers| * |SI\,variants|\right)$.

The main drawback of Cluster Placement is that it does not consider *how often* a data transfer between two containers occurs, e.g. there might be no data transfer between A_L and A_R at all in s. For instance, let us assume that the most frequent data transfer in Figure 4 is between FGRA type t_0 and t_2. In this case it is better to place the FGRA with type t_0 to container c_0. This leads to a larger expansiveness (thus it would be avoided by the Cluster Placement), but provides a reduced SI execution latency, as the delay for the most frequent data transfer is minimized.

Our proposed *Connectivity Placement* algorithm is shown in Algorithm 1 and addresses this problem by performing a communication-aware placement. During compile-time, all SI variants s are annotated with the *connectivity* between any two FGRA types t_x and t_y as shown in Eq. 8, i.e. the number of data transfers $d_{i,j}$ between operations of these types are enumerated.

$$s.connectivity[t_x, t_y] = \sum_{d_{i,j}, t(o_i)=t_x, t(o_j)=t_y} 1 \qquad (8)$$

The Connectivity Placement iterates over all container candidates c_p that match the replacement type t_R or are empty and examines the connectivity for all SI variants $s \in S$ that require the FGRA type t_P. The distance between c_p and all other containers c_q is weighted with the compile-time prepared connectivity value of s and summed up. The algorithm selects the container c_T with the least total connectivity score. The complexity of the Connectivity Placement is $\mathcal{O}\left(|containers|^2 * |SI\,variants|\right)$.

Algorithm 1 Connectivity Placement

INPUT: Set of containers C with the configured FGRAs $a_m = C[c_p]$, set of SI variants S, replacement FGRA type t_R, new FGRA type that shall be placed t_P
OUTPUT: Target container c_T

```
1: c_T := −1, best_connectivity := MAX_INT
2: for each container c_p ∈ C do
3:   if C[c_p] ≠ NULL and t(C[c_p]) ≠ t_R then continue
4:   total_connectivity := 0
5:   for each SI variant s ∈ S do
6:     if not s.requires(t_P) then continue
7:     for each container c_q ∈ C do
8:       if C[c_q] = NULL then continue
9:       total_connectivity := total_connectivity +
         s.connectivity[t_P, t(C[c_q])] * distance(c_p, c_q)
10:    end for
11:  end for
12:  if c_T = −1 or total_connectivity < best_connectivity then
13:    c_T := c_p
14:    best_connectivity := total_connectivity
15:  end if
16: end for
17: return c_T
```

V. Binding Operations

After the placement of FGRAs is decided, the so-called *fabric configuration* is known, i.e. the information which container c_p contains which FGRA $a_m, c_p = c(a_m)$. Before an SI variant can be executed, it has to be bound to the fabric configuration as explained in Section II-B. The control steps of an SI variant are bound one after each other.

To bind an operation o_j from the control step cs_l to a container c_p, a list of *container candidates* is constructed such that (i) the types match (see Eq. 4), (ii) no other operation is bound to c_p in this control step (see Eq. 5), and (iii) the connector that connects c_p with the other containers has sufficient free local memory to store the results of o_j (see Section II-B). When the fabric configuration contains only one container with the correct type, then binding is trivial. When multiple such containers exist, then the binding algorithm has a choice. For instance, in Figure 3 operation o_1 can be bound to containers c_5 or c_8, where binding to c_8 leads to a Transfer Delay Hazard. We propose the following operation binding strategies:

First Fit Binding (FFB) scans the containers from left to right and returns the first valid container candidate that fulfills the constraints. The complexity is $\mathcal{O}\left(|containers|\right)$.

Communication-Aware Binding (CAB) considers the input data that needs to be transferred from other operations o_i to o_j. These operations are already bound to containers as they execute in earlier control steps. All data transfers $d_{i,j}$ are examined for Link Saturation Hazards and Transfer Delay Hazards. For all container candidates c_p the number of cycles are calculated that are required to resolve the hazards, i.e. CAB performs a partial communication binding to evaluate the container candidates. The container that results in the least number of cycles to resolve the hazards is selected. The complexity is $\mathcal{O}\left(|containers| * |data\,transfers|\right)$.

