CRIMSON: Compute-intensive loop acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping on CGRAs

Mahesh Balasubramanian, *Member, IEEE,* and Aviral Shrivastava, *Member, IEEE*

***Abstract*—CGRAs are emerging accelerators that promise low-power acceleration of compute-intensive loops in applications. The acceleration achieved by CGRA relies on the efficient mapping of the compute-intensive loops by the CGRA compiler, onto the CGRA architecture. The CGRA mapping problem, being NP-complete, is performed in a two-step process namely, scheduling and mapping. The scheduling algorithm allocates timeslots to the nodes of the DFG, and the mapping algorithm maps the scheduled nodes onto the PEs of the CGRA. On a mapping failure, the II is increased and a new schedule is obtained for the increased II. Most previous mapping techniques use the Iterative Modulo Scheduling algorithm (IMS) to find a schedule for a given II. Since IMS generates a resource-constrained ASAP (as-soon-as-possible) scheduling, even with increased II, it tends to generate a similar schedule that is not mappable. Therefore, IMS does not explore the schedule space effectively. To address these issues, this paper proposes CRIMSON, Compute-intensive loop acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping technique that generates random modulo schedules by exploring the schedule space, thereby creating different modulo schedules at a given and increased II. CRIMSON also employs a novel conservative test after scheduling to prune valid schedules that are not mappable. From our study conducted on the top 24 performance-critical loops (run for more than 7% of application time) from MiBench, Rodinia, and Parboil, we found that previous state-of-the-art approaches that use IMS such as RAMP and GraphMinor could not map five and seven loops respectively, on a 4×4 CGRA, whereas CRIMSON was able to map them all. For loops mapped by the previous approaches, CRIMSON achieved a comparable II.**

***Index Terms*—Compiler, Coarse-grained Reconfigurable Arrays, Modulo Scheduling, Randomized Scheduling.**

I. Introduction

COMPUTING demands in human society continue to climb. Today there are numerous devices that collect, process, and communicate data from multiple sources such as the Internet, Cyber-Physical and Autonomous Systems, sensor networks, etc. [1]. Extracting intelligent and actionable information from all these data – whether or not done by machine learning – is extremely compute-intensive, and often times limited by power, thermal, and other resource constraints [2]. Efficiency in the execution of these functionalities can be achieved by using Application-Specific Integrated Circuits (ASIC). However, they suffer from high production costs, and they quickly become obsolete as applications and algorithms evolve. Another promising alternative is Field Programmable Gate Arrays or FPGAs, but they lose efficiency in providing bit-level configurability, which is essential for their primary purpose – prototyping [3]. Coarse-grained reconfigurable architectures or CGRAs provide a very good middle ground with coarse-grain configurability (word and arithmetic operator-level), without much loss in power-efficiency when compared to ASICs [4]. As a result, there is a renewed surge in the application of CGRAs for compute-intensive workloads including machine learning, embedded systems, and vision functionalities [5]–[7].

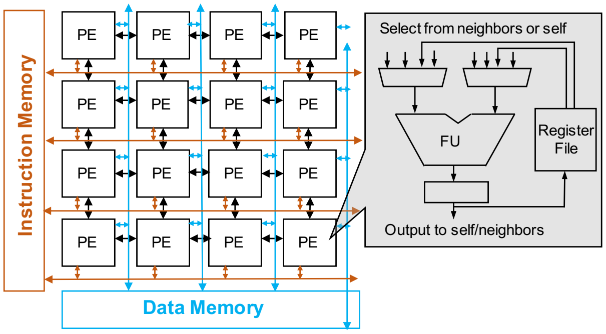


Fig. 1: A typical CGRA architecture consisting of 4x4 PEs connected in a 2D mesh. Every cycle, each PE gets an instruction from the Instruction Memory, and can operate on the outputs of neighboring PEs and/or the Data Memory.

As shown in Fig 1, CGRAs are simply an array of Processing Elements (PE) arranged in a 2-D grid. The PEs are just bare Arithmetic Logic Units (ALU) that can receive inputs from the neighboring PEs, from the *Data Memory*, and its own small set of registers. Every cycle, the PEs receive an instruction from the *Instruction Memory*, and write the results to the output buffer, local register file, and/or the data memory. CGRA-based execution is highly parallel (16 operations can be executed simultaneously on a 4×4 CGRA) and power-efficient because instructions are in the pre-decoded form. There is no extensive pipeline for instructions to go-through before and after execution, and the PEs can exchange operands directly rather than going through the register files. Some of the early works on CGRA architecture include ADRES [3], PADDI [8], Kressarray [9], MATRIX [10], Morphosys [11], and Remarc [12]. ADRES [3] which showed CGRAs to be promising power-efficient accelerators with power efficiency of 60 GOps/W using a 32 nm technology.

