CREPE: Concurrent Reverse-Modulo-Scheduling and Placement for CGRAs

Chilankamol Sunny, Satyajit Das, Kevin J M Martin, Philippe Coussy

CREPE: Concurrent Reverse-modulo-scheduling and Placement for CGRAs

Chilankamol Sunny*, Satyajit Das*, Kevin J. M. Martin† and Philippe Coussy†
* IIT Palakkad, Palakkad, Kerala, India, † Université Bretagne Sud, Lab-STICC UMR 6285, Lorient, France
Email:112004004@smail.iitpkd.ac.in, satyajitdas@iitpkd.ac.in, {kevin.martin, philippe.coussy}@univ-ubs.fr

IEEE Transactions on Parallel and Distributed Systems
Print ISSN: 1045-9219
Online ISSN: 1045-9219
Digital Object Identifier: 10.1109/TPDS.2024.3402098

***Abstract*—Coarse-Grained Reconfigurable Array (CGRA) architectures are popular as high-performance and energy-efficient computing devices. Compute-intensive loop constructs of complex applications are mapped onto CGRAs by modulo-scheduling the innermost loop dataflow graph (DFG). In the state-of-the-art approaches, mapping quality is typically determined by initiation interval (*II*), while *schedule length* for one iteration is neglected. However, for nested loops, *schedule length* becomes important. In this paper, we propose CREPE, a Concurrent Reverse-modulo-scheduling and Placement technique for CGRAs that minimizes both *II* and *schedule length*. CREPE performs simultaneous modulo-scheduling and placement coupled with dynamic graph transformations, generating good-quality mappings with high success rates. Furthermore, we introduce a compilation flow that maps nested loops onto the CGRA and modulo-schedules the innermost loop using CREPE. Experiments show that the proposed solution outperforms the conventional approaches in mapping success rate and total execution time with no impact on the compilation time. CREPE maps all kernels considered while state-of-the-art techniques Crimson and Epimap failed to find a mapping or mapped at very high *II*s. On a 2×4 CGRA, CREPE reports a 100% success rate and a speed-up up to 5.9× and 1.4× over Crimson with 78.5% and Epimap with 46.4% success rates respectively.**

***Index Terms*—coarse-grained reconfigurable array (CGRA), modulo-scheduling, loop optimization.**

I. Introduction

In the quest for an efficient yet programmable hardware architecture, CGRA (Coarse-Grained Reconfigurable Array) architectures have been intensively studied as they offer the best trade-off from an energy efficiency point of view [1], [2]. A generic CGRA consists of an inter-connected array of processing elements (PEs), each PE having a functional unit (FU), context register (CR), register file (RF), and I/O interconnects. Compute-intensive loop constructs of complex applications are offloaded to CGRAs to improve the overall performance of the application [2]. This is done by configuring the PEs and interconnects every cycle based on the context prepared by the compiler [3]. The extent to which an application benefits from the efficiency of the CGRA hardware depends on the efficacy of its compiler to map the loop kernels of the application onto the CGRA. For a spatio-temporal CGRA that maps applications in space and time dimensions, the mapping process must solve both scheduling and binding (also called placement and routing) problems [4].

Software pipelining is a compile-time optimization technique that improves performance by overlapping the execution of different iterations of a loop. The most acclaimed and widely used software pipelining technique in the CGRA domain is modulo-scheduling [4]–[6]. Its goal is to find a schedule of operations from different iterations of the innermost loop that can be repeated in a given interval called initiation interval (*II*), expressed in cycles. Basically, *II* is the number of cycles between the start of two consecutive iterations of the innermost loop [7]. It can also be seen as the length of the repeating schedule. There exists a lower bound for the II, called minimum *II* (*MII*), for which a modulo schedule exists for a given target architecture, determined by two constraints: the resources, and the recurrence. *MII* is computed as *MII = max(ResMII, RecMII)* where *ResMII* is the resource-constrained *MII*, computed as the ratio of the number of operation nodes in the loop DFG to the number of resources in the CGRA. *RecMII* is the recurrence-constrained *MII* that depends on the recurrence dependencies that exist in the loop [8]. The best achievable goal is to find a modulo schedule with *II* equal to *MII*. If such a schedule cannot be found, *II* is incremented and the algorithm tries again until a valid schedule is obtained. The data flow graph (DFG) formed by the repeating schedule of operations is referred to as modulo-DFG (*MDFG*). The set of operations that are executed before and after the repeating schedule is called *prologue* and *epilogue* respectively. Most works on modulo-scheduling consider *II* as the key performance indicator (KPI) [4]. However, real-world applications often feature complex nested loops and the total execution time depends also on the *schedule length* for one iteration of the innermost loop, which relates to the *prologue* and *epilogue* cycles as well as *II*. *Schedule length*, also called *iteration latency* is the length of the schedule for one iteration of the loop [8]. In this paper, we introduce an agile approach

to modulo-schedule and bind the innermost loop DFGs on CGRAs, focusing on minimizing the overall execution latency of the application. This is achieved by better solution space exploration compared to the state-of-the-art approaches. As a result, the proposed mapping finds the best IIs on smaller CGRAs (less resource) while the existing approaches use bigger CGRAs to find similar solutions.

The contributions of the paper are as follows:

- We propose CREPE, a **C**oncurrent **Re**verse-modulo-scheduling and **P**lac**e**ment mechanism that minimizes both *II* and *schedule length* of the innermost loop. To the best of our knowledge, CREPE is the first modulo-scheduling technique that performs simultaneous (modulo-)scheduling and placement by traversing the DFG in reverse topological order, coupled with dynamic graph transformation for better exploring solution space.
- We present a compilation flow that maps an arbitrary number of nested loops onto CGRAs with the innermost loop modulo scheduled. The compilation flow supports both hardware-based (employing hardware loop units) and software-based (executing loop control instructions) loop implementations.
- We perform an extensive set of experiments presenting the mapping quality and design space exploration capability of CREPE compared to the state-of-the-art methods. Results show that the proposed approach outperforms the state-of-the-art solutions regardless of the type of loop implementation (hardware/software-based). CREPE is compared with two state-of-the-art solutions, Epimap [7] and Crimson [9] by executing kernels with varying DFG sizes on different CGRA configurations. CREPE reported the least execution latencies and was able to find mapping solutions for all the kernels and configurations considered whereas Crimson and Epimap failed to map large DFGs.

The rest of the paper is organized as follows. Section II presents the related works on modulo-scheduling and explains the motivation behind this work. Section III is dedicated to detailing the proposed approach and illustrating its efficacy. Section IV discusses experimental results and section V concludes the paper.

II. Background and Motivation

Numerous modulo-scheduling techniques have been proposed to optimize loop execution on CGRAs [7], [9], [10], [12]–[15], [18]–[20]. Iterative modulo-scheduling (IMS) [8], the seminal work on modulo-scheduling, introduces a simple extension of the acyclic list scheduling algorithm with a height-based priority function to efficiently schedule the innermost loop. It first computes the theoretical best value for *II* called minimum initiation interval (*MII*) [8], by considering the recurrence and resource constraints imposed by the input loop DFG and the target architecture. Then the IMS algorithm is iterated with successively larger values of *II*, starting with *MII*, until the loop has been scheduled. Edge-centric Modulo Scheduling (EMS) [20] focuses on the routing problem as its primary objective. Placement is a by-product of the routing process and the schedule is developed by routing each edge in the DFG. Experimental results show that EMS outperforms IMS by 25% as the lack of a global resource management strategy causes frequent routing failures in IMS, leading to mappings with increased *II*s.

Epimap [7] defines the mapping problem as a graph epimorphism problem. It pre-processes the input DFG and finds the *MII* of the transformed graph which helps in reducing the mapping search space. Placement is done by finding a maximum common subgraph (*MCS*) between *MDFG* and time-extended CGRA (*TEC*) graph. The time-extended graph is prepared by replicating the nodes in the CGRA multiple times. For every pair (u, v) of connected nodes in the CGRA, there is an arc from (replication of) u at time t to (replication of) v at time $t+1$. Note that every node in the CGRA is connected to itself [10]. Epimap enables flexible mapping options by using recomputing as well as routing of operation nodes. This helps Epimap obtain better performance results compared to EMS which relies solely on routing (stage re-assignment) to find the mapping solution. Epimap remains one of the best modulo-scheduling techniques in terms of the quality of mapping it generates [11]. REGIMap [10] defines mapping as a simultaneous placement and register allocation problem. It uses register files to route dependencies. GraphMinor [11] formalizes the mapping problem as a restricted graph minor containment of the innermost loop DFG in the modulo routing resource graph (MRRG), representing the CGRA architecture. The efficient graph minor search algorithm with aggressive pruning and acceleration strategies finds mapping solutions with minimal compilation time. The scheduling quality of GraphMinor is claimed to be quite similar to that of Epimap. However, the recomputation concept in Epimap enables additional scheduling and routing options that can lead to better-quality solutions for certain kernels [11].

Gu et al in [16] present a heuristic two-stage stress-aware mapping algorithm that integrates the intra-kernel and inter-kernel stress optimization strategies into loop scheduling and binding procedures. The method guarantees an optimized performance comparable to EPIMap, in terms of II, while effectively reducing and balancing the stresses accumulated on PEs.

RAMP [12] explores various ways to map the data dependencies, before the scheduling step, to improve the mapping-ability and mapping quality. It models various routing strategies and one or more of these strategies are selectively applied to find a mapping solution for a given *II*. This enables RAMP to map those kernels that are not mapped by techniques like REGIMap which rely on one single strategy for routing. Zhao et al in [17] argue that temporal mapping (scheduling) should be paid more attention than spatial mapping (binding) to achieve better performance in minimal compilation time. The work integrates interconnection and computational constraints solving heuristics into the scheduling process to foresee failures at the spatial mapping stage and initiate rescheduling if necessary. In addition to the efficient scheduling technique, the approach employs spatial mapping with backtracking and reordering to achieve good success rates with lower IIs and compilation times compared to REGIMap and RAMP. Crim-

TABLE I
COMPARISON OF VARIOUS MODULO-SCHEDULING TECHNIQUES

Technique	Approach	Placement & Routing	Graph Transformation
Epimap [7]	schedule-then-place	MCS using Levi's algorithm	static
REGIMap [10]	schedule-then-place	constrained maximal clique	static
GraphMinor [11]	simultaneous scheduling and placement	graph minor	static
RAMP [12]	schedule-then-place	maximal clique	static
Crimson [9]	schedule-then-place	not mentioned	static
Dynamic-II [13]	schedule-then-place	not mentioned	static
PathSeeker [14]	schedule-then-place	MCS following reverse BFS	static
SAT_MapIt [15]	schedule-then-place	SAT-based formulation	not applicable
Gu et al [16]	schedule-then-place	modified maximal compatibility classes	not mentioned
Zhao et al [17]	schedule-then-place	cost function	static
CREPE [this paper]	simultaneous scheduling and placement	incremental version of Levi's algorithm following reverse topological order	dynamic

son [9] introduces randomness in iterative modulo-scheduling to explore the schedule space effectively. The technique generates different schedules on each invocation of IMS for a given or increased *II*. This improves the possibility of finding a mapping solution and enables Crimson to achieve better mapping success rates than RAMP and GraphMinor. PathSeeker [14] is a mapping technique that analyzes mapping failures and performs local adjustments to the schedule, achieving lower *II*s and compilation time compared to RAMP and GraphMinor.