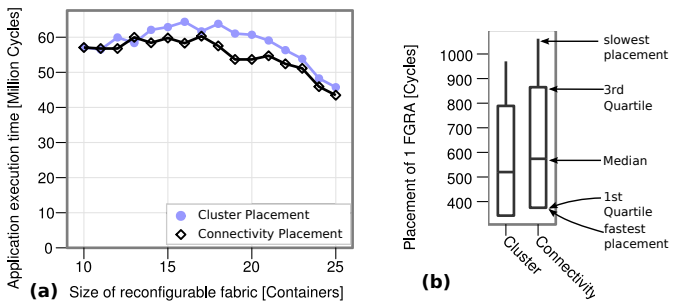

Fig. 5: Placement evaluation: **(a)** Performance of an H.264 video encoder **(b)** Boxplot of the overhead for placing one FGRA

Communication-Lookahead Binding (CLB) is an extension of CAB. In addition to the data transfer $d_{i,j}$ (input data of o_j) CLB also considers $d_{j,k}$ (output data of o_j). However, when binding o_j, then operation o_k that reads the results from o_j is not bound yet. CLB assumes that o_k will be bound to container $c_q, t(c_q) = t(o_k)$ that has the smallest distance to the examined container candidate c_p. The container that results in the least number of cycles to resolve the hazards for input and output data transfers of o_j is selected. The complexity is $\mathcal{O}\left(|containers|^2 * |DataTransfers|\right)$.

VI. Experiments & Results

To evaluate our proposed placement and binding algorithms and to compare reconfigurable architectures that use our proposed partial online synthesis with a reconfigurable architecture that synthesizes the configuration of the CGRI at compile time, we use a SystemC simulator based on our hardware prototype of a processor with a runtime reconfigurable fabric. The processor is a SPARC V8 LEON 2 [19] with a 5 stage pipeline, extended for integration with the reconfigurable fabric. For our prototype we implement both the FGRAs and the CGRI on a Xilinx Virtex-4 FPGA, using partial reconfiguration [20] to reconfigure the containers. The time to reconfigure one FGRA is 0.6-0.7 ms (depending on the complexity of the FGRA), as measured on our prototype. Comparing this reconfigurable processor to a LEON 2 without a fabric, we have measured a mean speedup of 14.11x among 126 different fabric configurations when using the H.264 Video Encoder application.

A. Placing FGRAs

A placement decision does not directly affect the execution latency of an SI variant (that depends on several placement decisions and the binding of the SI variant). To evaluate the placement, we compare the execution time of an H.264 video encoder when using different placement algorithms. Figure 5a shows the application execution time for different sizes of the reconfigurable fabric. The Connectivity Placement is on average 5.4% faster than Cluster Placement (at most 2.8% slower and 12.0% faster, respectively), as it considers the communications between the FGRAs during placement.

Placement and binding are software extensions of the runtime system, thus no additional area overhead is required. To measure the overhead of the placement algorithms, we implemented them as a standalone C program and simulated their execution time. Figure 5b shows a boxplot of the amount of cycles required to place one FGRA. On average, it takes 574 cycles using our Cluster Placement and 635 cycles using our Connectivity Placement. Compared to the time it takes to reconfigure one FGRA this overhead is negligible (1 ms corresponds to 100,000 cycles at 100 MHz), therefore we recommend the Connectivity Placement algorithm.

Parameter	Value
Links	4, 6, 8
Maximum *distance* D that can be traversed in one cycle	2, 6, 10
Containers	20
Fabric configurations	100
SI variants	453 variants of 29 SIs
Binding Algorithms	Random, First Fit (FFB), Comm.-aware (CAB), Comm.-Lookahead (CLB)

TABLE I: Parameters for binding algorithm evaluation

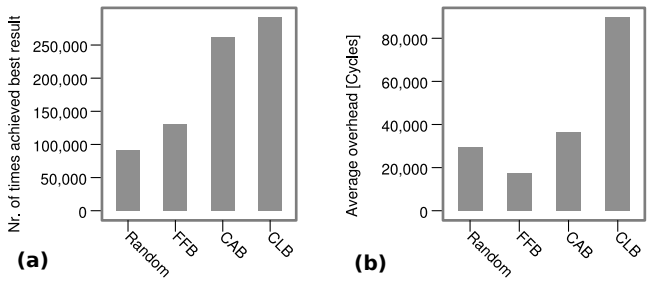

Fig. 6: Binding evaluation: **(a)** Amount of times a binding strategy achieved the best result among all other strategies **(b)** Average overhead for binding one SI variant