The most common way to use CGRAs is to employ them as co-processors to CPU cores or processors, to speed up and power-efficiently execute compute-intensive applications – similar to GPUs. The execution of compute-intensive loops in the application can then be“offloaded” onto these CGRA co-processors, while the rest of the application can still execute on the CPU. This heterogeneous computing paradigm requires compiler support to map compute-intensive loops of the application onto the PEs of the CGRA. Since the execution of a loop happens by software-pipelining on a CGRA, the objective of mapping is to layout the nodes of the DFG onto a graph of the CGRA extended in time, so that the nodes can exchange the operands through the interconnection among the PEs and achieve correct and efficient execution. The repetition interval of the mapping (the time at which the next iteration of the loop can start) is called the Initiation Interval (II) and is the metric that determines the quality of mapping. Many techniques have been proposed to solve NP-complete [13] mapping problem of CGRAs efficiently [13]–[20]. Most of the newer methods work in these four steps: i) create the Data Flow Graph (DFG) of the loop, and estimate the minimal II, ii) create the CGRA graph unrolled II times, iii) schedule the nodes of the loop onto the CGRA graph, and finally, iv) map the nodes onto the PEs at their scheduled timeslots such that the dependencies among the nodes can be transferred through the connectivity among PEs. In case a valid mapping is not found, the II is increased, and steps from ii) onward are executed again. This process is repeated until a valid mapping is found. A mapping failure can occur in the fourth step due to the limited connectivity among the PEs of the CGRA, and because of the need to map new routing nodes. Routing nodes occur when dependent operations are scheduled in non-contiguous timeslots. In this case, the operands need to be routed from the PE on which the source operand is mapped, to the PE on which the destination operation is mapped. This is commonly referred to as the routing problem. One solution is to route the operands through the PEs in the intermediate timeslots. Since routing and mapping attempts often fail, existing CGRA mapping techniques have heavily focused on solving the problem encountered in the mapping and routing step. For example, [16], [17] route dependencies via PEs, [19] routes dependencies through the registers inside the PEs, [18] overlaps the routing paths carrying the same value, and [13] uses recomputation as an alternative to routing. MEMMap [21] routes dependent operations via data memory by adding store and load nodes. RAMP [20] proposes a heuristic to explore all the different routing options. However, all the previous approaches use the same Iterative Modulo Scheduling (IMS) [22] to find a valid schedule – and therein lies the problem.

The problem with IMS is that it generates a resource-constrained, As Soon As Possible (ASAP) schedule of nodes onto the CGRA PEs. When a mapping is not found, the traditional mapping techniques increase the II, and return to the scheduling step. The generated schedule does not change much, even when more resources are added towards the bottom of the CGRA graph. The resource-constrained ASAP schedule will be almost identical to the one obtained before, and the extra resources are not used! As a result, the mapping algorithm keeps on exploring the schedule space with the same schedule, and often no mapping can be found, even after huge increases in the II. Table I shows the evaluation of the 24 performance-critical loops from MiBench, Rodinia and Parboil on a 4×4 CGRA, while being executed on the state-of-the-art IMS based mapping algorithms, GraphMinor [18] and RAMP [20]. We can see that state-of-the-art RAMP was unable to find a valid mapping for five loops and GraphMinor was unable to find a valid mapping for seven loops on evaluation up to a maximum II=50. One major observation was that, when these previous algorithms find a mapping, they achieve a very good II, but when the mapping fails, they are unable to map the loops even with II increments up to 50. For example in loop *jpeg1*, while the minII was 3, both the techniques were unable to map the loop, even when the II was increased to 50.

Suites	Loops	4x4		
		MII	RAMP-II	GraphMinor-II
MiBench	bitcount	3	3	6
	susan	2	3	3
	sha	3	3	3
	jpeg1	3	**X**	**X**
	jpeg2	2	**X**	**X**
Rodinia	kmeans1	2	2	2
	kmeans2	2	2	2
	kmeans3	2	2	2
	kmeans4	2	2	2
	kmeans5	2	2	2
	lud1	2	2	2
	lud2	2	2	2
	b+tree	2	2	2
	streamcluster	2	2	2
	nw	2	2	2
	BFS	2	2	2
	hotspot3D	5	**X**	**X**
	backprop	5	**X**	**X**
Parboil	spmv	3	3	3
	histo	2	2	2
	sad1	2	2	**X**
	sad2	2	2	2
	sad3	2	2	**X**
	stencil	4	**X**	**X**

TABLE I: On evaluating 24 applications of the top three benchmark suites on a 4×4 CGRA, we find that IMS-based RAMP was unable to map 5 of the loops and IMS-based GraphMinor was unable to map 7 of the loops. The “**X**” in the table denotes an II was not obtained even at a maximum II of 50. The MII in the table denotes the minimum II, which is the maximum of either ResMII or RecMII.

