A Just-In-Time Modulo Scheduling for Virtual Coarse-Grained Reconfigurable Architectures

Ricardo Ferreira*§, Vinicius Duarte*, Waldir Meireles*, Monica Pereira†, Luigi Carro‡ and Stephan Wong§
*Departamento de Informatica - Universidade Federal Vicosa, CEP 36570000, Vicosa, Brazil
Email: ricardo@ufv.br
† DIMAP, Univ. Federal Rio Grande do Norte, CEP:59078-970, Natal, Brazil
Email: monicapereira@dimap.ufrn.br
‡Univ. Federal do Rio Grande do Sul, CEP 91501-970 Po Box: 15064, Porto Alegre, Brazil
Email: carro@inf.ufrgs.br
§ EEMCS, Computer Engineering, TU-DELFT
P.O. Box 5031, 2600 GA Delft - The Netherlands
Email: j.s.s.m.wong@tudelft.nl

***Abstract*—In the past decade, most solutions concerning the mapping of the compute-intensive loop kernels to accelerators have used heuristics and compiler-based strategies. These facts require that most of the decisions be taken at design time, thus precluding efficient solutions that can take run-time information into account. Any success in accelerating such applications greatly depends on two steps, extracting the loops and mapping them into the architecture. This last step is a challenge in itself since it is a NP-complete problem. In this paper, we propose a run-time solution that can provide speed ups of 3 to 6 orders of magnitude for the mapping step when compared to the state-of-the-art at minimal performance degradation, by the combined usage of 3 distinct mechanisms: 1) a simple and efficient modulo scheduling heuristic, 2) a crossbar network, which simplifies the placement and routing, 3) a virtual coarse-grained reconfigurable architecture (CGRA). Additionally, since the CGRA is a virtual layer on top of an FPGA, it is possible to use any off-the-shelf FPGA without the need of special tools or IP solutions. Although the mapping is NP-complete even for crossbar-based CGRAs, experimental results demonstrate a huge reduction in compilation time, as opposed to previous solutions that require seconds to map the applications, our solution requires only microseconds to find near optimal schedules. Besides the speed up, the proposed solution enables the use of just-in-time compilation, hence it is intrinsically adaptive to a changing scenario.**

I. Introduction

Reconfigurable computing emerged as a solution to balance the tradeoff between flexibility, achieved with software programming of general-purpose processors, and high performance of application specific circuits. A particular approach in reconfigurable computing entails Coarse-Grained Reconfigurable Architectures (CGRAs), which have shown that they can provide both power efficiency [1] and hardware acceleration [2].

Accelerating software execution requires a detailed application analysis to find data dependencies, extract parallelism and map the application into the target CGRA. For this reason, many software execution acceleration techniques found in the literature use compile and design time strategies to analyze and map applications [1], [2], [3], [4]. The drawback of these techniques is the fact that no advantage can be taken from the application behavior during its execution. In addition, portability is very important in the embedded system market, and it is normally ensured by virtual machines, just-in-time (JIT) compilers, and binary translation approaches, where a short compilation time and/or dynamic mechanisms are important issues.

In the last decade, a significant advance has been made in techniques to accelerate software execution for CGRAs [2], [3], [5], especially multimedia applications [1], [4]. These applications are characterized by the large amount of software pipelining loops, which have a potential speedup factor [1]. However, mapping them into a CGRA is a challenge in itself since it is a NP-complete problem [2]. To cope with this, the solutions found in literature attempt to improve the mapping step through the use of heuristics and compiler-based strategies. In spite of their efforts, the solutions present a significant delay due to the complexity of the mapping step. Therefore, they can only be used during compile time [2], [4], [6], [7]. Recently, a new formulation for the loop mapping problem on CGRA was presented [2]. Although, this approach could achieve near optimal mapping, the average compilation time is 30 seconds, which makes its use difficult in JIT compilation environments.

This work proposes a new solution to accelerate software execution that combines three distinct mechanisms: a mapping algorithm, a crossbar network, and a virtual coarse-grained reconfigurable architecture. The mapping algorithm consists of a simple and efficient modulo scheduling algorithm to map loops into the virtual CGRA. The algorithm is a greedy heuristic and its implementation is straightforward. The virtual CGRA is a layer that runs on top of a commercial FPGA. The main advantages of this strategy are the low configuration overhead since the CGRA works at word-level instead of bit-level, and the possibility to use any off-the-shelf FPGA. As part of the strategy to reduce mapping delay, we also propose a CGRA based on a crossbar network instead of mesh topologies [1], [2], [3], [4]. In the experimental results, we show that this interconnection model reduces the complexity to map applications into the CGRA and consequently allows a simpler and faster mapping step. Although using a crossbar network reduces mapping complexity, the mapping problem

978-1-4799-0103-6/13/$31.00 ©2013 IEEE

is still NP-complete problem. For this reason, the proposed solution is based on a heuristic approach.

Experimental results demonstrate compilation time reduction by about 3 to 6 orders of magnitude when compared to other solutions [2], [4], [6], [7], [8]. Moreover, as opposed to previous solutions that require seconds to map the applications, our solution is in the order of microseconds. This enables the use of just-in-time compilation and in the future, a complete dynamic solution. We also demonstrate that there is no loss in scheduling quality when evaluating the loop initial interval, the amount of extracted instruction level parallelism, and the resource utilization. For a set of 15 benchmarks, the initial interval is on average 92% of the optimal solution. Additionally, since 90% of multimedia dataflow graphs have less than 100 operations [4], it is possible to successfully map them into a CGRA with only 16 PEs. Even though crossbar cost is $O(N^2)$, for $N = 16$ it has a low cost in area for the proposed CGRA, which is emulated on the top of commercial FPGAs. In addition, a crossbar network significantly accelerates the mapping process. To complement the solution, partial reconfiguration could also be used as a mechanism to ensure performance gains for large loops.