B. Binding Operations

Binding directly affects the execution latency of an SI variant, so we directly compare the results of the binding instead of the application's execution time. We use various system configurations (summarized in Table I) that reflect different hardware architectures, as for instance the number of links and the link speed D depends on manufacturing parameters. On each of these configurations each SI variant from a set of multimedia SIs is bound to 100 random fabric configurations (resulting in $\approx$ 1.6 million bindings in total). Figure 6a shows how often a particular binding results in the fastest SI execution latency. Our Communication-Lookahead Binding (CLB) leads to the best results, followed closely by our Communication-Aware Binding (CAB).

Figure 6b shows the average overhead in cycles for the operation binding algorithms. While CLB creates the fastest SI variants on average, it takes over twice as long as CAB and over four times longer than First Fit Binding (FFB). Whether or not the overhead of CLB is acceptable depends on how often the SI is executed. For example, one of the SI variants for the *DCT* SI is bound by CLB resulting in an latency of 35 cycles per execution and an overhead of 46,104 cycles. The same SI variant is bound by FFB to the same fabric configuration resulting in an execution latency of 37 cycles and an overhead of 7,469 cycles. Therefore, the overhead of CLB amortizes if this SI variant is executed for 19,318 times on the same fabric configuration. During our evaluation we have not encountered a situation where an SI was executed for this amount of times for the same fabric configuration. Therefore, we recommend to use CAB or FFB as they provide a better compromise of the obtained result and the overhead.

C. Comparison of our online synthesis vs. offline synthesis

We use a multi-tasking scenario for the comparison of a dynamic runtime system with partial online synthesis and a runtime system with offline synthesized CGRI configurations. The applications for this scenario are an in-house developed *H.264 video encoder*, and the two MiBench [21] applications *Susan image recognition* and *JPEG Decoding*. Four tasks (the JPEG application is used twice) are started, as shown in the Gantt chart at the bottom of Figure 7. Each task performs a certain work (e.g. decode an 800x600 pixel JPEG image) and terminates afterwards. After a task is started, it obtains a portion of the reconfigurable fabric onto that it may reconfigure FGRAs. The size of that portion depends on the task priority (set at compile time) and

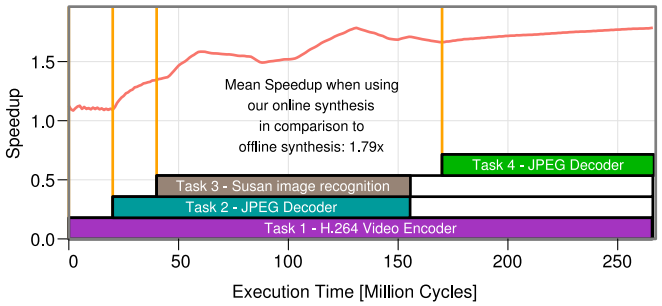

Fig. 7: Comparison of our online synthesis vs an offline synthesis system in a multi-tasking scenario

changes when the number of running tasks changes. Both systems use a reconfigurable fabric with 15 containers and all other parameters are identical as well. The online system uses the Connectivity Placement and Communication-Aware Binding strategies.

As shown in Section II-D, providing CGRI configurations for all possible SI variants of an SI is practically infeasible. Thus, the system with the offline synthesized CGRI configuration can only use a limited set of SI variants. To ensure a fair comparison, we profiled the applications for different fabric sizes and identified the most efficient SI variant, i.e. where the ratio $performance/\#FGRAs$ is highest. For each computational kernel of the application, the offline synthesized system reconfigures one compile-time determined fabric configuration and uses the compile-time placed and bound SI variants.