Thus, the main problem in IMS is the absence of randomness in the scheduling algorithm. As a result, even when the II is increased, the same schedule is generated without obtaining a valid mapping. Hence, this creates a need for an enhanced scheduling algorithm that explores the schedule space to increase the mappability of the compute-intensive loops. A more detailed explanation with a motivating example is given in Section IV. In this paper, we propose CRIMSON (**C**ompute-intensive loop acceleration by **R**andomized **I**terative **M**odulo **S**cheduling and **O**ptimized mappi**n**g on CGRA). Instead of just using the Resource constrained ASAP schedule, CRIMSON generates both the Resource Constrained As Soon As Possible (RC_ASAP) and Resource Constrained As Late As Possible (RC_ALAP) schedules for all the nodes of DFG, similar to the concept of mobility used in high-level synthesis (HLS) [23]. CRIMSON then chooses a random time between RC_ASAP

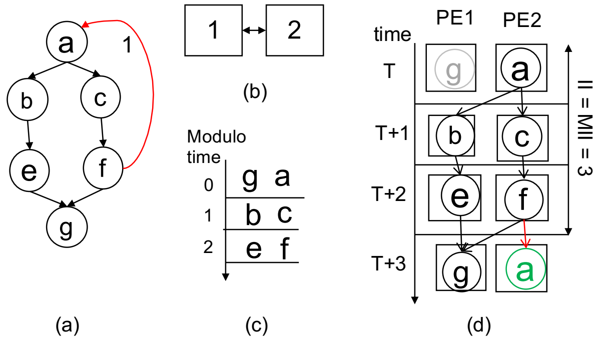


Fig. 2: (a) DFG of an application loop. (b) a 1x2 CGRA target architecture. (c) An IMS schedule of nodes of DFG. The X-axis is the modulo time. (d) A mapping of the scheduled nodes on the time-extended CGRA (TEC).

and RC_ALAP as the scheduling time for each node. As a result, every time a "new" schedule is obtained, CRIMSON is able to effectively explore the schedule space. CRIMSON also incorporates a novel conservative feasibility test after the scheduling step to check the mappability of the obtained schedule. This conservative test makes sure that the generated schedule will be mappable even after the addition of the new routing nodes, thereby rendering feasibility by quickly weeding out some unmappable schedules, and saving time. Among the 24 performance-critical (that account for more than 7% of execution time of the application) loops from MiBench, Rodinia, and Parboil, our approach CRIMSON was able to map all the loops for various CGRA sizes ranging from 4×4 to 8×8. Our approach CRIMSON achieved a comparable II for the loops which were mappable by RAMP.

II. Background and terminology

CGRA compilers in general first create the Data Flow Graph (DFG) $D = (V, E)$ of a compute-intensive loop, where V refers to the nodes of the loop and E refers to the edges (data dependencies between nodes) in the graph. The constructed DFG is then software pipelined using IMS [22], where each node is assigned a schedule-time at which it should be executed.

Fig 2(a) shows the DFG of a loop, and Fig 2(b) shows the target CGRA architecture. The schedule of the DFG nodes are shown in Fig 2(c), considering the resource and the recurrence cycle constraints. After scheduling, the nodes are then mapped onto the PEs of CGRA such that the dependent operands can be routed from the PE on which the source operation is mapped to the PE on which the destination operation is mapped through either registers, memory or paths in the CGRA graph. A register can be used to route operands when the dependent operation is mapped to the same PE as the source operation. Memory can be used to route operands, but that requires inserting additional load and store instructions. A path is a sequence of edges and nodes in the CGRA graph that connect two PEs. In the simplest case, a path is just a single edge.