SAT-MapIt [15] addresses the mapping problem through a SAT-based formulation where data dependency, architectural constraints, and schedule are expressed as boolean statements. It is the first exact solution to the modulo-scheduling problem. It can even support DFG sizes that were previously only managed via heuristics and achieves better quality mappings compared to RAMP and PathSeeker. One limitation of this technique is that it does not apply any routing strategy, leading to increased *II*s compared to the state-of-the-art techniques for certain kernels. Dynamic-II pipeline scheme [13] realizes a pipeline with variable *II* by accommodating multiple iterations of shorter length in one static configuration. This method is designed exclusively for loops with irregular branches. It can realize a pipeline with two *II* for true-path and false-path respectively in a static configuration and dynamically switch *II* at runtime. A hybrid compilation framework is also implemented to extract branch features and correctly choose the processing method to achieve a smaller average *II* and shorter execution latency compared to conventional approaches.

All these works are targeted to find mapping solutions for the innermost loop DFG, considering *II* as the main performance metric. However, the innermost loop execution latency (T_{Loop}) depends on both initiation interval (*II*) and *schedule length* of one iteration (*L*) [8]. Equation 1 presents the T_{Loop} formulation, where N is the innermost loop iteration count.

$$T_{Loop} = (N-1) * II + L \tag{1}$$

In the case of nested loops, the total execution latency (T_T_{Loop}) is computed as shown in equation 2

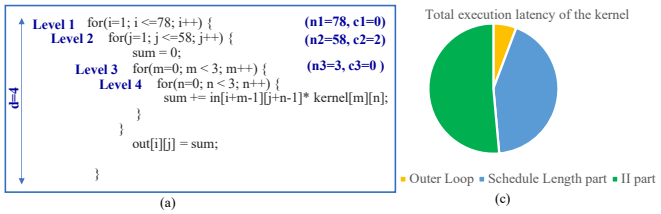
$$T\_T_{Loop} = T_{Loop} * \prod_{i=1}^{d-1} n_i + \sum_{j=1}^{d-1} c_j * \prod_{k=1}^{j} n_k \tag{2}$$

where d is the depth of loop nesting with the innermost loop at level d, c_j is the number of cycles incurred by the loop at level j for one iteration, excluding the cycles spent in its inner loops, and n_i and n_k are the iteration counts of loops at level i and k respectively. For perfectly nested loops in which assignment statements appear only in the innermost loop, the computation reduces to:

$$T\_T_{Loop} = T_{Loop} * \prod_{i=1}^{d-1} n_i \tag{3}$$

The impact of *schedule length* on the execution latency of the loop is greater when the number of iterations for the innermost loop is much smaller than the outer loops which is often the case. Fig. 1(a) gives the source code of a 2D convolution filter, that is widely used in signal processing and AI applications. The kernel features a loop structure with a very small iteration count for the innermost loop compared to the outer loops. Fig. 1(b) shows the computation of T_T_{Loop} assuming the ideal values for *II* and *schedule length*. T_T_{Loop} breakdown given in Fig. 1(c) shows that the *schedule length* affects the total execution time of the kernel significantly. Furthermore, minimizing the *II* may not always result in minimizing the *schedule length*. Fig. 2 (b) presents a mapping solution with modulo scheduling for the DFG and CGRA given in Fig. 2 (a). Fig. 2 (c) presents another valid mapping of the modulo scheduled DFG onto the CGRA. Here, the DFG formed by the repeating schedule of operations is different from that in Fig. 2 (b), resulting in a different *schedule length*, even though the length of the repeating schedule (i.e., *II*) is the same. This illustrates that it is possible to generate different mappings with the same *II* but different *schedule length*s. For this reason, optimizing the *II* alone is not sufficient to achieve the best performance.

Further, as shown in Table I, most of the modulo-scheduling techniques employ a schedule-then-place approach, performing resource-constrained (sometimes random-based) scheduling as a first step for mapping. Once the schedule is prepared, placement options are analyzed for predecessor nodes before



N	II	L	Time spent in executing the innermost loop		Time spent in executing the outer loops	T_TLoop
			II part	Schedule Length (L) part		
			$(N-1)*II*\prod_{i=1}^{d-1} n_i$	$L*\prod_{i=1}^{d-1} n_i$	$L\sum_{j=1}^{d-1} c_j * \prod_{k=1}^{j} n_k$	
3	3	5	81,432	67,860	9,048	158,400

(b)

Fig. 1. Motivating example demonstrating the impact of *schedule length (L)* on total execution latency (T_T_{Loop}). (a) Sample kernel with a loop nest structure of depth *d=4* and an innermost loop of iteration count *N=3*. The iteration count (n_1, n_2, n_3) of outer loops at each level as well as the number of cycles (c_1, c_2, c_3) exclusively incurred by those loops are shown; (b) Computation of T_T_{Loop} assuming an ideal mapping solution with *II* equal to *MII* (for a 2×4 CGRA) and *L* equal to the ASAP schedule length of the innermost loop; (c) T_T_{Loop} breakdown

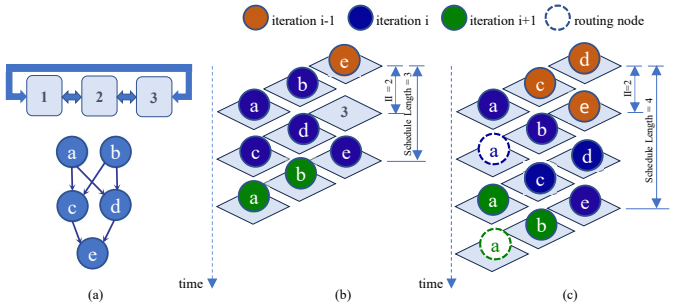


Fig. 2. (a) CGRA model and application DFG; (b) Mapping with *II* = 2 and *schedule length* = 3; (c) Mapping with *II* = 2 and *schedule length* = 4

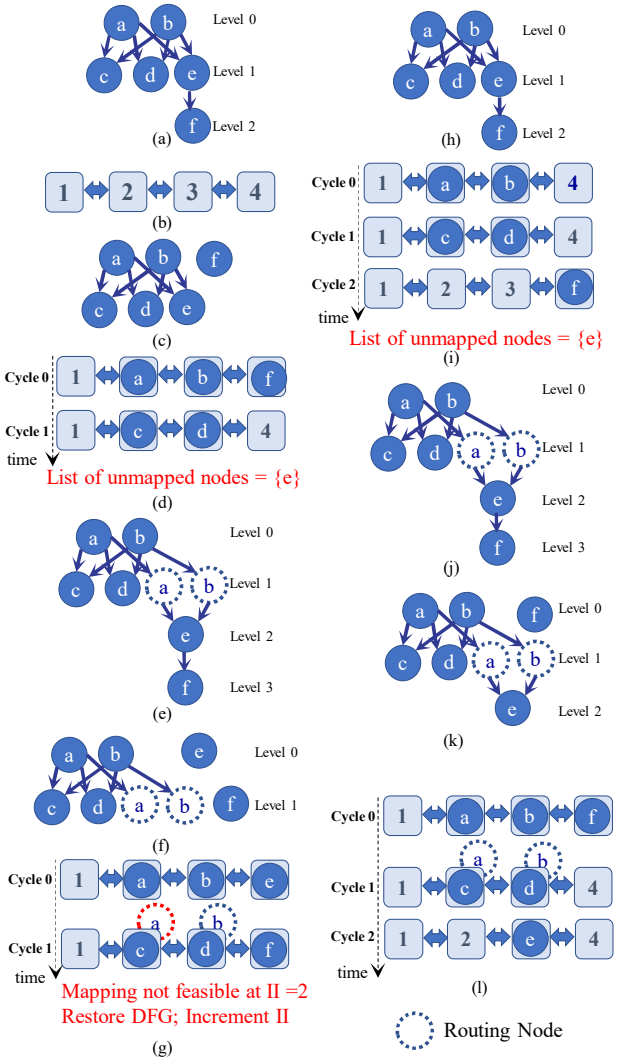


Fig. 3. Motivating example showing that schedule-then-place approaches employing static graph transformation and forward traversal of the DFG, like Epimap, results in mapping at increased *II*s. (a) Innermost loop DFG; (b) CGRA model; (c) MDFG constructed choosing *II=MII=2*; (d) Placement and Routing. Placement of *e* fails. (e) DFG after graph transformation. The input nodes of the unmapped node *e* (nodes *a* and *b*) are re-routed (through register file); (f) MDFG constructed from the transformed DFG with *II=MII=2*; (g) Mapping is found to be infeasible with this transformation as there exists no connection between the PEs to which node *e* and routing node of *a* are placed. Input DFG is restored and II is incremented; (h) MDFG constructed from the original DFG with *II=3*; (i) Placement and Routing. Mapping of *e* fails; (j) DFG after graph transformation. The input nodes of *e* are re-routed; (k) MDFG constructed from the transformed DFG with *II=3*; (l) Placement and Routing. Mapping is successful, reporting an $II > MII$.

the successor nodes. If no placement solution is found, then, static graph transformations are applied, *II* is increased and the scheduling process restarts from the beginning. They often cannot analyze data dependency accurately and tend to unnecessarily transform the loop DFG. This affects the *II* and *schedule length*, significantly increasing the total execution cycles of the complex loop nests of the application.