The remainder of this paper is organized as follows. Section II contextualizes the work and present some of the work related to modulo scheduling algorithms, coarse-grained reconfigurable architectures, and the computational complexity. Section III presents the proposed solution, detailing the algorithm and the virtual architecture. Experimental results are presented in section IV. A comparison with previous solutions is also presented in this section. Finally, section V presents conclusion and future works.

II. Background and Related Work

In order to contextualize this work, this section presents a brief overview of the main techniques used: CGRAs and modulo scheduling. We focus on a CGRA as a target architecture and the time to scheduling the loops. The quality of a scheduling is measured by the minimum initial interval (MII) between two successive i^{th} and $i + 1^{th}$ loop iterations. Next subsections present the main concepts behind the techniques and the previous works that first proposed and other works that implement the techniques.

A. CGRA Architectures

The interconnections of CGRA architectures are crucial to simplify the scheduling, placement and routing steps (SPR). Depending on the interconnection model, its complexity can cause a significant impact on SPR time. One possible strategy to reduce complexity is by adding routing resources. Therefore, one must consider the tradeoff between architecture interconnection cost and compilation time.

One of the first works to propose a CGRA was PADDI [9]. PADDI uses a crossbar network where the routing and placement are straightforward steps. Nevertheless, PADDI CGRA cannot be configured dynamically, as the configuration is loaded in setup time. Recently, an approach proposed to replace the crossbar networks by using multistage network [10]. However, only medium and large architectures with 64 and 256 processing elements (PEs) has been evaluated due to the register allocation strategy. In addition, multistage networks are blocking and the routing could fail.

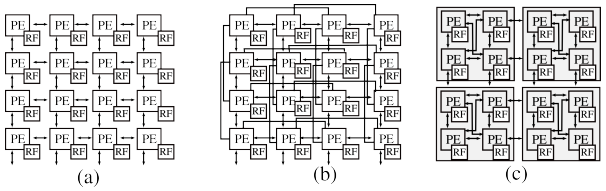

Fig. 1. (a) Mesh (b) Mesh Plus (c) Cluster

Concerning modulo scheduling in CGRA, the DRESC compiler from the ADRES framework [3] is one of the pioneer works, using simulated annealing to reach the best scheduling [11]. Nevertheless, the DRESC compiler is very time consuming, even for small loops. One goal of the ADRES framework [3] was the design exploration of mesh CGRA based topologies, as shown in Fig. 1(a-c), to evaluate the tradeoffs between resources (interconnections, registers) and scheduling quality. A large amount of modulo scheduling approaches uses mesh-based topologies [12], [8], [6], [13], [1], [2], [7], which reduces the interconnection costs, however the challenge is how to reduce the compilation time.

Fig. 1(a) depicts a 4×4 CGRA where each PE has a direct connection to its four neighbors (north, south, east, west). In addition, each PE has a local register file to store temporally values. Fig. 1(b) shows a mesh plus interconnection, which adds connections that routes over the neighboring PEs. This feature improves the routability and simplifies the placement and routing which in turn, improves the scheduling quality and reduces the compilation time.

Fig. 1(c) depicts a local cluster structure where each 2×2 tile is fully interconnected by a local crossbar. Since the crossbar cost is $O(n^2)$, and for this reason, only small values of N (2 or 4) and few crossbar-based CGRA has been proposed [9].

We propose a simplification in both architectural support and algorithm implementation. Moreover, as most of the loops from multimedia applications has small size, they can fit into a 16-PEs architecture [2]. In current solutions, a 32-bits width 16 in/out crossbar has a reasonable size in current commercial FPGA, which is similar to 4x4 CGRA with enriched 4x4 mesh interconnection. In addition, the crossbar routing is O(1), and there is no placement problem since any PE could achieve any other PE. Therefore, the algorithm is simplified without additional costs at architecture level as demonstrated in following sections.

B. Modulo Scheduling Algorithms

Modulo scheduling (MS) [5] is a software pipelining technique which overlaps different iterations of a loop to exploit a higher degree of Instruction Level Parallelism (ILP). This technique is based on the "*schedule-and-move*" approach where a schedule for one loop iteration is originated and repeated for the other loop iterations at regular intervals.

Fig. 2 shows an example of ADRES modulo scheduling algorithm. Fig. 2 (a) depicts a dataflow graph (DFG) of a loop section. For ease of explanation, it is considered a 2x2

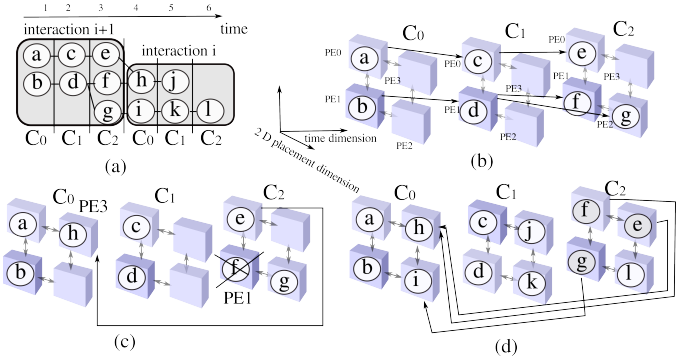

Fig. 2. Mesh Modulo Scheduling: (a) Graph (b) TEC Initial Placement (c) TEC Routing Fail (d) TEC Successful