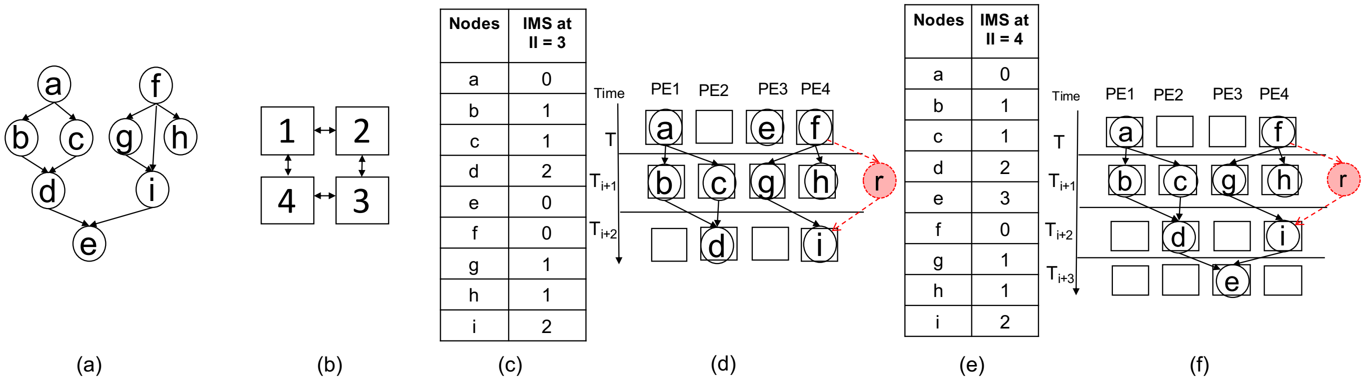
For simplicity, the mapping shown in figure 2(d) uses only edges to route dependencies. In this mapping, node a of iteration i (shown in dark color) is mapped to $PE2$ at time T, nodes b and c are mapped to PEs, $PE1$ and $PE2$ respectively, at $T+1$. Similarly, nodes e, f of i^{th} iteration are mapped in $PE1$ and $PE2$ respectively at $T+2$. Node g of i^{th} iteration is mapped at $PE1$ at $T+3$. It can also be observed that a and g are mapped at T and $T+3$ in $PE1$ and $PE2$. Modulo schedule repeats itself every II cycles, in this case II=3. The node g at T (shown in gray) is from $(i-1)^{th}$ iterations. Likewise, the node a mapped at $T+3$ is from $(i+1)^{th}$ iteration (shown in green). Based on the schedule, which considers the recurrences while scheduling, mapping a in $PE2$ satisfies the recurrence constraint of $f \xrightarrow{1} a$. i.e., the value of f at i^{th} iteration can be routed to a at $(i+1)^{th}$ iteration. In modulo scheduling, the interval in which successive instructions can begin execution is called the *Initiation Interval* (II) [22]. II is considered as the performance metric for DFG mapping onto CGRA, as the total number of cycles required to execute a loop will be proportional to the II.

III. Related Works

Coarse-Grain Reconfigurable Arrays or CGRAs have been a luring accelerator option owing to their high performance and high power-efficiency. The ADRES CGRA [3] demonstrated to operate at 60 GOps/W (Giga Operations per Watt). The high power-efficiency of CGRA is due to instructions being in pre-decoded format with no long pipelines before and after execution, and the fact that PEs can exchange operands directly without going through a centralized register file. CGRAs rely on the compiler to map loops onto the PEs. Some CGRA application mapping techniques use generic algorithms [24]–[26] like genetic algorithms or simulated annealing [14], [27], [28] to explore the various possible mappings and come up with a valid one. While these genetic algorithms and simulated annealing come with inherent randomness, these methods take exorbitantly long times to find a valid mapping, since they have no conception of the DFG and CGRA graph structures.

Some of the older application-specific compilation techniques like DRESC [14] attempt to solve the scheduling and mapping problems together in one shot. However, this is inefficient, since these algorithms may spend a lot of time exploring a mapping, when even the schedule is infeasible. A valid schedule is a pre-requisite of a valid mapping, and since scheduling is quite quick [22], it makes sense to first find a valid schedule, and then explore mapping solutions only for those schedules. As a result, most modern approaches separate the scheduling and mapping steps. When a mapping attempt fails due to limited connectivity or additional routing requirements, the II is increased, and a new schedule for this increased II is obtained followed by the another attempt on mapping. This scheduling and mapping is repeated until a valid mapping is obtained.