We define the speedup at time t between two systems a and b as shown in Eq. 9, where $instructions\_executed(i,t)$ returns the number of executed instructions for all tasks of system i in the period $[0,t]$. Figure 7 shows the speedup of our online synthesis system in comparison to the offline synthesis system. Between cycles 0-20M, only Task 1 is running, thus the performance of both systems is comparable – the online synthesis uses high-performance SI variants on all 15 containers, while the offline-synthesis system uses smaller but efficient SI variants, yielding almost the same overall performance. However, when additional tasks are scheduled, less containers are available to each task and the offline-synthesis system cannot execute all SIs in hardware (emulation in software is used instead). Our online synthesis system uses SIs that have less FGRA requirements and creates new CGRI configurations for them on demand. This results in the measured average speedup of 1.79x over the runtime of the four tasks.

$$speedup(t) = \frac{instructions\_executed(a,t)}{instructions\_executed(b,t)} \quad (9)$$

To compare our approach with common techniques as used in high-level synthesis, we have assumed an ideal system that produces bindings *instantly* (i.e. no overhead) and ignores hazards (i.e. 1 cycle per control step). These assumptions lead to an optimal binding result. Our approach is only 6.4% slower than that theoretical upper bound.

VII. Conclusion

In this paper, we present a novel partial online-synthesis approach for mixed-grained reconfigurable fabrics. It combines Fine-Grained Reconfigurable Accelerators (FGRAs, synthesized at compile time) with a Coarse-Grained Reconfigurable Infrastructure (CGRI) for which the configuration is synthesized at runtime. We present all steps of the online synthesis for the CGRI and focus on the crucial parts of Placing FGRAs on the reconfigurable fabric and binding operations to them. We propose and evaluate different binding and placement algorithms and recommend using the Connectivity Placement algorithm (due to the highest quality of placements among the evaluated algorithms while keeping a low overall overhead) and the Communication Aware Binding algorithm (due to its good performance/overhead ratio). Our partial online synthesis exploits the inherent available adaptivity of the reconfigurable fabric. We compare our approach with a state-of-the-art reconfigurable architecture that synthesizes the configurations for the CGRI at compile time (and thus is unable to react on runtime varying application requirements) and we obtain an average speedup of 1.79x.

VIII. Acknowledgment

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) http://invasic.de.

References

[1] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll, "Design flow for embedded FPGAs based on a flexible architecture template", in *DATE*, 2008, pp. 56–61.

[2] M. Huebner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture", in *RAW*, 2011.

[3] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[4] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The MOLEN polymorphic processor", *IEEE Transactions on Computers (TC)*, vol. 53, no. 11, pp. 1363–1375, 2004.

[5] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri, "XiSystem: a XiRisc-based SoC with reconfigurable IO module", *Journal of Solid-State Circ.*, vol. 41, no. 1, pp. 85–96, 2006.

[6] P. Heysters, G. Smit, and E. Molenkamp, "A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems", *Journal of Supercomputing*, vol. 26, no. 3, pp. 283–308, 2003.

[7] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the ADRES coarse-grained reconfigurable array", in *ARC*, 2007, pp. 1–13.

[8] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture", *Journal of Supercomp.*, vol. 26, no. 2, pp. 167–184, 2003.

[9] L. Bauer, M. Shafique, and J. Henkel, "Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors", in *AHS*, June 2011, pp. 80–87, invited paper.

[10] L. Bauer, M. Shafique, and J. Henkel, "A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor", in *FPL*, 2008, pp. 203–208.

[11] L. Bauer, M. Shafique, S. Kreutz, and J. Henkel, "Run-time system for an extensible embedded processor with dynamic instruction set", in *DATE*, 2008, pp. 752–757.

[12] R. König, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "KAHRISMA: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture", in *Design, Automation and Test in Europe*, 2010, pp. 819–824.

[13] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions", in *DATE*, 2011, pp. 1554–1559.

[14] S. Koh and O. Diessel, "Communications infrastructure generation for modular fpga reconfiguration", in *IEEE International Conference on Field Programmable Technology*, 2006, pp. 321–324.

[15] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing", in *Reconfigurable Architectures Worshop*, 2003, pp. 178–185.

[16] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors", *Trans. on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, 2006.

[17] A. Beck, M. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications", in *DATE*, 2008, pp. 1208–1213.

[18] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1989, pp. 661–679.

[19] Aeroflex Gaisler, "LEON 2", http://www.gaisler.com/leonmain.html.

[20] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs", in *FPL*, 2006, pp. 1–6.

[21] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *Int'l Workshop Workload Characterization*, 2001, pp. 3–14.