Fig. 3 illustrates the Epimap technique of modulo-scheduling and placement [7]. As the first step, *MII*, the minimum possible value of *II* is computed as the maximum of resource-constrained *MII* (*ResMII*) and recurrence-constrained MII (*RecMII*) [8]. The next step is MDFG construction with *II=MII*. Epimap creates the MDFG by assigning a level to each node in the DFG, which is given by the level of that node in the DFG modulo II. The algorithm then tries to find a placement solution for the MDFG by finding the maximum common subgraph (MCC) between TEC and MDFG using Levi's algorithm [21]. In case of failure, the algorithm collects the nodes left unmapped and transforms the DFG by re-routing or re-computing [7] the input nodes of the unmapped nodes and tries again until a valid mapping is obtained. Re-routing adds a routing node that makes explicit the storage of a value to delay one operation and to keep data dependencies. Re-computing duplicates an operation node by keeping its inputs the same and distributing output edges to reduce the number of successors of the original operation node. If a mapping solution cannot be found with current *II*, the algorithm re-stores the original DFG, increments *II*, and attempts for a new mapping. Fig. 3 depicts that the Epimap approach of scheduling followed by placement and routing by forward traversing the DFG pushed to map the DFG at an *II* greater than the *MII*. Schedule-then-place approaches show similar effects employing static graph transformations and forward traversal of the DFG. In this paper, we propose a simultaneous reverse-modulo-scheduling and placement approach

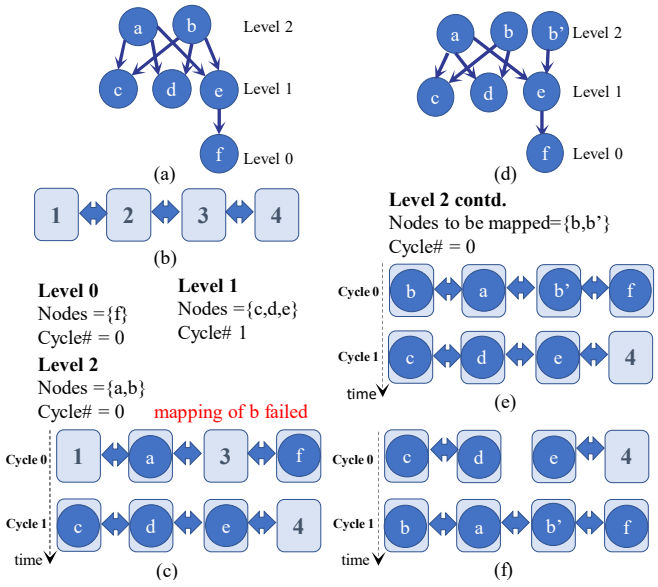


Fig. 4. Running example demonstrating the proposed CREPE technique of mapping. (a) Innermost loop DFG with levels assigned in reverse topological order; (b) CGRA model; (c) Simultaneous modulo-scheduling and placement by CREPE, following Algorithm 1. Starting from level *0*, nodes at each level are mapped at *cycle#* computed as *level#%II*, choosing *II=MII=2*. Placement of node *b* fails which calls for a dynamic graph transformation; (d) DFG after graph transformation. Node *b* is recomputed, adding node *b'* to the list of nodes to be mapped; (e) Mapping process continued. All nodes are successfully mapped. (f) Final *MDFG* mapping obtained by reversing the schedule. The schedule is reversed since it is prepared by traversing the DFG backward. Mapping is successful, achieving an *II* equal to *MII*.

with dynamic graph transformation that minimizes both *II* and *schedule length* of one iteration of the innermost loop. The detailed approach is presented in the next section.

III. Proposed Approach

In this section, we introduce CREPE, a **C**oncurrent **Re**verse-modulo-scheduling and **Pl**ac**e**ment method for CGRAs. We also propose a compilation flow that maps arbitrarily nested *for* loops (the nesting structure is not confined to a predetermined number of levels or a finite count of loops at each level) onto the CGRA and modulo-schedules the innermost loop using CREPE. We present two variants of the compilation flow, designed separately for CGRAs employing hardware loops and for those supporting the traditional software-based method of executing loop control instructions as part of the kernel code.

A. Concurrent Reverse-modulo-scheduling and Placement (CREPE) Technique

Fig. 4 illustrates the proposed CREPE mapping technique with the DFG and CGRA model used in the motivating example in Fig. 3. The CREPE algorithm traverses the input DFG in reverse topological order and maps the nodes at each level. A node appearing at a particular level is modulo-scheduled by assigning it to a cycle, computed as $cycle = level\%II$. Once the cycle is identified, the algorithm tries to place the node, respecting the mappings already generated for that cycle. If it fails to find a placement for the node, it transforms the DFG dynamically to improve the placement options. Here in this example, CREPE could map the *MDFG* at an *II* equal to *MII* which is 2 while Epimap mapped it at an *II* equal to 3; details will be given in the next subsections. The simultaneous reverse-modulo-scheduling and placement approach, coupled with dynamic graph transformation helped CREPE achieve better *II* results by avoiding unnecessary graph transformations.

1) CREPE Algorithm Overview: Algorithm 1 presents the CREPE mapping algorithm which generates the *MDFG*, *prologue*, and *epilogue* mappings for the innermost loop. The algorithm takes the innermost loop DFG as well as the CGRA model as inputs. It traverses the DFG in reverse topological order (starting at the leaf node level) and ensures that a node is mapped only after all its successor (according to the data dependencies) nodes are mapped. This helps to accurately analyze dependencies and efficiently explore the solution space [22]. Once the list of operation nodes appearing at a particular level of the DFG is prepared, it modulo-schedules and binds each node in that level to the time-extended CGRA (TEC) model.

If a mapping solution cannot be found for any of the nodes, the DFG is transformed dynamically. If such a transformation is not feasible, DFG is restored and mapping is restarted with an incremented *II*. This repeats until all nodes in the DFG are mapped, forming the *MDFG* mapping. The algorithm avoids performing the complex mapping operation for the *prologue* and *epilogue* parts of the loop. Instead, the mappings are prepared from the *MDFG* mapping and ASAP schedule of the input DFG (see sub-section 3). This saves the compilation time. Moreover, the good-quality *MDFG* mappings that CREPE generates result in low-latency *prologue* and *epilogue* parts, ensuring optimized *schedule length*s.

2) MDFG Mapping: As the first step, the mapping algorithm computes *MII*. *MII* is the resource-constrained *MII* (*ResMII*) or recurrence-constrained MII (*RecMII*), whichever is larger [8]. It then attempts to modulo-schedule and place and route the DFG onto the given CGRA model with this *II* (line 3-53).

The function *getNodesByRevTopSort* returns the list of nodes appearing at a given level of the DFG by traversing the graph in reverse topological order (line 10). All nodes appearing at a particular level are assigned to the same cycle. Modulo-scheduling is done by setting $cycle = level\%II$. The list of nodes returned by *getNodesByRevTopSort* is then sorted based on the mobility and fanout of the nodes. The algorithm chooses the first node in the sorted list and checks whether all its successor nodes are mapped or not (line 14,15). If any of the successor nodes are not mapped, then the chosen node is re-routed and removed from the list of nodes to be mapped in the current cycle. The list is updated to include the newly created routing node and sorted again (line 37-41). If all the successor nodes are mapped, the algorithm uses the function *findPnRByIncLevi (find Placement and Routing By Incremental Levi)* to find a placement solution for the chosen node, respecting the mappings already generated for the current cycle (line 16). *findPnRByIncLevi* implements an incremental version of Levi's algorithm [21] to place and route the node and applies a stochastic pruning on the partial map-

Algorithm 1: Concurrent Reverse-modulo-scheduling and Placement

```
   Input: DFG D, CGRA Model CM, MII
   Output: Prologue Mapping PM, MDFG Mapping MM, Epilogue
     Mapping EM
1  MII=computeMII(D,CM);
2  II=MII;
3  while true do
4      D'=D;
5      Mappings.reset();
6      backtrack=false;
7      NodestoBeMapped=getAllNodes(D');
8      level=0;
9      while NodestoBeMapped ≠ ∅ do
10         NodestoBeMappedCurrCycle=getNodesByRevTopSort(D',level);
11         NodestoBeMappedCurrCycle.sortByPriority();
12         cycle=level%II;
13         while NodestoBeMappedCurrCycle ≠ ∅ do
14             N=getFirstNode(NodestoBeMappedCurrCycle);
15             if isMapped(N.successorNodes) then
16                 P=findPnRByIncLevi(N,Mappings[cycle],D',CM);
17                 if P is null then
18                     newNodes=doRouteOrRecompute(N,D');
19                     if newNodes is null then
20                         II=II+1;
21                         backtrack=true;
22                         break;
23                     else
24                         NodestoBeMappedCurrCycle.add(newNodes);
25                         NodestoBeMapped.add(newNodes);
26                         if N.reRouted() then
27                             NodestoBeMappedCurrCycle.remove(N);
28                         end
29                         NodestoBeMappedCurrCycle.sortByPriority();
30                     end
31                 else
32                     Mappings[cycle].add(P);
33                     NodestoBeMappedCurrCycle.remove(N);
34                     NodestoBeMapped.remove(N);
35                 end
36             else
37                 newNode=doRoute(N,D');
38                 NodestoBeMappedCurrCycle.remove(N);
39                 NodestoBeMappedCurrCycle.add(newNode);
40                 NodestoBeMapped.add(newNode);
41                 NodestoBeMappedCurrCycle.sortByPriority();
42             end
43         end
44         if backtrack then
45             break;
46         end
47         level++;
48     end
49     if NodestoBeMapped = ∅ then
50         MM=Mappings.reverseCycle();
51         break;
52     end
53 end
54 AS=getASAPSchedule(D);
55 (PM,EM)=Prepare_Prologue_and_Epilogue_Mappings(AS,MM,MII);
56 return (PM,MM,EM);
```

ping set to prevent it from growing exponentially. The function finds the placement solution by choosing a PE from a set of candidate PEs. The candidate list preparation algorithm makes the CREPE technique suitable for heterogeneous CGRAs. A PE is chosen as a candidate PE to place an operation node only if it supports the functionality of the operation node.