Since mapping is harder, previous works concentrate on solving the mapping and routing issues. EPIMap [13] uses recomputation of some nodes to solve the routing problem. REGIMap [19] uses register file in the PEs to route the



Nodes	IMS at II = 3
a	0
b	1
c	1
d	2
e	0
f	0
g	1
h	1
i	2

Nodes	IMS at II = 4
a	0
b	1
c	1
d	2
e	3
f	0
g	1
h	1
i	2

Fig. 3: (a) DFG of an application loop. (b) a 2x2 CGRA target architecture. (c) Column 1 shows the nodes in the DFG and Column 2 shows an IMS schedule for the nodes at II=MII=3. (d) The mapping algorithm tries to the map the nodes scheduled, but fails due to additional routing nodes "r" required to route nodes f and i. Failure to find a valid mapping, the II is increased to 4 and IMS is called again to schedule the nodes based on the workflow given in Fig 4. (e) IMS schedule for an increased II (II=4). (f) Even at an increased II, the mapping algorithm cannot find a valid mapping due to resource constraint at T_{i+1} which is not resolved at II=4 and will not be resolved on any further increase in II.

dependent operations, where as MEMMap [21] uses the data memory to route the dependencies. More recent techniques like, RAMP [20] presents a heuristic to choose among a variety of routing options to try for unmapped nodes, CASCADE [29], on the other hand, increases data throughput by decoupling the memory accesses and the execution. Even though all these techniques have different mapping and routing strategies, they use the same scheduling algorithm, namely - Iterative Modulo Scheduling, IMS [22] proposed by Rau et. al, for VLIW architectures, uses resource constrained As Soon As Possible (ASAP) approach to schedule the nodes of DFG. The problem is that, even when II is increased, IMS generates the same schedule, and is unable to explore the newly created scheduling space created by increased II.

Instead of taking a conventional ASAP/ALAP scheduling approach, EMS [16] proposes an alternative approach, where the nodes of the recurrent cycles are lifted or lowered on the time axis by assigning stages. These stages can consist of multiple schedule times. When an operation stage is reassigned based on the placement of its predecessor, all the dependent operation stages is also reassigned. However, while the EMS schedule is not ASAP, but it is still not randomized. As a result the generated schedule for a higher II is very similar (if not the same) to the generated schedule at lower IIs. HyCube [30] proposes a mapping technique for a highly connected CGRA that uses multi-hop multi-cast path system to communicate data in a single-cycle. In addition, HyCube's interconnect crossbar switch is a part of the ISA, which makes it power-efficient. Like DRESC [14] approach of integrated scheduling, placement and routing (P&R), HyCube's *ScheduleAndRoute* schedules and performs P&R in one shot. This faces the same issues as DRESC discussed above. Evidently, HyCube's single-cycle communication may provide better II, but at the cost of scalability. Since, the interconnect crossbar selection is a part of HyCube's instruction set, for higher CGRA sizes HyCube's instruction becomes longer.

The main contribution of this paper is a random iterative modulo scheduling algorithm to effectively search the scheduling space and an enhanced application mapping workflow to efficiently find a valid mapping of loops.

IV. Motivating Example

Let us consider the DFG of loop to be mapped on a 2×2 CGRA, shown in Fig 3(a) and (b), respectively. Previous state-of-the-art techniques like RAMP, get a schedule from IMS [22] before mapping the nodes. IMS starts by computing the resource constrained minimum II ($ResMII$) and recurrence constrained minimum II ($RecMII$) from the DFG and the architecture description. For the given example in Fig 3, total nodes $=$ 9 and total resources available $=$ 4. The minimum II (MII) is the maximum of $RecMII$ and $ResMII$. Therefore for the above example, *MII* = *ResMII* = $\lceil 9/4 \rceil$ = *3*. After computing the MII, IMS sets the priorities for each node. Priority is a number assigned to each node, which is utilized during scheduling. Based on the height of the node, from the given DFG, the deepest node is given the least priority using depth-first search strategy. For the loop DFG given in Fig 3(a), node e gets priority 0, nodes d and i get priority 1, nodes b, c, g, h get priority 2 and finally a and f get priority 3. The nodes with higher priority number are scheduled first with earliest start time. The modulo scheduling starts with II=MII for scheduling the nodes. The CGRA is time-extended, II times and a modulo resource table (MRT) is maintained to check for resource overuse for each timeslot. While trying to schedule each node, resource conflicts are checked. If there is a resource conflict a higher schedule time is tried. For the example DFG, the II=MII=3. Nodes a and f are scheduled at modulo time 0 (0%3). Nodes b, c, g, and h are scheduled at modulo time 1 (1%3) without any resource constraint because there are 4 resources(PEs) at each modulo time. Nodes d and i are scheduled at modulo time 2 (2%3). Finally, e is scheduled at modulo time 0 (3%3). The IMS schedule of nodes (shown in column 1 Fig 3(c)) at $II = 3$ is shown in Fig 3(c) column 2.