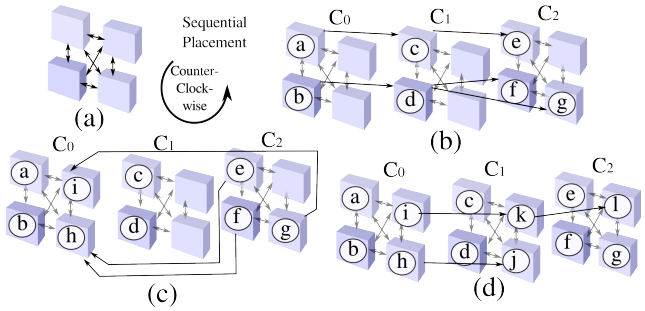

Fig. 3. Example of Crossbar Modulo Scheduling

mesh architecture or 4 PEs. As the graph has 12 nodes, the scheduling will need at least (12 nodes/ 4 PEs) = 3 configurations labeled as C_0,C_1, and C_2. Fig. 2 depicts Time-Extended CGRA (TEC) graph [2] where the CGRA is extended in time. For this example, the CGRA is extended 3 steps in time, one for each configuration. The DFG must be mapped into the TEC. In the example of Fig. 2, the nodes a and b are placed in PE_0 and PE_1 at C_0. Then c and d should be scheduled at C_1 due to the data dependence on a and b. Fig. 2(b) shows a placement for c and d in the same PEs where a and b have been placed. For e, f, and g, a possible placement and routing at C_2 is shown in Fig 2(b). However, if h is placed in PE_3 at C_0, it is not possible to route f from PE_1 to h in PE_3 since there is no diagonal connection. The routing fails as shown in Fig 2(c).

A possible solution to enable the complete routing is to change the placement of e, f, and g at C_2. Then, e and f can route to h from C_2 to C_0, as well as the entire graph: $g \rightarrow i$, $h \rightarrow j$, $i \rightarrow k$, and $k \rightarrow l$ (see Fig 2(d)). Therefore, in general, more than one placement and routing should be evaluated to reach a valid mapping where all data dependencies between nodes are preserved.

For this example, the optimal solution is found, where the II=3 and the ILP is 4 as all PEs are used in all configurations. It is important to highlight that the loop iterations are overlapped. When the iteration i executes nodes $h, \ldots, l$, the next iteration $i + 1$ executes nodes $a, \ldots, g$ as shown in Fig 2(a). Moreover, nodes from different iterations share the same configuration, as for example a, b and h, i at C_0. The modulo scheduling executes instructions of different loop iterations at the same time. For this example, two iterations are overlapped.

Traditional scheduling approaches [12], [13], [14] assign nodes to PEs during the placement step. For this reason, they are called node-centric solution. In this case, routing is done to verify if the assignment is feasible. On the other hand, in an edge-centric approach the routing is the primary objective, and the placement is done during the routing process. An edge-centric solution was proposed in [4], called edge-centric modulo scheduling (EMS). The main advantage of EMS technique is the reduction of compilation time when compared to ADRES algorithm [4]. In spite of that, the II is increased in comparison to ADRES approach, which means a performance penalty at execution time. Recently, a new approach called EPIMAP is presented in [2], which reaches the optimal II by using recomputation and routing approaches. Nevertheless, the compilation time is 6 times slower than EMS [4]. The recomputation and routing are performed in the multicast nodes. Although, the number of operators could be increased, the II is minimized. EPIMAP presents the best results for optimal II in comparison to others approaches. Moreover, the EPIMAP presents a formal definition of the modulo scheduling problem and proves its NP-Completeness.

In order to reduce routing cost, two approaches [10], [7] propose the use of multicast and sharing connections. The first approach, called MSPR, is based on a multistage network [10] to simplify the placement and routing. This is achieved because with the multistage all the PEs become fully connected. The second approach [7] uses the register files as routing resources and shares multicast connections, where the MS is modeled as a graph mirror problem improving resource allocation and reaching the same II compared to DRESC [11].

The algorithm proposed in this work reaches scheduling near the optimal, such as EPIMAP [2]. It also handles efficiently multicast to share connections, such as G-Minor [7], and simplify the placement and routing like MSPR [10]. In addition, the compilation time of the proposed approach is significantly reduced in more than three orders of magnitude. Although the previous works present compilation time improvements, they are not simple enough for an efficient JIT implementation once their average time are measured in seconds. In this work, we propose to use a crossbar network as the placement and routing steps for a crossbar network are O(1). Fig. 3(a) depicts the proposed CGRA drawn in two dimensions. The crossbar CGRA graph is a complete graph. Fig. 3(b-d) display the Crossbar TEC. Moreover, we propose to place the nodes sequentially in a counterclockwise direction as depicted in Fig. 3(b), where nodes $a, \ldots, g$ are sequentially placed in $PE_i, PE_{i+1}, \ldots$ through the configurations C_0, C_1 and C_2. Then Fig. 3(b) depicts the TEC, where nodes h and i are placed at C_0 by using the sequential placement without routing conflicts. Finally, the final TEC is depicted in Fig. 3(d). If there is a data dependency between x and y (ie:$x \rightarrow y$), and x is scheduled at C_i, then y should be scheduled at $C_{(i+1) moduloII}$, where II is the initial interval.

C. NP-Complete

The mapping process consists of three steps: modulo scheduling, placement and routing. The placement and routing itself is NP-complete for mesh interconnections [14], [8]. The

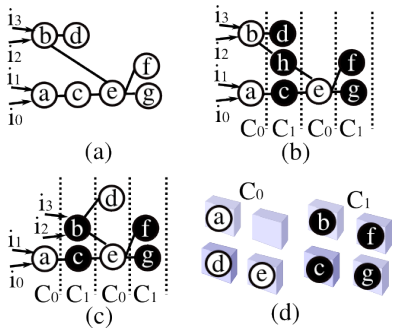

Fig. 4. Example: (a) DFG (b) Balanced (c) Valid Scheduling (d) TEC

```
1:  For each node t, with sources s1 and s2 {
2:    If (s1 and s2 are inputs) {
3:      cfg = Initial;  Place[t] = free_PE[cfg]++;  // Placement
4:      Scheduling[t] = Cfg[t] = cfg;  // Initial Configuration
5:    } Else {
6:      If (Scheduling[s1] == Scheduling[s2]){
7:        PE = Place[t] = free_PE[cfg]++;    // Placement
8:        Scheduling[t] = Scheduling[s1] + 1;
9:        Cfg = (Cfg[s1]+1) mod II; Cfg[t] = cfg; // Modulo Sched.
10:       RouteA[PE][cfg] = Place[s1]; RouteB[PE][cfg] = Place[s2];
11:     } Else {
12:       Reg = Insert_Register(Sched[s2],Sched[s1]);
13:       PE = Place[t] = free_PE[cfg]++;
14:       Cfg[t] = cfg;            // cfg is a Register Configuration
15:       RouteA[PE][cfg] = Reg; RouteB[PE][cfg] = Place[s2];
16:     }
17:   }
18: }
```