If a placement solution is not found for the current node, the DFG is transformed dynamically, i.e., graph transformation is applied (line 18) before attempting to map the next node. Re-routing and re-computation are the two transformations we employ. The function *doRouteOrRecompute* creates either routing nodes or recompute nodes whichever is applicable, for the failed node. The newly created nodes are added to the list of nodes that need to be mapped in the current cycle. If the node that could not be placed is re-routed in the transformation step, then it is removed from this list. Then the set of nodes to be mapped in the current cycle is sorted again (line 24-29). If a valid transformation is not possible, the algorithm restores the original DFG, increments *II*, and starts again all over again (lines 20-22, 45, 3). This is repeated until all nodes in the DFG are modulo-scheduled and placed (line 9). Once this mapping is completed, the schedule of the mapping solution is reversed since the modulo-schedule is prepared by traversing the DFG backward (line 50). This forms the *MDFG* mapping. The algorithm then proceeds to prepare mappings for the *prologue* and *epilogue* parts of the loop by invoking the function $Prepare\_Prologue\_and\_Epilogue\_Mappings$.

3) Prologue and Epilogue Mapping: The pseudocode of the function, $Prepare\_Prologue\_and\_Epilogue\_Mappings$ is presented in Algorithm 2. The algorithm takes *MDFG* mapping (MM) and ASAP schedule (AS) of the input DFG as inputs. It maintains a schedule pointer *sPtr* and three cycle counters, *mCycle*, *pCycle* and *eCycle* corresponding to the *MDFG*, *prologue* and *epilogue* mappings respectively. The algorithm creates multiple copies of MM from which the *prologue* and *epilogue* mappings are composed, by traversing the MM multiple times. The number of copies to be created depends on the number of times the innermost loop gets unrolled (unroll factor) while preparing the modulo-schedule. The unroll factor is computed as the ratio of the *schedule length* and *II* [23]. The function *getScheduleLength* takes AS and MM as inputs and returns the *schedule length* of one iteration of the loop (line 2). The algorithm traverses the MM $(scheduleLength/II) - 1$ times (denoted by *count* in the algorithm) to create the required copies.

In each round, the algorithm compares *ASAPHeight* of nodes that appear at each cycle of MM with the value of the schedule pointer *sPtr* (line 12). If *ASAPHeight* of a node is less than or equal to *sPtr*, mapping information of that node is used to prepare the *prologue* mapping and if it is greater, the mapping is used to prepare *epilogue* mapping. Mapping is prepared by copying the node's placement information from MM and schedule information from the *prologue* / *epilogue* cycle counter maintained by the algorithm (line 17-18, 24-25). Schedule pointer *sPtr* is conditionally incremented (line 28,33) as it visits each cycle (*mCycle*) in the *MDFG* mapping.

4) Computational Complexity: The MDFG mapping algorithm consists of an iterative modulo scheduling (IMS) procedure and an incremental version of Levi's algorithm that finds the maximum common subgraph (MCS) between DFG and TEC. Both IMS and finding MCS are NP-complete problems that are solvable in non-polynomial time i.e., they have exponential time complexity. However, the empirical time complexity of IMS is $O(n^2)$ where n is the number of operation nodes in the DFG [8]. CREPE implements an incremental version of Levi's algorithm to place and route the node and apply stochastic pruning on the partial mapping set to prevent it from growing exponentially. The empirical time complexity of the placement procedure with stochastic pruning is $(n \times p)$ where p is the number of PEs in the CGRA. The algorithm that prepares prologue and epilogue mappings copies the mapping of each operation node in the

Algorithm 2: *Prepare_Prologue_and_Epilogue_Mappings*

Input: ASAP Schedule AS, MDFG Mapping MM, MII
Output: Prologue Mapping PM, Epilogue Mapping EM

```
1  sPtr=0; pCycle=0; eCycle=0;
2  count=ceil(getScheduleLength(AS,MM)/MII)-1;
3  II=MM.size(); //II is the schedule length of MDFG mapping
4  while count > 0 do
5      mCycle=0; pPrevC=0; ePrevC=0;
6      while mCycle < II do
7          incrementSptr=false;
8          i=0;
9          while i < MM[mCycle].size() do
10             N=getDFGNode(MM[mCycle],i);
11             ASAPHeight=getASAPHeight(AS,N);
12             if ASAPHeight <= sPtr then
13                 if mCycle!=pPrevC then
14                     pCycle++;
15                     pPrevC=mCycle;
16                 end
17                 PE=getCGRANode(N,MM[mCycle]);
18                 PM[pCycle].add(N,PE);
19             else
20                 if mCycle!=ePrevC then
21                     eCycle++;
22                     ePrevC=mCycle;
23                 end
24                 PE=getCGRANode(N,MM[mCycle]);
25                 EM[eCycle].add(N,PE);
26             end
27             if ASAPHeight%II >= mCycle OR
                 mCycle==getCycle(MM,N.getPredecessorNodes())+1 then
28                 incrementSptr=true;
29             end
30             i++;
31         end
32         if incrementSptr then
33             sPtr++;
34         end
35         mCycle++;
36     end
37     count = count-1;
38 end
39 return (PM,EM);
```

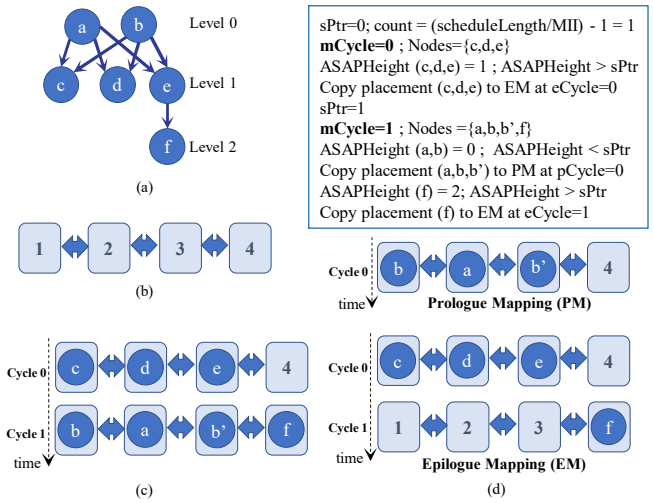


Fig. 5. Running example demonstrating the preparation of *prologue* and *epilogue* mappings by CREPE (a) ASAP schedule of innermost loop DFG; (b) CGRA model; (c) *MDFG* mapping (Refer Fig. 4); (d) *Prologue* and *epilogue* mappings prepared by replicating the *MDFG* mapping, following Algorithm 2. Mappings of the nodes mapped at cycle *0* (*c,d,e*) and node f mapped at cycle *1* are copied to *epilogue* mapping since their ASAP heights are greater than the schedule pointer (*sPtr*) value. Mappings of *a,b* and *b'* are copied to *prologue* mapping since the ASAP heights of *a* and *b* are less than the schedule pointer value.

MDFG multiple times as determined by the ratio of schedule length and II. The time complexity of this procedure is $O(n)$ as the number of operation nodes in MDFG is the number of operation nodes in the input DFG. Thus the statistical complexity of the CREPE mapping algorithm is $O(n^2)$ since no sub-routine's complexity is worse than $O(n^2)$, provided the number of operation nodes in the DFG (n) is greater than the number of PEs in the CGRA(p). When $p > n$, the complexity is $O(n \times p)$.

5) Running Example: Fig. 4 illustrates the *MDFG* mapping process of CREPE. The first step is to compute the *MII*. As there are no recurrence constraints in this example, *MII* is determined by the resource constraints i.e., *MII=ResMII*. *ResMII* is computed as the ratio of the number of nodes in DFG to the number of PEs in the CGRA. There are 6 nodes in the DFG and 4 PEs in CGRA, setting *MII* to 2. Next, the algorithm tries to modulo-schedule the DFG with *II* equal to *MII* and place and route it to the time-extended CGRA model (*TEC*). The algorithm traverses the DFG in reverse topological order, starting with the leaf node level. Modulo-scheduling of nodes appearing at a particular level is done by assigning them to a cycle computed as $cycle = level\%II$. Here, $level = 0$, $II = 2$ and $cycle$ is computed as $0\%2 = 0$. In this example, the only node at level 0 is f. The algorithm modulo-schedules and binds f by assigning it to cycle 0 and the PE denoted by 4 in the *TEC*. The list of mappings corresponding to cycle 0 is updated with this binding information.

Next, the list of nodes appearing at level 1 is selected and sorted based on the mobility and fanout of the nodes. The sorted list is $\{c, d, e\}$. The cycle number is computed as 1 $(1\%2)$. The algorithm chooses c, the first node from the list and modulo-schedules it by assigning it to cycle 1. Then it finds a placement solution for c, which is the PE denoted by 1 in the *TEC*. Similarly, node d is mapped to cycle 1 and PE 2. Next, e is mapped, considering the routing options from PE 4 to which its child node f is assigned. These mapping details are added to the mapping set corresponding to cycle 1. Moving to level 2, the algorithm identifies the list of nodes as $\{a, b\}$. The cycle to which these nodes are to be scheduled is computed as 0 $(2\%2)$. Placement of these nodes is to be done respecting the mappings already generated for cycle 0. The first node in the list, a is chosen and assigned to cycle 0. The next step is to find a placement solution for a, considering the routing options from the PE nodes in the *TEC* to which its child nodes (c, d, e) are mapped. PE 2 is chosen and node a is assigned to it. Coming to node b, the algorithm fails to find a placement solution in cycle 0. Indeed, PE 2 is also the only solution for node b, but already occupied by node a. This calls for a dynamic graph transformation.

Recall that the two graph transformations CREPE employs are re-computation and re-routing. It is worth noting that re-routing often tends to increase the latency and hence it is wise to choose recomputation whenever possible. Recomputation is feasible if the number of unoccupied PE nodes in a particular cycle in the *TEC* is more than the number of operation nodes left to be mapped in that cycle. Thanks to dynamic graph transformation, CREPE has the flexibility to choose between these transformations based on the current mapping

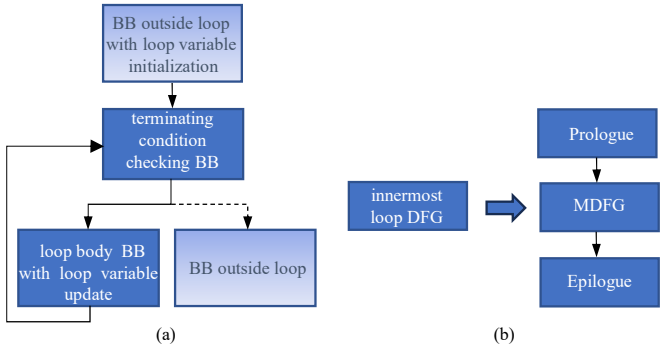


Fig. 6. (a) Generic CDFG representation of a *for* loop with no control statements in its loop body; (b) *MDFG*, *prologue*, and *epilogue* construction featured in CREPE mapping technique

status. Here, CREPE recomputes node b as the number of PEs left unoccupied in cycle 0 is more than the number of nodes that are yet to be mapped. The child nodes of b are distributed between b and b' in the transformed graph, relaxing the placement constraints. The newly created b' is added to the list of nodes to be mapped in level 2. Following the graph transformation, CREPE successfully maps the nodes in level 2. The schedule is reversed to obtain the final mapping solution as the mapping is done by traversing the DFG backwards. With this, the *MDFG* construction and its mapping onto the CGRA is completed, achieving an *II* equal to *MII* and a *schedule length* equal to the ASAP length of the input DFG.