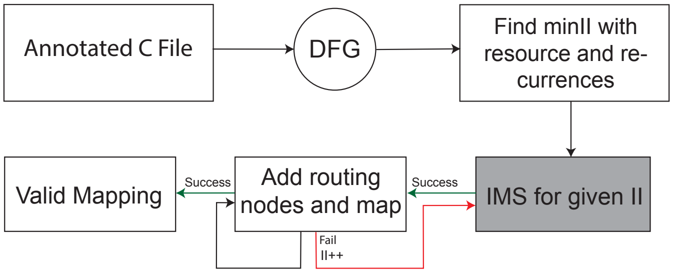


Fig. 4: Overview of scheduling and mapping workflow of previous techniques.

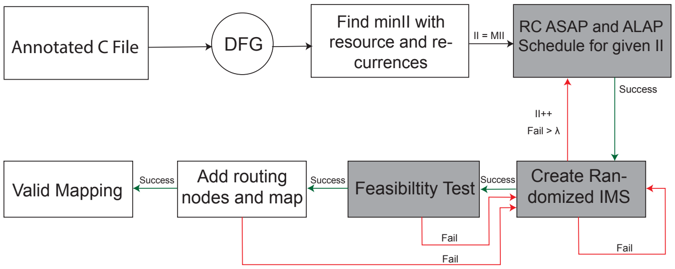


Fig. 5: An overview of CRIMSON workflow, with addition of RC_ASAP and RC_ALAP computation, Randomized Scheduling algorithm, and a Feasibility Test (shaded blocks in the image are proposed by this paper).

With this prescribed schedule, mapping algorithms start to map the nodes, but eventually find that a routing node needs to be added to route operation f and i. Due to the unavailability of PEs in that timeslot a routing node cannot be added, as shown in Fig 3(d). At this juncture, the mapping algorithm increases the II in an effort to find a schedule that is mappable. On increasing the II from 3 to 4, the IMS algorithm is invoked again to get a schedule. Since the priority calculation of IMS is DFG-based, all the nodes get the same priority. Now, IMS algorithm starts to schedule nodes based on the priorities for each node. Nodes a and f are scheduled at modulo time 0 (0%4). Nodes b, c, g, and h are scheduled at modulo time 1 (1%4). Nodes d and i are scheduled at modulo time 2 (2%4) and e is scheduled at modulo time 3 (3%4). The IMS schedule for II=4 is shown in Fig 3(e) column 2. Again, on failure to map, the mapping algorithm increases the II to 5. IMS repeats the process of assigning priorities to the nodes and as seen in II=4, the priorities do not change. Nodes a and f are scheduled at modulo time 0 (0%5). Nodes b, c, g, and h are schedule at modulo time 1 (1%5). Nodes d and i are scheduled at modulo time 2 (2%5) and finally e is scheduled at modulo time 3 (3%5). On comparing the schedules obtained for II=3, II=4, and II=5, it can be seen that only node e has a different schedule time (from II=3 to II=4) and rest of the nodes have the same schedule. Hence, with IMS, it can be seen that an increase in the II does not correspond to a change in modulo schedule time of the nodes.

The algorithm keeps trying to find a valid mapping at higher II even when there is a mapping failure at a given modulo schedule. This process keeps on repeating endlessly. In the workflow of the previous techniques, as shown in Fig 4, after finding the MinII and obtaining an IMS schedule, the mapping of the nodes begin assuming that the schedule is mappable. There are no mechanism to statically and systematically find the feasibility of the obtained schedule, which results in an infinite loop between the scheduling and the mapping stages.

V. CRIMSON: Efficiently accelerate loops by Randomized Iterative Modulo Scheduling and Optimized mapping

A. Overview

To alleviate the challenges posed by IMS and the previous mapping algorithms, CRIMSON randomizes the schedule time of each node of the DFG by choosing a time between RC_ASAP and RC_ALAP. Additionally, CRIMSON proposes a change to the previous mapping algorithm workflow Fig 4 by performing a feasibility test before the actual mapping.

Fig 5 shows the modification to the traditional IMS-based workflow shown in Fig 4. CRIMSON modifies the IMS-based mapping workflow by adding RC_ASAP and RC_ALAP computation steps before finding a random schedule. The "Create Randomized Schedule" block uses Algorithm 1 and Algorithm 2 to find a random modulo schedule time. On a failure to find a schedule, "Create Randomized IMS" block is invoked λ times before increasing the II. When a random modulo schedule is obtained, the feasibility test statically analyzes if the obtained random schedule honors the resource constraints when routing nodes are added. If a schedule is found to be infeasible due to possible resource overuse, a different modulo schedule is tried for the same II. If the random schedule obtained is valid and feasible, then the mapping algorithm is called to add routing nodes and map the scheduled DFG onto the CGRA architecture.