Fig. 5. Pseudo Code for the Basic Algorithm

mesh has a $O(N)$ connection cost. When, a mesh is replaced by a crossbar network, the connection cost increases to $O(N^2)$, however the placement and routing become $O(1)$. In spite of the reduction in placement and routing complexity, the overall complexity of the mapping process includes all three steps, modulo scheduling, placement and routing. For this reason, one of the key questions that must be answered is what is overall complexity of the mapping process.

Only recently, the problem of mapping an input DFG into a CGRA has been proved to be NP-Complete in [2]. Although the CGRA in [2] is defined as an array of PEs interconnected by a 2-D grid or a mesh interconnection, the prove is based on a CGRA as a general graph. The mapping is described as the problem of finding a subgraph in a minimally TEC graph that is Epimorphic to the input graph. Therefore, since the crossbar CGRA is also modeled as a TEC, where the scheduling constraint should be satisfied, the problem remains NP-complete. Since there are no placement and routing constraints, it is still possible to accelerate mapping process, by investing in scheduling strategies. Nevertheless, the problem is still NP-complete and heuristic approaches should be applied to reduce even more the solution space. These heuristics should balance the tradeoffs between scheduling quality and time complexity.

Fig. 4(a) presents a DFG to be mapped into a 4-PEs CGRA. First, the graph needs to be balanced as shown in 4(b), where node h is inserted into the path $b \rightarrow e$. The minimal II is 8 nodes$/$ 4 PE $= 2$ configurations. However, it is not possible to map due to the scheduling depicted in 4(b). As can be observed, the level or configuration constraint is violated in C_1, where 5 nodes are scheduled (d, h, c, f, g) and the maximum is 4 PEs, even if a crossbar network is used. Nevertheless, there is a valid scheduling with the Minimal II=2 as shown in 4(c). Therefore, if the nodes are re-scheduled it is possible to achieve MII=2. If b is scheduled at C_1, since b's sources are external inputs, d is shifted to C_0 and the node h is not inserted to balance the path $b \rightarrow e$. Finally, the TEC graph has at most 4 nodes in each configuration as shown in 4(d), and the minimal scheduling is found. Although it is a simple example, by using a threshold value to limit the maximum of external inputs per configuration, the most efficient scheduling could be found since the DFG nodes are better distributed across the configurations.

III. Modulo Scheduling Heuristic

This section details the proposed modulo scheduling heuristic considering a crossbar CGRA design. In addition, two register saving techniques are presented which improves the scheduling quality.

A. Basic Algorithm

Our proposed algorithm is based on a greedy heuristic composed of simple data structures (vectors) and with time complexity $O(N)$. The pseudo-code is described in Fig. 5. It is important to note that the pseudo-code closely resembles C code, and its implementation is straightforward and very efficient. Similar to others approaches [2], the graph is traversed in topological order. However, the input graph is not balanced, and registers are inserted on-the-fly during the graph traversing. In this work, we assume that the starting point is the dataflow graph (DFG), similar to others approaches [2], [4], [6], [7], [8], [13]. Moreover, our approach is based on a greedy heuristic where the DFG is traversed just once for a given initial interval (II), which leads to a huge reduction in the compilation time. In contrast, others approaches try different schedules, placements and routings for each II. If the scheduling is not feasible, the II is increased and the scheduling is performed again until a feasible solution is found. Although we just try once for each II, we achieve the optimal solution in the first try for 10 of 15 DFGs.

The graph traversal is performed per node. The code uses six vectors: *Place* maps nodes into PEs, *Scheduling* keeps track of scheduling time; *Cfg* stores the configuration; *Free_PE* is used to point to the next free PE; *RouteA* and *RouteB* store the routing per configuration. Let t be the target node, and s_1 and s_2 are the source nodes. If t has external inputs (lines 2-4 in Fig. 5), the scheduling, placement and routing is straightforward. It is important to note that the placement is performed sequentially by using a simple assignment (line 3), which is an O(1) operation. If s_1 or s_2 have been already scheduled, there are two possibilities. First, s_1 and s_2 has the same scheduling and no balance is needed (lines 6-10). t is scheduled in the next configuration by using the modulo operation (line 7). The routing is also a single assignment as shown in line 10. The second possibility occurs when s_1 and s_2 has different scheduling. Then, one or more registers are inserted (line 12), and finally the scheduling, placement and routing is performed.