Using the *MDFG mapping*, MM and ASAP schedule of the input DFG, the algorithm prepares the *prologue* and *epilogue* mappings as shown in Fig. 5. In this example, the algorithm creates one copy of the MM, as determined by the *schedule length* (3) and *II* (2), by traversing the MM once. The initial value of the schedule pointer, $sPtr$ is 0. The list of nodes in cycle 0 of MM is $\{c, d, e\}$. The ASAP height of all three nodes is 1, greater than the current value (0) of $sPtr$. This implies that these nodes are a part of the *epilogue*. The placement information of these nodes is copied from MM to the *epilogue* mapping at cycle 0. Before moving to cycle 1 of MM, $sPtr$ is incremented to 1, as per the algorithm. The nodes appearing at cycle 1 are b, a, b' and f. As the ASAP height (0) of the nodes a and b in the input DFG is less than the $sPtr$ value, their placement information is copied to the *prologue* mapping at cycle 0. The ASAP height of node f is 2, which is greater than $sPtr$. Hence, the placement details of f are copied from MM to *epilogue* mapping at cycle 1. Thus, the *prologue* and *epilogue* mappings are prepared from the *MDFG* mapping.

B. Compilation Flow

We introduce a compilation flow that maps arbitrarily nested loops onto the CGRA and modulo-schedules the innermost loop using CREPE. A variant that supports hardware-based loop implementation is also designed, in view of leveraging the benefits of both CREPE and hardware loop. We assume that the innermost loop does not contain any control statements in its loop body. State-of-the-art CGRA, IPA [24] is chosen as the platform to implement the proposed model. The kernel to be accelerated is represented as a set of single-entry-single-exit blocks of instructions called basic blocks (BBs). The set of BBs forms a control and data flow graph (CDFG) with nodes representing BBs and edges representing the control flow between them. Each BB is further represented as a DFG. IPA supports direct CDFG mapping to facilitate the execution of arbitrarily nested loops on the CGRA with no host intervention.

1) Proposed compilation flow for CGRAs supporting software-based loop implementation: A loop in the CDFG is identified as a subgraph with a back edge [25], as shown in Fig. 6(a). This means that the innermost loop comprises a set of DFGs while CREPE expects a single DFG as its input. The proposed solution addresses this issue by modulo-scheduling only the BB that represents the body of the innermost loop and by properly orchestrating the control flow between the BBs in the CDFG. Fig. 7(a) presents the proposed compilation flow for CGRAs with software-based loop implementation. The compilation process starts with generating a CDFG representation of the kernel written in C language, using a GCC plugin. The compiler selects each BB in the CDFG and maps it onto the time-extended CGRA model. If the chosen BB (represented as a DFG) corresponds to the innermost loop body, mapping is done by the CREPE module. The module takes the DFG and generates mappings for *MDFG*, *prologue*, and *epilogue* DFGs formed by modulo-scheduling the input DFG. The control flow between these three DFGs is as depicted in Fig. 6(b). The BB that does the terminating condition checking is mapped without applying modulo-scheduling. The control flow restructuring module in the compilation flow resets the control flow between these BBs as shown in Fig. 8 and ensures proper execution of the kernel code. Once all BBs are mapped, the compiler generates the assembly code for the whole CDFG mapping. The assembler converts the assembly code to a bitstream for the CGRA.

2) Proposed compilation flow for CGRAs supporting hardware-based loop implementation: The baseline IPA compilation flow that supports hardware-based loop execution [26] is extended to include modulo-scheduling of the innermost loop by CREPE as depicted in Fig. 7(b). The compilation flow involves a cyclic-to-acyclic graph transformation, an illustration of which is given in Fig. 9. The transformation removes the terminating condition-checking BB and the associated back edge from the CDFG, reducing the innermost loop to a single DFG, provided the loop has no control statements in the loop body. The next step is the BB selection. If the selected BB corresponds to the innermost loop, mapping is done by the CREPE module. Every other BB is mapped by the scheduling and placement module featured in the baseline compilation flow [27]. The CREPE module generates mappings for *MDFG*, *prologue*, and *epilogue* DFGs. The repeated execution of *MDFG* is handled by the hardware. Once all BBs are mapped, the compiler generates the assembly code and the assembler generates the bitstream.

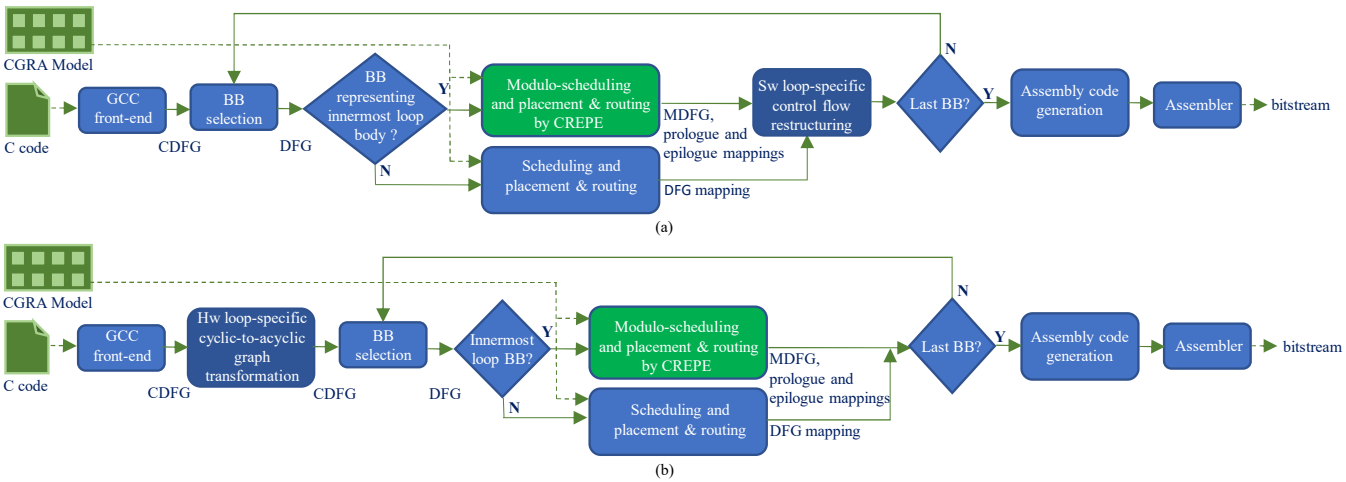


Fig. 7. Proposed compilation flow for (a) CGRAs supporting software-based loop implementation; (b) CGRAs supporting hardware-based loop implementation

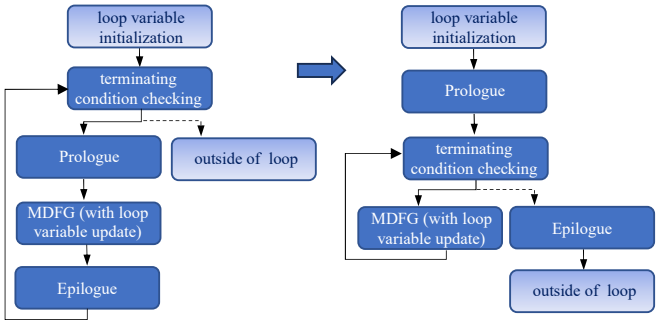


Fig. 8. Control flow restructuring in the compilation flow designed for modulo-scheduling software-based loops

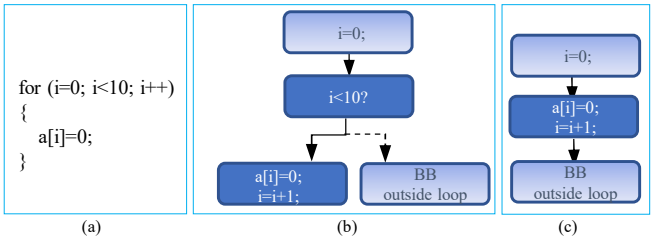


Fig. 9. (a) Sample *for* loop; (b) Corresponding CDFG; (c) CDFG after hardware-loop-specific cyclic-to-acyclic graph transformation

IV. Experiments And Results

A. Experimental Setup

The proposed mapping flow is fully automated through a software tool implemented using Java and Eclipse Modeling Framework (EMF). GCC is used to generate CDFGs from applications described in C language. A GCC plugin is developed to parse the intermediate representation (IR) of GCC and produce an equivalent IR in the Java world. Hence, all the optimizations GCC offers are included in the transformation including optimizations like function inlining. State-of-the-art Integrated Programmable Array (IPA) [24] is considered as the target CGRA. The IPA system is integrated into the PULP cluster, a near-threshold tightly-coupled cluster of RI5CY processors [28], as shown in Fig. 10. The SoC features a multilevel memory hierarchy. Targeting low-power embedded domains, an L1 scratchpad memory (TCDM) is employed instead of a data cache. The TCDM (tightly coupled data memory) is sized at 32 KB with 4 memory banks. The Global context memory of IPA is sized at 4 KB to fit the configuration data (i.e., instructions and constants). We consider 2×4, and 4×4 configurations of IPA with each PE having a 64×21-bits instruction memory, a 32×24-bits constant register file, and a 8×32-bits regular register file. The DMA controller loads data and instruction/context from the off-chip L2 memory.

All experiments have been performed on a post-placement-and-routing netlist. Execution latency results are collected using the cycle-accurate simulator, QuestaSim. Load and store operations incur two cycles each without stall whereas arithmetic operations (integer) take one cycle each.

The CGRA design is synthesized with Cadence Genus Synthesis Solution using 90nm CMOS technology libraries at 17.5 MHz frequency, 0.9 V supply voltage, in typical process conditions. Placement-and-routing is performed using Cadence Innovus and power analysis is done with Cadence Voltus. Energy results are computed using the switching activity obtained by simulating the placement-and-routed netlist.