B. Computing Resource-Constrained ASAP and Resource-Constrained ALAP

Algorithm 1 shows the CRIMSON's randomized iterative modulo scheduling algorithm. Lines 1-2 finds the RC_ASAP from the Strongly Connected Components (SCCs)[1] of the DFG. The RC_ASAP is computed in Line 3 of Algorithm 1 as a top-down, depth-first search approach, from the nodes that do not have any incoming edges in the current iteration. After computation of RC_ASAP, RC_ALAP is computed, starting from the nodes that do not have any outgoing edges in the current iteration and in a bottom-up (reverse), depth-first search manner, in Line 4 of Algorithm 1.

C. Randomized Scheduling Algorithm

After computing RC_ASAP and RC_ALAP, Algorithm 1 Line 5 populates the unscheduled array whereas line 6 sets a boolean Scheduled operation to false for all the nodes, which is used in Algorithm 2. For all the unscheduled sorted nodes in the array, a random modulo timeslot is picked by honoring

[1] Getting the list of SCCs ensures that the nodes in recurrence-cycles are scheduled first using $Sort\_SCC()$ function in Line 5.

Algorithm 1: $Rand\_Iterative\_Mod\_Schedule$ (Input DFG D, CGRA CA, Input II)

```
1  D' ← D;
2  SCCs ← $Find\_List\_of\_Sccs(D')$;
3  $Find\_RC\_ASAP(II, Sccs, CA)$;
4  $Find\_RC\_ALAP(II, Sccs, CA)$;
5  unscheduled ← $Sort\_Sccs(Sccs)$;
6  $Set\_Scheduled\_op\_false(unscheduled)$;
7  iter ← 0;
8  while unscheduled_size > 0 & iter < threshold do
9      operation ← unscheduled[0];
10     TimeSlot ←
         $Find\_Random\_ModuloTime(operation, CA)$;
11     if ($schedule(nodes, TimeSlot)$) then
12         scheduled ← nodes;
13     else
14         return failure;
15     unscheduled ←
         $Subtract(unscheduled, scheduled)$;
16     iter++;
17 if (iter == threshold & unscheduled_size > 0 ) then
18     return failure;
19 return success;
```

Algorithm 2: $Find\_Random\_ModuloTime$ (Operation op, CGRA CA)

```
1  $op\_ASAP \leftarrow get\_RC\_ASAP(op)$;
2  $op\_ALAP \leftarrow get\_RC\_ALAP(op)$;
3  $sched\_slot \leftarrow \varnothing$;
4  timeslots ← $get\_all\_timslots(op\_ASAP, op\_ALAP)$;
5  $Randomize(timeslots)$;
6  while ($sched\_slot == \varnothing$ & $timslots\_size > 0$ ) do
7      $currTime$ ← timeslots[0];
8      if ($ResourceConflict(op, currTime, CA)$) then
9          timeslots ← $Subtract(currTime, timeslots)$;
10         continue;
11     else
12         $sched\_time \leftarrow currTime$ ;
13 if ($sched\_slot == \varnothing$) then
14     if (!Scheduled[op] ||
         $op\_ASAP > Prev\_Sched\_Time[op]$ ) then
15         $sched\_slot \leftarrow op\_ASAP$;
16     else
17         $sched\_slot \leftarrow Prev\_Sched\_Time[op] + 1$;
18 return $sched\_slot$;
```

the resource constraints maintained by MRT, in Line 10 of the Algorithm 1.

The $schedule()$ function in Line 11 of the Algorithm 1, schedules the node at chosen random timeslot . This $schedule$ function sets the schedule time of the current operation and consecutively displaces the nodes that have resource conflicts. Previously scheduled nodes having a dependence conflicts with the current operation are also displaced after updating the RC_ASAP and RC_ALAP based on the current schedule operation. The displaced nodes are added to queue of unscheduled nodes. Similar to the $BudgetRatio$ in IMS [22], the $iter$ is a high value. On a failure to find a schedule, either due to unscheduled nodes lines 13-14 or if the $iter$ value is greater than a threshold (lines 17-18), the Algorithm 1 is invoked again. This is repeated λ times before increasing the II, in an attempt to find a valid schedule. This λ value is not reset for a particular II and used to control the failure due to unmappable schedule or a failure in the mapping step.