B. Sharing Registers

Fig. 6(a) depicts an example of reconvergent paths, very frequent in multimedia algorithms. These paths could have

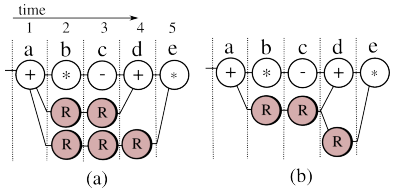

Fig. 6. (a) Original Balance (b) Sharing Registers

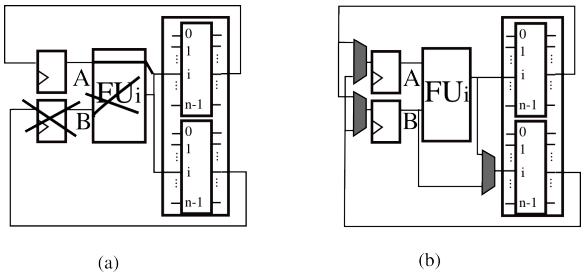

Fig. 7. (a) Bypassing A (b) Using both Input Registers

different lengths. If the edges are not balanced, it is necessary to insert temporary registers or nodes for balancing, which in turn increases the cost [2], [7]. In case of multicast edges, they will be balanced separately (see Fig. 6(a) for $a \rightarrow d$ and $a \rightarrow e$), as proposed in previous approaches [3], [4], [6]. Recently, a shared multicast routing was presented in [7] for the ADRES based architecture to minimize the number of temporary registers. Although, our approach is based on a different target architecture, we propose a simple technique to reduce the number of registers in presence of multicast, as depicted in Fig. 6(b). When node d is processed, two registers are inserted. An additional vector will keep track of the last node connection per configuration, which is implemented by the function *insert_register* (line 12) in Fig. 5. When the node e is being processed, the source node a is checked. Since the last source node a is scheduled at time 3, only one register is needed to balance the path from a to e.

C. Local Registers

The CGRA architecture presented in this work is based on the target architecture proposed in [10], where the multistage interconnection network is replaced by a crossbar network. Each PE_i has one internal functional unit (FU) and two input registers: A and B. The FU's output is connected to the two crossbar networks. This enables the inputs A and B to receive data from different FUs at the previous configuration. When a register is needed at PE_i, either the register A_i or B_i is used. Moreover, the FU_i is bypassed. Therefore, the register B_i and the FU_i are idle, and only the register A_i is used to implement a register operator as shown in 7(a).

We propose to use both local registers by adding multiplexers as shown in Fig. 7(b). Although the basic algorithm should be changed, the modifications are simple and do not cause a major impact. First, a vector is needed to store the first free register (free_REG) in addition to the vector free_PE, which keeps track of the first free PE of each configuration. The free_REG vector will be updated by the function Insert_Register at line 12. In addition, only line 15 should be modified if the last allocated register uses the B crossbar network to route.

TABLE I. NUMBER OF PES USED AS REGISTERS FOR THREE STRATEGIES: BASIC (BA), SHARING REGISTERS (SH), AND LOCAL REGISTERS (LO)

Name	BA	SH	LO
arf	13	13	7
motion	17	17	10
ewf	51	34	18
fir2	9	9	5
fir	6	6	4
Cplx8	34	31	16
Fir16	28	28	15
h2v2	52	51	27
feedback	24	23	13
FilRGB	33	31	16
collapse	-	-	20
cosine1	16	8	5
cosine2	-	-	25
DCT	27	17	12
write	-	-	20

IV. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the proposed greedy heuristic solution, a set of dataflow benchmarks from [15] were selected. These DFGs are derived from Mediabench benchmarks. In addition, six DFGs are extracted from DSP algorithms: FIR16 and FIR64 are an 1-D finite-impulse response filter (versions with different number of taps are used), CPLX8 is a FIR filter using complex arithmetic, FDCT is a fast discrete cosine transform implementation, and The FilRGB is an image filter to highlight an image by brightening or darkening the pixels in the images. The DFGs range in complexity from 28 to 196 nodes, and from 1 to 48 inputs.

A. Register Saving

Table I shows the results for the amount of PEs used as a register to balance the loop graphs. This result is based on the three strategies described in Section III. To obtain the amount of required PEs, 15 benchmarks were evaluated, ranging in complexity from 28 to 106 nodes (without considering balancing registers). The target architecture consists of a 16-FUs CGRA with a crossbar network. Each configuration could receive at most 4 external inputs. Similar assumptions are made in EMS [4], and EPIMAP [2].

Column *name* depicts the benchmark name. The other columns are labeled according to the register strategies: basic (BA), sharing registers (SH), and local registers (LO). Considering that the basic strategy (BA) uses an entire PE as a bypassing register, in specific configurations, there is no enough register or PE. For this reason, the scheduling fails for 3 of 15 benchmarks. On the other hand, the sharing register strategy (SH) reduces the number of registers in 7 of 15 benchmarks. However, this strategy also fails in 3 of 15 benchmarks. The reduction is more effective in three cases: ewf (33%), cosine1 (50%), and DCT (37%). For these benchmarks, there are multicast edges. The local register strategy is simple and effective, where both input registers could be used as a bypassing resource. In the best case, this strategy reduces up to 50% the number of bypassing PEs. This approach saves on average 42% of bypassing PEs for 12 benchmarks. In addition, 3 benchmarks which could not be scheduled (with one trying per II) for the others strategies are successfully mapped by the local register approach.

TABLE II. INITIAL INTERVAL FOR THREE STRATEGIES:BASE (BA), SHARING REGISTERS (SH), AND LOCAL REGISTERS (LO)

Name	In	Op	MinII	Achieved II			Op+R
				BA	SH	LO	
arf	8	28	3	3	3	3	35
motion	14	32	4	4	4	4	42
ewf	2	34	4	6	5	4	52
fir2	16	40	4	4	4	4	45
fir	22	44	6	6	6	6	48
Cplx8	1	46	4	6	5	4	62
Fir16	1	49	4	5	5	5	64
h2v2	16	51	5	10	10	6	78
feedback	21	53	6	6	6	6	66
FilRGB	2	57	5	6	6	5	73
collapse	6	59	5	-	-	6	79
cosine1	16	66	5	7	7	7	71
cosine2	31	81	8	-	-	8	106
DCT	1	92	7	11	11	11	104
write	38	106	10	-	-	10	126

B. Initial Interval

Table II depicts the achieved initial interval (II) for the three register strategies. The columns In, OP and MinII depict the number of external inputs, the number of graph operators, and the minimum II, respectively. The MinII is computed by $Max((OP + R)/FU, In/MaxIn)$ where $OP + R$ (last column in Table II) is the number of bypassing PE plus the number of graph operators, In is the number of external inputs and $MaxIn$ is the maximum number of external inputs allowed per configuration. To enable a comparison with recent works [2], [7], [4], we make the same assumption about the amount of external inputs or memory operations, $Maxin = 4$. Furthermore, graph size definition was based on the fact that most loops for multimedia applications has less than 100 operations. As demonstrate in [4], for 214 loops from H.264, ACC, 3D and MP3 codes, 90% of loops has less than 100 operators. Based on this, we defined the graph size of the benchmarks ranging from 35 to 126 nodes for the balanced graphs.