We analyze the efficiency of CREPE against the state-of-the-art solutions, Epimap [7] and Crimson [9]. We chose Epimap and Crimson as they exhibit superior performance in terms of mapping quality and success rate respectively, over the earlier methodologies, discussed in Section II. We profiled a set of loop-intensive kernels including those from PolyBench [29] benchmark suite to analyze the efficiency of our approach. An extensive series of experiments is carried out with varying DFG sizes, obtained by unrolling the innermost loop DFG with different unroll factors. Table II presents the kernel statistics, such as the iteration count of the kernel, loop unroll factor, and the innermost loop DFG details such as ASAP length and the number of operation nodes after unrolling. $\#memnodes$ denotes the number of memory access (load/store) nodes and $\#nodes$ represents the total number of operation nodes including compute nodes and memory access

TABLE II
KERNEL STATISTICS

Kernel	Max Iteration Count	Unroll Factor	ASAP Length	#nodes	#memnodes
syrk	64x64x64=262 144	2	5	13	4
		4	7	23	8
		8	11	43	16
		16	19	83	32
		32	35	163	64
gemm	64x64x64=262 144	2	5	13	4
		4	7	23	8
		8	11	43	16
		16	19	83	32
		32	35	163	64
bicg	32x32=1 024	2	5	21	11
		4	7	37	19
2DConv	58x38x3x3=19 836	3 (1 loop full unroll)	5	15	6
		9 (2 loops full unroll)	12	40	19
sobel	62x62x3x3=34 596	3 (1 loop full unroll)	5	15	6
		9 (2 loops full unroll)	15	44	19
2DNon-sep	58x78x3x3 = 40 716	3 (1 loop full unroll)	5	15	6
		9 (2 loops full unroll)	14	42	19
matrixMul	32x32x32=32 768	4	6	19	8
		8	10	35	16
		16	18	67	32
matrixAdd	32x32=1 024	4	3	18	12
		8	3	34	24
		16	3	66	48
histogram	80x60=4 800	4	4	18	8
		6	4	26	12
		10	4	42	20
		15	4	62	30

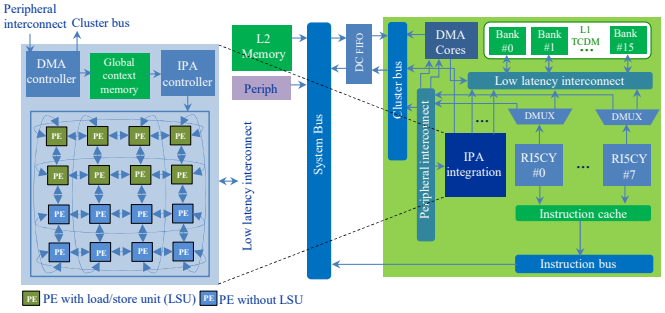


Fig. 10. IPA used as an accelerator in PULP SoC

nodes in the innermost loop DFG. The test cases are split into three categories that involve DFGs with less than 40 nodes (15 DFGs), 40 to 60 nodes (6 DFGs), and greater than 60 nodes (7 DFGs).

B. Performance results on IPA with software-based loop implementation

This section gives a performance comparison of CREPE with that of Epimap and Crimson in terms of mapping success rate and quality of mapping, on a 2×4 PE array configuration of IPA with 8 Load Store Units (LSUs).

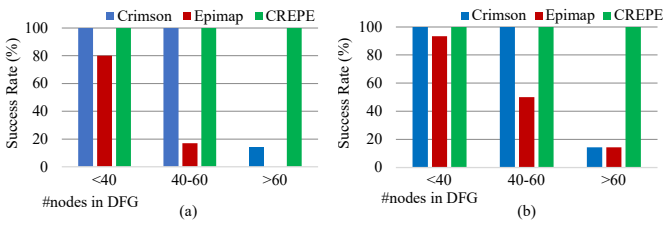


Fig. 11. Mapping success rate achieved by different approaches for varying sizes of innermost loop DFG on (a) 2×4 and (b) 4×4 CGRAs

*1) **CREPE achieves 100% success rate in mapping DFGs of varying sizes on constrained CGRA architectures (small PE array)**:* Fig. 11(a) presents the mapping success rates achieved by the three techniques for varying sizes of the innermost loop on a 2×4 CGRA. The success rate of an approach is defined as the percentage of times the approach finds a mapping solution. As done in Crimson experiments [9], if a valid mapping cannot be found even after increasing *II* up to 50 cycles, it is considered a mapping failure. The best of five runs are considered to collect the mapping results to nullify the effect of stochasticity in the placement process.

The results go with the findings of Crimson paper that IMS-based modulo-scheduling techniques (like RAMP [12] and Epimap) fail to explore the solution space efficiently and they tend to generate schedules that are not mappable even with increased *II*. Epimap could not map any of the kernels with DFGs having more than 60 nodes and the success rate reported for DFGs with 40 to 60 nodes is only 16.7%. Due to the static scheduling with resource constraints, solution space is limited as it often generates similar schedules with increased II. Crimson that employs randomized-IMS [9] for scheduling could map all kernels with less than 60 nodes in the innermost loop DFG. However, the success rate is only 14.3% in the category of DFGs with more than 60 nodes. As the DFG size increases randomized scheduling either fails to explore huge solution space efficiently or finds increased II with unnecessary static graph transformations. We discuss the effect of graph transformations in the later sections as well. Thanks to the simultaneous backward modulo-scheduling and placement approach, the proposed technique, CREPE can efficiently traverse the solution space and find valid mappings for all the different sizes of DFGs.

*2) **CREPE reports the least IIs and execution latencies for all kernels**:* Table III presents a comparison of the effective *II* values achieved by different approaches against *MII* for the considered kernels. An effective *II* is different from the mapping *II* in that the effective *II* value denotes the actual number of cycles between the launch of two consecutive iterations of the *MDFG*. This includes the cycles spent in implementing the loop control flow. Symbol X in the results table denotes a mapping failure. CREPE achieved lower *II*s compared to Crimson and Epimap for all kernels. Table IV gives the total execution latencies reported by Crimson, Epimap, and CREPE. In this comparison, we included only those kernels for which at least two of the techniques could find a mapping solution. It is observed that CREPE reports the least execution latency for all the kernels. It achieves an average of 1.4× and a maximum of 3.6× speed-up over Crimson, the technique that could map a comparable number of kernels.

C. Performance results on IPA with hardware-based loop implementation

In this section, we analyse the efficiency of our technique against Crimson and Epimap on CGRAs with hardware loop support. The results reflect the combined effect of modulo-scheduling and hardware-based loop implementation.

TABLE III

COMPARISON OF *II* (CYCLES) ACHIEVED BY CRIMSON, EPIMAP AND CREPE AGAINST MINIMUM INITIATION INTERVAL (*MII*) FOR VARIOUS KERNELS ON A 2×4 CGRA THAT EMPLOYS SOFTWARE-BASED LOOP IMPLEMENTATION

Kernel	**syrk**					**gemm**					**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**			**matrixAdd**			**histogram**			
Unroll Factor	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**16**	**4**	**8**	**16**	**4**	**6**	**10**	**15**
Crimson	6	9	15	X	X	6	9	14	X	X	8	11	6	20	6	30	7	24	8	12	X	7	10	X	7	9	11	15
Epimap	6	7	X	X	X	6	8	X	X	X	8	X	7	X	6	X	7	X	7	14	X	7	8	X	7	7	9	X
CREPE	5	7	11	19	35	5	7	11	19	35	6	9	6	8	6	7	6	9	7	11	19	6	8	12	6	7	9	11
MII	2	4	8	16	32	2	4	8	32	32	3	5	3	5	3	6	3	6	4	8	16	3	5	9	3	4	4	8

TABLE IV

TOTAL EXECUTION LATENCY (CYCLES) REPORTED ON A 2×4 CGRA WITH SOFTWARE-BASED LOOP IMPLEMENTATION

Kernel	**syrk**			**gemm**			**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**		**matrixAdd**		**histogram**			
Unroll Factor	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**4**	**8**	**4**	**6**	**10**	**15**
Crimson	1 098 705	988 164	824 392	1 102 801	861 188	832 592	8 088	5 803	94 066	72 145	89 386	247 565	114 994	285 087	137 954	129 830	4 064	3 477	14 456	14 322	10 284	9 721
Epimap	1 098 739	796 020	-	1 102 835	815 796	-	7 488	-	94 618	-	89 394	-	114 564	-	114 179	146 778	4 305	3 111	13 908	11 105	9 027	-
CREPE	984 026	791 539	715 823	979 938	795 652	713 817	6 520	4 890	90 168	34 230	89 352	69 535	104 968	85 542	112 346	97 070	3 296	3 084	11 968	8 882	8 453	6 993

TABLE V

COMPARISON OF *II* (CYCLES) ACHIEVED BY DIFFERENT APPROACHES AGAINST MINIMUM INITIATION INTERVAL (*MII*) ON 2×4 AND 4×4 CGRAS WITH HARDWARE-BASED LOOP IMPLEMENTATION

II Achieved on 2x4 CGRA with 8 LSUs																												
Kernel	**syrk**					**gemm**					**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**			**matrixAdd**			**histogram**			
Unroll Factor	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**16**	**4**	**8**	**16**	**4**	**6**	**10**	**15**
Crimson	3	5	11	X	X	3	6	11	X	X	5	15	4	18	3	30	4	34	4	9	X	4	12	X	4	5	11	12
Epimap	3	4	X	X	X	3	5	X	X	X	5	X	4	X	4	X	4	X	4	12	X	4	6	X	4	4	6	X
CREPE	2	4	8	16	32	2	4	8	16	32	3	6	3	6	3	7	3	6	4	8	16	3	6	9	3	4	6	8
MII	2	4	8	16	32	2	4	8	16	32	3	5	3	5	3	6	3	6	4	8	16	3	5	9	3	4	4	8
II Achieved on 4x4 CGRA with 8 LSUs																												
Kernel	**syrk**					**gemm**					**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**			**matrixAdd**			**histogram**			
Unroll Factor	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**8**	**16**	**32**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**16**	**4**	**8**	**16**	**4**	**6**	**10**	**15**
Crimson	2	4	10	X	X	2	4	8	X	X	3	X	3	15	3	21	3	22	4	10	X	3	5	X	2	4	5	15
Epimap	2	4	9	X	X	2	4	9	X	X	4	X	3	X	3	X	3	X	4	12	X	3	4	X	2	4	5	8
CREPE	2	4	8	16	32	2	4	8	16	32	3	6	3	4	3	4	3	4	4	8	16	2	4	9	2	2	4	8
MII	2	4	8	16	32	2	4	8	16	32	3	5	3	3	3	3	3	4	4	8	16	2	3	5	2	2	3	4