The quality of scheduling results were measured by comparing the minimum II with the achieved II for the three strategies. The local register strategy achieves the minimum II in 10 of 15 benchmarks. To evaluate the scheduling quality, we performed a comparison between the initial intervals of the proposed solution and EPIMAP [2] solution. EPIMAP proposed a solution based on smart exploration of solution space, and it is considered one of the most efficient modulo scheduling solutions. In the solution proposed in this work, only one scheduling is performed per II, and even in this case, the achieved II is closed to the optimal solution Moreover, the proposed approach achieves 92% of MinII for 15 DFG with on average 70.1 nodes. On the other hand, the EPIMAP results show on average 93% of MinII for 14 DFG with an average 57.6 nodes. Therefore, the scheduling quality is similar even considering larger DFGs. Our DFGs are available in [15]. BA and SH strategies fails (represented by a dashed line -) for three benchmarks with more than 79 operators and bypassing registers.

C. Architecture Size

Table III depicts the initial interval (II), the instruction level parallelism (ILP) and the occupancy for three architecture sizes with 16, 20 and 24 PEs. The DFG sizes range in complexity from 28 to 196 nodes. The DFGs are ordered by size in Table III. The ILP is computed by the number of effective operations divide by the II not counting the bypassing registers. The occupancy represents the percentage of used resources during execution ($100 - OCC$ indicates the percentage of idle resources).

TABLE III. INITIAL INTERVAL, ILP AND OCCUPANCY FOR THREE ARCHITECTURE SIZE: 16, 20 AND 24 PE

Name	II 16 / 20 / 24	ILP 16 / 20 / 24	Occ (%) 16 / 20 / 24
arf	3 / 2 / 2	9.3 / 14.0 / 14.0	72.9 / 87.5 / 72.9
motion	4 / 4 / 4	8.0 / 8.0 / 8.0	65.6 / 52.5 / 43.8
ewf	4 / 3 / 3	8.5 / 11.3 / 11.3	81.3 / 86.7 / 72.2
fir2	4 / 4 / 4	10.0 / 10.0 / 10.0	70.3 / 56.3 / 46.9
fir	6 / 6 / 6	7.3 / 7.3 / 7.3	50.0 / 40.0 / 33.3
Cplx8	4 / 4 / 3	11.5 / 11.5 / 15.3	96.9 / 77.5 / 87.5
Fir16	4 / 4 / 3	12.3 / 12.3 / 16.3	100.0 / 81.3 / 88.9
h2v2	5 / 4 / 4	10.2 / 12.8 / 12.8	97.5 / 96.3 / 80.2
feedback	6 / 6 / 6	8.8 / 8.8 / 8.8	68.8 / 55.0 / 45.8
FilRGB	5 / 4 / 4	11.4 / 14.3 / 14.3	91.3 / 91.3 / 76.0
collapse	5 / 4 / 4	11.8 / 14.8 / 14.8	98.8 / 98.8 / 82.3
cosine1	5 / 4 / 4	13.2 / 16.5 / 16.5	88.8 / 88.8 / 74.0
cosine2	8 / 8 / 8	10.1 / 10.1 / 10.1	82.8 / 66.3 / 55.2
DCT	7 / 6 / 5	13.1 / 15.3 / 18.4	92.9 / 85.0 / 85.0
write	10 / 10 / 10	10.6 / 10.6 / 10.6	78.8 / 63.0 / 52.5
interpol	12 / 12 / 12	9.0 / 9.0 / 9.0	75.0 / 60.0 / 50.0
matmul	- / 7 / 6	- / 15.4 / 18.0	- / 92.1 / 88.9
jpgfast	- / - / 14	- / - / 11.9	- / - / 99.1
jpgslow	- / 15 / 12	- / 11.5 / 14.4	- / 93.7 / 96.9
idctcol	- / - / 15	- / - / 12.4	- / - / 96.4
TFir64	19 / 15 / 13	10.2 / 12.9 / 14.8	96.1 / 96.3 / 92.9
smooth	- / 16 / 16	- / 12.3 / 12.3	- / 82.5 / 68.8
aver.		10.3 / 11.9 / 12.8	82.8 / 77.5 / 72.2

In this experiment, the results allows performing a set of different analysis. Firstly, the quality of the scheduling for 7 in 22 DFGs does not improve when more PEs are added. For instance, the II for interpol benchmark remains 12 for all the architecture sizes. In this case, a large number of external inputs limits the maximum parallelism. Moreover, the 16 PEs have a better occupancy, as well as reduce the amount of idle resources, as shown in last three columns in Table III. The second analysis is related to the last row, which depicts the average ILP of 10.3, 11.9, 12.8 for 16, 20 and 24 PEs, respectively. The results demonstrate that the 20-PEs architecture only improves in 15% the ILP but it uses 25% more PEs. Furthermore, the 24 PEs architecture increases the number of PEs in 50%, however the ILP is only 24% better than the 16-PEs architecture. For most cases, the 16-PEs architecture shows a good tradeoff in quality and ILP even for a small architecture. However, for larger DFGs (last fives line in Table III), due to the greedy heuristic (just one scheduling per II) and the small number of local register, the scheduling for 16-PEs architecture may fail, and for this reason, larger architectures with 20 or 24 PEs are needed. For these cases, partial reconfiguration could be used to dynamically reconfigure the CGRA when larger DFGs are executed.

The second experiment is used to measure the area of the proposed CGRA implemented on the top of a commercial FPGAs. Most modulo scheduling solutions does not report any results regarding physical implementations [12], [8], [6], [13], [2], [7], only the scheduling quality and compilation time are reported. For this reason, a detailed comparison among area results was not possible. Nonetheless, in Table IV, we present area results of the proposed solution in comparison to ADRES

TABLE IV. PERCENTAGE OF FPGA RESOURCES FOR CROSSBAR AND ADRES BASED CGRA IN A XILINX XC6VLX75T

Size	FF (%)	LUT %	RAM %	Feq Mhz	Clk ns
Crossbar 16 PEs	2.7	17.6	4.5	90	11.1
Crossbar 20 PEs	3.4	30.1	6.4	72	13.8
Crossbar 24 PEs	4.0	39.5	7.7	57	17.5
ADRES 4x4	2.5	14.7	16.0	110	8.8

architecture.

Table IV depicts the FPGA resource used for all three architecture sizes with 16, 20 and 24 PEs in comparison to ADRES architecture. We select a medium size FPGA xc6vlx75t from XILINX. This FPGA has 93,120 FlipFlops, 46,560 6-input LUTs, and 156 embedded RAM blocks. In this work, we assume homogeneous PEs, which implement 32-bits logic and arithmetic operations (including multiplication). The ADRES architecture has a global register file with 64 registers, 4-write and 8-read ports, meshplus interconnections and 8 local registers as described in [16].

The number of registers or FF (shown in Column FF) is negligible for all architectures. Moreover, the number of LUTs for the 16-PEs architecture and ADRES are very similar. The 16-PEs architecture uses more LUTs, since crossbar interconnections is more expensive than mesh-local connections. Therefore, although 16-PEs architecture is feasible with a small area overhead, since the complexity is $O(n^2)$, the required LUT for 20 and 24 PEs increases significantly as well the clock cycle (Column Clk in Table IV). Nevertheless, it is still feasible in medium size FPGAs. Regarding the number of RAM used to store the configuration bits for all architectures and the global register file for ADRES, the 16-PEs architecture uses very few resources, only 4.5% or 7 RAMs compared to ADRES, which uses 9 RAMs to store the configuration bits plus 16 RAMs for the global register file.

D. Compilation Time

The proposed solution presents a significant reduction in compilation time. The substantial reduction enables the use of Just-in-Time implementation, which is a technique not feasible in previous solutions. Since the previous approaches [12], [8], [6], [13], [2], [7] use a desktop general-purpose processor (CPU) to measure the compilation time, we have also executed the proposed heuristic in a commercial CPU. In addition, we evaluated the proposed heuristic in two FPGA softcore processors. Softcores could be one solution for future complete dynamic approaches.

Table V shows the compilation time for the three different platforms. First, we use a commercial superscalar processor: an Intel i7, 1.7Ghz with 256Kb L1 cache. For this processor, the algorithm was compiled using gcc 3.4.2 with -O3 optimization option. The second processor is the softcore Xilinx Microblaze running at 150Mhz, and the algorithm was compiled using Cygwin make 3.79.1 with -O3 optimization option. The third processor is ρ Vex, a VLIW softcore processor [17] at 100Mhz, with 4-issue. The algorithm was compiled using HP VEX compiler 3.41 with -fno-xnop and -O3 optimization option.

The results are presented in Table V. The first significant result analysis is related to the time unit. While the previous solutions uses seconds as time unit, the proposed solution presents results in μ seconds. These results show up to 6 orders of magnitude in compilation time reduction. The last three rows in Table V show average results for the 15 benchmarks evaluated. First, the total time shows that the i7 is on average 41.2x and 18.5x faster than the softcore Microblaze and ρ Vex processors, respectively. Moreover, the i7 clock is 11.3x and 17x faster than the softcore processors, respectively. Second, we propose to normalize the results to compare the complexity of the algorithm implementation. The row **Time per Node** depicts the average time spent to scheduling one DFG node. Finally, the row **Number of Cycles per node** depicts the average number of processor cycles spent to map a DFG node. This result shows that the complexity of implementation (which depends on processor and compiler features) is similar for i7 and ρ-Vex processor, and it is around 300 processor clock cycles per node.

TABLE V. COMPILATION TIME IN μ SEC. FOR THREE DIFFERENT PROCESSORS: I7, MICROBLAZE, AND ρ-VEX

Name	i7	MicroBlaze	ρVex
arf	7.20	232.79	107.23
motion	3.80	127.75	60.35
ewf	5.40	368.99	165.17
fir2	3.70	125.56	58.68
fir	3.80	120.54	56.52
Cplx8	8.80	339.37	153.86
Fir16	9.50	370.13	168.36
h2v2	14.20	570.02	261.17
feedback	4.30	163.60	75.86
FilRGB	9.70	387.17	175.27
collapse	14.60	610.13	274.04
cosine1	12.70	484.17	218.60
cosine2	5.60	246.49	111.82
DCT	33.30	1,460.56	628.74
write	5.60	246.43	111.83
Total Time	142	5854	2627
Time per Node	0.17	6.99	3.14
Number of Cycles per node	292.57	1047.80	313.54