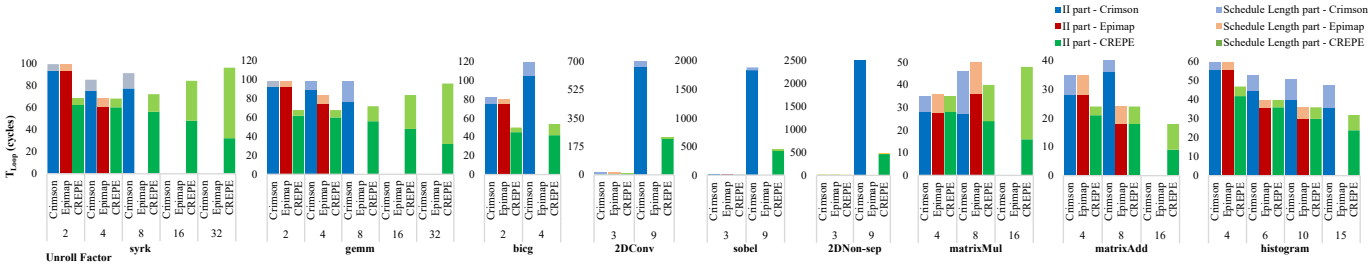


Fig. 12. Comparison of innermost loop latency (T_{Loop}) achieved by different approaches on a 2×4 CGRA

1) ***CREPE achieves the optimum II (MII) for majority of the kernels and the least schedule lengths for all kernels in the comparison****:* The comparison of *II* values (expressed in cycles) achieved by Crimson, Epimap, and CREPE against *MII* is presented in Table V. A mapping failure is denoted by the symbol X in the table and the same is highlighted in red colour. CREPE generates mappings with the least *II*s compared to Crimson and Epimap for all kernels. In the case of CGRAs that implement loop control on hardware, mapping *II* and effective *II* are the same, making it possible to achieve an effective *II* equal to the *MII*. The proposed approach achieves an *II* equal to *MII* (highlighted in green colour) for 82% of the kernels on 2×4 CGRA. On the other hand, Crimson and Epimap could achieve this for only 7% of the kernels. We discuss the performance results for 4×4 CGRA in section IV-C4.

Theoretical values of innermost loop execution latency (T_{Loop}) are computed by using equation (1). Results are given in Fig. 12 with *II* part and *schedule length* part stacked in each bar. Among the three approaches, CREPE reports the least *schedule length*s and *II*s, and consequently the least execution latencies. The T_{Loop} charts support the claim that the *schedule length* for one iteration of the innermost loop has a significant role in determining the execution latencies. It is also observed that the impact of *schedule length* grows with the unroll factor, except for 2DConv, sobel, and 2DNon-sep kernels. This is because, by partially unrolling the loop, the iteration count decreases with the increasing unroll factors.

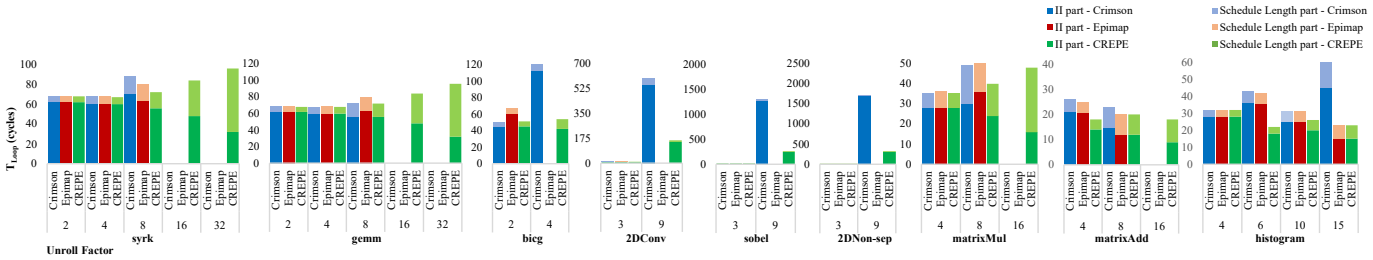


Fig. 13. Comparison of innermost loop latency (T_{Loop}) achieved by different approaches on a 4×4 CGRA

As discussed in section II, *schedule length* becomes more important when the innermost loop iteration count gets smaller. In the case of 2DConv, sobel, and 2DNon-sep kernels, two innermost loops are fully unrolled with a factor of 9. Hence, the resulting innermost loops with iteration counts 38, 62, and 78 respectively are the ones that got modulo-scheduled. Here, the innermost loop iteration counts are not small anymore compared to the number of iterations of the outermost loop. Consequently, *II* becomes more significant in the execution latency figures, compared to the *schedule length*.

The concurrent reverse-modulo-scheduling and placement approach, coupled with dynamic graph transformation helped CREPE track the dependencies in the DFG accurately and avoid unnecessary graph transformations. This led to the generation of good-quality mappings with lower execution latencies than the state-of-the-art works that follow the static graph transformation and schedule-then-place approach, in less compilation time. Table VIII presents the average number of graph transformations performed by each approach for different configurations. It can be noted that our approach keeps the number of graph transformations under control, which helps in managing the scalability. CREPE follows a "highly adaptable" approach by dynamic transformation vs "less adaptable" static graph transformation in state-of-the-art methods in the context of finding valid and good-quality mappings. Hence, the proposed method is agile in exploring architectural space and finding high-quality mappings with limited resources. Considering the kernels for which all three approaches could find a mapping solution, the average compilation time reported by Crimson, Epimap, and CREPE is 48.9 sec, 41.1 sec, and 38.2 sec respectively. The average compilation time that Epimap reported is found to be less than that of Crimson and CREPE since it could not map large DFGs which incur more time for mapping.

*2) **CREPE achieves up to 5.9× speed-up over Crimson**:* Table VI compares the total execution latencies reported by Crimson, Epimap, and CREPE on a 2×4 CGRA with 8 LSUs. As seen in Table V, Crimson failed to map kernels with very large DFGs and Epimap could not map even those kernels that Crimson mapped while the proposed CREPE technique mapped all kernels. The kernels for which at least two techniques could find a mapping solution are considered in this comparison. It is observed that CREPE reports the least execution latency among the three approaches, for all kernels and achieves up to 5.9× and 1.4× speed-up over Crimson and Epimap respectively.

*3) **The high-quality mapping generated by CREPE results in reduced energy consumption compared to Crimson and Epimap**:* The energy consumption (μJ) results by different approaches on a 2×4 CGRA are presented in Table VII. Results include the energy spent on the entire PE array including the LSUs interfacing with TCDM and the interconnects used for data transfer between PEs. As in the case of execution latency, we presented only the results for those kernels that are mapped by at least two of the three techniques we consider. Results confirm that CREPE achieves a considerable reduction in energy consumption compared to Crimson and Epimap by generating mappings with better *II*s and *schedule length*s.

*4) **CREPE explores the solution space better**:* We have seen that CREPE generates mappings that lead to good performance results on a small CGRA with 2×4 PE array as well as large application graphs where the conventional modulo-scheduling techniques fail to find a mapping or map at very high *II*s and latencies. Next, we analyze how well CREPE as well as the other two techniques we consider perform on a larger CGRA, by choosing an IPA implementation with 4×4 PE array and 8 LSUs for our experiments.

Fig. 11(b) presents the mapping success rate achieved by Crimson, Epimap, and CREPE on a 4×4 CGRA. Crimson mapped all kernels with less than 60 nodes in the innermost loop DFG. Epimap could map only 50% for DFGs with 40 to 60 nodes, and 93.3% of the DFGs with less than 40 nodes. For the DFGs with more than 60 nodes, the mapping success rate of both Crimson and Epimap is as low as 14.3%. As expected, CREPE achieved a 100% success rate in all categories. The comparison of *II* values achieved is presented in Table V and the estimated innermost loop execution latency is given in Fig. 13 with *II* part and *schedule length* part stacked in each bar. Table V shows that for some cases CREPE achieves similar II for 2×4 and 4×4 CGRA. Furthermore, most of these IIs are similar to MII. Crimson and EpiMap find the same solutions only for 4×4 configuration. As a result, the state-of-the-art methods require increased compilation time with a greater area footprint to achieve similar performance. This degrades the area and energy efficiency. These methods are unable to find solutions with fewer resources in contrast to CREPE due to inefficient solution space exploration. Further, CREPE improved its performance on 4×4 CGRA for DFGs (like matrix addition and histogram equalization) of which the *MII* is determined by resource constraints, confirming that CREPE scales well with resources.

Table VIII gives the average number of graph transforma-

TABLE VI
EXECUTION LATENCY (CYCLES) FOR DIFFERENT APPROACHES ON 2×4 AND 4×4 CGRAS WITH HARDWARE-BASED LOOP IMPLEMENTATION

Total Execution Latency Reported on 2x4 CGRA with 8 LSUs																						
Kernel	**syrk**			**gemm**			**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**		**matrixAdd**		**histogram**			
Unroll Factor	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**4**	**8**	**4**	**6**	**10**	**15**
Crimson	721 075	537 579	601 640	721 832	733 156	607 272	6 976	7 586	71 426	126 938	114 654	302 851	127 553	467 892	64 218	66 334	3 392	3 773	10 928	9 170	9 669	9 321
Epimap	594 865	476 123	-	590 835	537 572	-	6 047	-	69 216	-	123 930	-	154 672	-	66 009	74 808	3 512	2 643	9 488	7 180	6 827	-
CREPE	586 730	467 922	549 903	588 961	472 043	490 535	5 631	4 483	58 134	42 022	103 904	70 839	118 471	79 662	63 170	61 814	2 464	2 428	8 528	6 616	6 827	5 910
Total Execution Latency Reported on 4x4 CGRA with 8 LSUs																						
Kernel	**syrk**			**gemm**			**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**		**matrixAdd**		**histogram**			
Unroll Factor	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**4**	**8**	**4**	**6**	**10**	**15**
Crimson	459 896	468 140	594 655	447 843	598 107	500 878	5 357	8 309	64 801	119 698	127 837	212 851	118 530	312 451	66 169	70 561	2 973	2 786	7 917	7 975	7 373	12 433
Epimap	458 896	533 518	560 304	594 983	609 634	559 784	6 539	-	62 703	-	116 275	-	127 620	-	65 299	74 808	2 992	2 584	7 349	8 289	7 382	5 376
CREPE	453 867	455 468	549 903	437 963	472 043	490 535	5 112	4 483	56 103	40 816	102 495	44 112	117 610	57 422	61 321	61 814	2 119	2 106	7 307	5 400	5 948	5 153