TABLE VI. COMPILATION TIME AND CYCLES PER NODE

Algorithm	clk	Time (sec.) Graph	Node	Cycles	Reduction Factor i7	ρvex
DRESC	2.66	104	0.73	$2.0\ 10^9$	$6.8\ 10^6$	$6.2\ 10^6$
EPIMAP	2.66	30	0.17	$4.5\ 10^8$	$1.6\ 10^6$	$1.4\ 10^6$
RF	1.0	110	0.5	$4.9\ 10^8$	$1.7\ 10^6$	$1.6\ 10^6$
EMS	2.66	5.6	0.04	$1.04\ 10^8$	$3.6\ 10^5$	$3.3\ 10^5$
Gminor	2.66	3.4	0.04	$1.04\ 10^8$	$3.6\ 10^5$	$3.3\ 10^5$
RAM	2.66	3.9	0.01	$2.7\ 10^7$	$9.3\ 10^4$	$8.6\ 10^4$
MSPR	2.66	0.09	0.0002	$4.6\ 10^5$	$1.6\ 10^3$	$1.5\ 10^3$

Table VI presents the compilation time, in orders of magnitude, for seven modulo scheduling approaches found in literature: DRESC [11]; EMS [4]; RF [18]; RAM [6]; MSPR [10]; G -Minor [7]; and EPImap [2], which the label is used in Column Algorithm VI. The compilation time results were obtained from the respective references, with exception of DRESC, which time results were based on information reported in [7], [6]. Column Clk displays the clock frequency used to measure the compilation time. Column Graph shows the average time required to compile an entire graph. In 6 of 7 algorithms, the time ranges from 3.4 to 110 seconds per graph, which is feasible for offline or static compiler. However, only the MSPR algorithm [10] presents a short time suitable for Just-in-Time. Based on these data, we have computed the average time spent per node, displayed in Column Node. Furthermore, we have also computed the average number of cycles to process one DFG node. Finally, the two last columns

depict the reduction factor in number of clock cycles per node obtained by the proposed approach for the i7 and the ρ vex execution. The results show that a massive reduction ranging from three to six orders of magnitude was achieved.

To achieve the significant compilation time reduction presented in the experimental results, a set of design strategies was adopted. Firstly, all previous works present high level pseudo algorithms, and the implementation of some steps is complex and not straightforward. The algorithm proposed in this work is very simple and close to the final codification in C language. Moreover in most solutions, several possible schedules are evaluated per II and, for each scheduling, several placement and routing steps are performed. On the other hand, the proposed approach tries only one scheduling per II, and each DFG node is visited only once. Additionally, a complexity reduction is generated due to the use of a crossbar network which simplifies the placement and routing to O(1) steps.

As a last possible comparison to previous solution, we have also compared the proposed solution with the one proposed in [10], called MSPR solution. MSPR uses a similar approach based on multistage networks. However, there is a significant difference in compilation time. The first reason for the MSPR presenting worst compilation time is related to the routing step. The routing step is $O(lgN + 2^E)$ for multistage, where E is the number of extra levels. Moreover, there are routing conflicts, since multistage are blocking networks. Secondly, due to the routing conflicts, all available PEs are evaluated during the placement. While in the approach proposed in this work, sequential placement and direct routing are used (single assignments). Finally the algorithm has a more complex implementation and the quality of the results are worst since MSPR does not save or use local register to achieve better II, and therefore, more scheduling should be tried.

V. Conclusion

This paper proposed a solution to accelerate software execution by using modulo scheduling heuristic to map applications into a virtual coarse-grained reconfigurable architecture. The virtual architecture runs on top of a commercial FPGA and has 16 processing elements. A crossbar interconnection network simplifies the placement and routing steps, which becomes straightforward, as any PE can directly reaches any other PE. Although, the crossbar networks are not scalable as the cost complexity is O(N^2), for $N = 16$, the CGRA size is similar to the mesh-based CGRA. Experimental results demonstrate gains around 3 to 6 orders of magnitude when comparing compilation times of existing solutions. Moreover, the results also show a high level of parallelism extraction and an optimized initial interval in more than 92% of the tested benchmarks for most data-flow graphs (DFGs) found in multimedia algorithms with an average size of 70 operations per software pipelining loop. Therefore, by combining a straightforward architecture with simple heuristic based on greedy decisions, simple assignments and few vectors to store the scheduling, placement and routing data, it was possible to achieve high performance in order to allow a JIT compiler as efficient as the static compilation time algorithms. Future works include more investments on local register to allow efficient execution of large DFGs in small size architecture. As proposed in [2], recomputation could be used to explore the scheduling solution space. However, the search mechanism should be simple enough to keep algorithm simplicity and short execution time.

Acknowledgment

This work was supported by TU Delft, Netherlands and the Brazilian Institutions and Companies: Science without Borders/CNPq, CAPES, UFV, UFRGS, UFRN, Funarpos/FUNARBE, FAPEMIG, and Gapso.

References

[1] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proc. MICRO*, 2009, pp. 370–380.

[2] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proc. DAC*, 2012, pp. 1280 –1287.

[3] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proc. DATE*, 2003.

[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 *Proc. PACT*, 2008.

[5] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proc. MICRO*, 1994, pp. 63–74.

[6] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. LCTES*, 2009, pp. 21–30.

[7] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," in *Proc. FPT*, 2012.

[8] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proc. DATE*, 2006, pp. 363–368.

[9] D. Chen, L. Guerra, E. Ng, M. Potkonjak, D. Schultz, and J. Rabaey, "An integrated system for rapid prototyping of high performance algorithm specific data paths," in *ASAP*, 1992, pp. 134 –148.

[10] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proc. CASES*, 2011.

[11] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. FPT*, 2002, pp. 166 – 173.

[12] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures," in *Proc. CASES*, 2006, pp. 136–146.

[13] A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template," in *IPDPS 2007*, 2007, pp. 1 –8.

[14] J. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "Spkm : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proc. ASPDAC*, 2008, pp. 776 –782.

[15] ExPRESS, "Electrical computer engineering dep., ucsb, usa," http://express.ece.ucsb.edu/benchmark/.

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

[17] S. Wong and F. Anjam, "The delft reconfigurable vliw processor," in *17th International Conference on Advanced Computing and Communications (ADCOM)*, 2009.

[18] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proc. LCTES*, 2008, pp. 151–160.