TABLE VII
COMPARISON OF ENERGY (μJ) CONSUMED BY DIFFERENT APPROACHES ON 2×4 AND 4×4 CGRAS WITH HARDWARE-BASED LOOP IMPLEMENTATION

Energy Consumption Reported on 2x4 CGRA with 8 LSUs																						
Kernel	**syrk**			**gemm**			**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**		**matrixAdd**		**histogram**			
Unroll Factor	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**4**	**8**	**4**	**6**	**10**	**15**
Crimson	209.62	168.53	188.61	209.84	229.84	190.38	2.03	2.38	22.39	39.80	35.94	94.94	39.99	146.68	18.67	20.80	1.06	1.18	3.24	2.72	3.09	2.98
Epimap	172.93	149.26	-	171.76	168.53	-	1.76	-	21.70	-	38.85	-	48.49	-	19.19	23.45	1.10	0.83	2.81	2.13	2.18	-
CREPE	170.56	146.69	172.39	171.21	147.99	153.78	1.64	1.41	18.23	13.17	32.57	22.21	37.14	24.97	18.36	19.38	0.77	0.76	2.48	1.96	2.14	1.89
Energy Consumption Reported on 4x4 CGRA with 8 LSUs																						
Kernel	**syrk**			**gemm**			**bicg**		**2DConv**		**sobel**		**2DNon-sep**		**matrixMul**		**matrixAdd**		**histogram**			
Unroll Factor	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**3**	**9**	**3**	**9**	**3**	**9**	**4**	**8**	**4**	**8**	**4**	**6**	**10**	**15**
Crimson	239.07	243.36	309.13	226.42	315.01	263.80	2.82	4.38	34.13	63.04	67.33	112.10	59.79	164.56	33.38	35.59	1.50	1.41	4.00	4.03	3.73	6.29
Epimap	232.01	269.74	283.28	300.82	311.00	285.57	3.34	-	31.99	-	59.32	-	64.38	-	32.94	37.74	1.51	1.31	3.72	4.19	3.85	2.72
CREPE	229.47	230.28	278.03	221.43	238.66	248.01	2.58	2.27	28.37	20.64	51.82	22.30	59.33	29.03	30.93	31.18	1.07	1.06	3.69	2.73	3.01	2.61

TABLE VIII
COMPARISON BETWEEN THE NUMBER OF GRAPH TRANSFORMATIONS PERFORMED BY DIFFERENT APPROACHES

Average No. of Transformed Nodes on 2x4 CGRA			
#nodes in DFG	**<40**	**40-60**	**>60**
Crimson	41	156	151
Epimap	15	8	X
CREPE	1	1	2
Average No. of Transformed Nodes on 4x4 CGRA			
#nodes in DFG	**<40**	**40-60**	**>60**
Crimson	38	164	156
Epimap	19	50	4
CREPE	1	1	2

tions performed by each approach. The results testify that CREPE limits graph transformations to unavoidable situations. The average compilation time reported by Crimson, Epimap, and CREPE in generating mappings for 4×4 CGRA configuration is 228.4 sec, 73.5 sec, and 132.2 sec respectively. Table VI and Table VII respectively present a comparison of the total execution latencies and energy consumption (μJ) results reported by the three techniques. As in the case of 2×4 CGRA, CREPE reports the least execution latencies and energy consumption among the three approaches.

V. CONCLUSION

In this paper, we proposed an efficient mapping technique for CGRAs, named CREPE. Most of the existing techniques follow a schedule-then-place policy, perform placement by forward traversing the DFG, and employ static graph transformation. Such an approach often cannot analyze data dependency accurately and results in either a mapping failure or unnecessary graph transformations that affect the execution latency. CREPE performs simultaneous reverse-modulo-scheduling and placement, coupled with dynamic graph transformation that enables it to generate good-quality mappings with high success rates. We also introduced a compilation flow that maps arbitrarily nested loops onto the CGRA and modulo-schedules the innermost loop using CREPE. We presented the compilation flow that supports hardware-based and software-based loop implementations for CGRAs.

Experimental results show that i) the proposed CREPE technique finds a mapping solution on constrained architectures as well as large application graphs where the conventional modulo-scheduling techniques fail to find a mapping or map at very high *II*s and latencies. It reported a 100% mapping success rate for DFGs of varying sizes including those with more than 60 nodes where Crimson and Epimap failed to find mapping solutions for large DFGs. ii) CREPE outperforms the state-of-the-art solutions in terms of *II*, *schedule length*, execution latency, and energy efficiency with no impact on the compilation time. The agile mapping approach proposed in this paper explored the architectural space better, thanks to its ability to find accurate dependencies in the DFG, and resulted in good-quality mappings with fewer resources.

REFERENCES

[1] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3357375

[2] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[3] Z. Li, D. Wijerathne, and T. Mitra, "Coarse grained reconfigurable array cgra," *Book Chapter in Springer Handbook of Computer Architecture*, 2022.

[4] K. J. M. Martin, "Twenty years of automated methods for mapping applications on cgra," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 679–686.

[5] K. Choi, "Coarse-grained reconfigurable array: Architecture and application mapping," *IPSJ Transactions on System LSI Design Methodology*, vol. 4, pp. 31–46, 2011.

[6] J. a. M. P. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Comput. Surv.*, vol. 42, no. 4, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1749603.1749604

[7] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1284–1291.

[8] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 63–74.

[9] M. Balasubramanian and A. Shrivastava, "Crimson: Compute-intensive loop acceleration by randomized iterative modulo scheduling and optimized mapping on cgras," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3300–3310, 2020.

[10] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–10.

[11] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–25, 2014.

[12] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ramp: Resource-aware mapping for cgras," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[13] B. Yuan, J. Zhu, X. Man, Z. Ma, S. Yin, S. Wei, and L. Liu, "Dynamic-ii pipeline: Compiling loops with irregular branches on static-scheduling cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 9, pp. 2929–2942, 2021.

[14] M. Balasubramanian and A. Shrivastava, "Pathseeker: a fast mapping algorithm for cgras," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 268–273.

[15] C. Tirelli, L. Ferretti, and L. Pozzi, "Sat-mapit: A sat-based modulo scheduling mapper for coarse grain reconfigurable architectures," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[16] J. Gu, S. Yin, L. liu, and S. Wei, "Stress-aware loops mapping on cgras with dynamic multi-map reconfiguration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2105–2120, 2018.

[17] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.

[18] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *2003 Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 296–301.

[19] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3061639.3062262

[20] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 166–176.

[21] G. Levi, "A note on the derivation of maximal common subgraphs of two directed or undirected graphs," *Calcolo*, vol. 9, no. 4, pp. 341–352, 1973.

[22] T. Peyret, G. Corre, M. Thevenin, K. J. M. Martin, and P. Coussy, "Efficient application mapping on cgras based on backward simultaneous scheduling/binding and dynamic graph transformations," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 169–172.

[23] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 158–169, 1992.

[24] S. Das, K. J. Martin, D. Rossi, P. Coussy, and L. Benini, "An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1095–1108, 2018.

[25] J. Stanier and D. Watson, "Intermediate representations in imperative compilers: A survey," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, pp. 1–27, 2013.

[26] C. Sunny, S. Das, K. J. M. Martin, and P. Coussy, "Hardware based loop optimization for cgra architectures," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 17th International Symposium, ARC 2021, Virtual Event, June 29–30, 2021, Proceedings*. Springer, 2021, pp. 65–80.

[27] S. Das, K. J. M. Martin, P. Coussy, D. Rossi, and L. Benini, "Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 127–132.

[28] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè *et al.*, "Energy-efficient near-threshold parallel computing: The pulpv2 cluster," *Ieee Micro*, vol. 37, no. 5, pp. 20–31, 2017.

[29] L.-N. Pouchet and S. Grauer-Gray, "Polybench: The polyhedral benchmark suite, 2012," *URL http://www-roc. inria. fr/~ pouchet/software/polybench*, 2012.

Chilankamol Sunny is currently pursuing the Ph.D. degree in Computer Science at Indian Institute of Technology (IIT) Palakkad, Kerala, India. Her research interests include computer architecture, compilers, reconfigurable computing and low-power system design.

Satyajit Das is an Assistant Professor in the Department of Data Science, and Computer Science and Engineering at IIT Palakkad. He received his joint Ph.D. degree from the University of South Brittany, France, and the University of Bologna, Italy. Prior to joining IIT Palakkad, he was a postdoctoral fellow at Lab-STICC, UBS. His research spans the areas of Systems for AI, architecture, methods, and tools for low power systems, including CGRAs, custom processors, multi-cores, high-level synthesis, and compilers. The main focus of Dr. Das's research is to implement highly energy-efficient solutions for digital architectures in the domain of heterogeneous and reconfigurable multi-core System on Chips (SoCs).

Kevin J. M. Martin received a M.S. degree in electrical and computer engineering in 2004 and a PhD in computer science in 2010 from the Universite de Rennes, France. He is since 2011 an associate professor at Universite Bretagne-Sud in Lorient, France, in the Lab-STICC. His research interests stand at the crossing point between architecture, methods and tools, including but not limited to: custom processors, CGRA, multi-processor platforms, high-level synthesis, computer-aided design tools, compilers and software engineering.

Philippe Coussy is a full professor at the Université de Bretagne-Sud, France. He is Deputy-head of the Lab-STICC (UMR CNRS). He has been Vice-head of the Doctoral School Math-STIC, Head of the Master program in Electrical and Computer Engineering and Head of the Lab-STICC CACS Department. He is an elected member of the technical committee of the IEEE DISPS since 2011. His research interests include High-Level Synthesis and Coarse Grained Reconfigurable Architectures. He has organized several conferences, workshops and tutorials. He was guest editor for several special issues of scientific journals and co-editor of two books (Springer). He regularly serves as a (inter)national scientific expert and participates as PC member/ reviewer in many ACM/IEEE conferences/journals. He is Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) and he has been Associate Editor of the IEEE Signal Processing Letters (2013-2017).