Algorithm 2 is called by CRIMSON's randomized iterative modulo schedule ($Rand\_Iterative\_Mod\_Schedule$) Algorithm 1 line 10, to find a random timeslot between RC_ASAP and RC_ALAP. The RC_ASAP and RC_ALAP for a given operation is retrieved in lines 1-2 of Algorithm 2. Then, an array of timeslots is constructed using the op_ASAP and op_ALAP, line 4 of Algorithm 2. The array holds all the timeslots from op_ASAP with an increasing value of 1 until op_ALAP. If op_ASAP is equal to op_ALAP then the array size is one with either ASAP or the ALAP time. Each timeslot from the randomized array is checked for the resource constraint using MRT. The first valid timeslot is returned as the modulo schedule time for the operation. Due to the resource conflict if a valid timeslot is not present, there are two things to handle, (a) a timeslot for the operation should be chosen and (b) an already scheduled operation from that timeslot should be displaced. Concern (a) in handled in lines 13-17 of Algorithm 2 where if the nodes has not been scheduled previously, op_ASAP is chosen as the schedule, else the previous schedule time of the operation is found and the modulo schedule time is computed using line 17. Concern (b) is addressed in the $schedule()$ function in Algorithm 1 line 11, explained earlier. The methods addressing these concerns are similar to IMS implementation.

D. Novel Feasibility Test

Given a valid schedule, it may not be possible to map it because of two main reasons: i) limited connectivity among the PE nodes, and ii) the need to map the extra routing nodes that will be created as a result of scheduling. In a valid schedule dependent operations may be scheduled in non-contiguous timeslots. When this is the case, the operands need to be routed from the PE on which the source operand is mapped, to the PE on which the destination operation is mapped. The operands can be routed using a string of consecutive CGRA interconnections and PEs. These PEs are referred to as routing PEs, and the operation that is mapped on these PEs (just forward the operand from input to output) is called a routing operation. Because of the addition of these routing nodes, the generated schedule may not be mappable. Previous techniques assume that the schedule is mappable and spend a lot of time searching for a mapping when none is available. In order to avoid wasting time in exploring unmappable schedules, CRIMSON adds a conservative feasibility test to prune schedules that can be proven to be unmappable.

The feasibility test examines the random schedule produced, and for each routing resource that will be added in the future, it estimates the resource usage, considering path-sharing [18]. The feasibility test checks if the total number of unique nodes including the routing nodes per timeslot is less than or equal to the number of PEs in that timeslot. $schedule\_nodes_i + routing\_nodes_i <= PEs_i$, where i is the modulo timeslot. This feasibility check is performed for all the II timeslots. The mapping algorithm is invoked only for schedules that are feasible, unlike the previous approaches such as RAMP [20], where the mapping algorithm is invoked even for infeasible schedules. Since the time complexity of such mapping algorithms is high (time complexity of RAMP is $\mathcal{O}(N^8)$, where $N = n * m$, and 'n' is the total nodes in the loop DFG, and 'm' is the size of the CGRA), invoking them for infeasible schedules is counter productive. The feasibility test reduces the overhead incurred by the mapping algorithm by pruning the infeasible schedules.

E. Determining the λ value

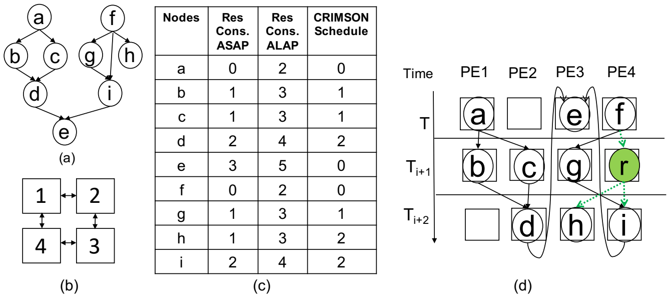
With every failure in the feasibility test a new schedule is obtained for a given II. The number of times a schedule is obtained for a given II is controlled by the λ value. The scheduling space that can be explored for a given II is calculated by the product of the total nodes in the DFG, the size of the CGRA, and the II, given in Equation 1. A brute force exploration of the schedule space is time consuming. Lower λ values may increase the II prematurely, by superficial exploration of schedule space, whereas higher λ values increase the compilation time, due to elaborate exploration of the schedule space. Due to the randomness in the scheduling algorithm, a feasible schedule may be obtained faster by chance even for a higher λ value. The λ value is computed using,

$$\lambda = exploration\_factor \times n \times m \times II \tag{1}$$

where, 'n' is the total number of nodes in the loop DFG, 'm' is the size of the CGRA and, *exploration_factor* is the percentage of the schedule space that is to be explored. The *exploration_factor* is a user defined parameter. II is also one of the parameters that determines the λ value in Equation 1, which means that a new λ is computed for each II. When the II is increased, the scheduling space is also increased therefore the scope of exploration gets broadened. A detailed discussion on the effects of exploration_factor on the scheduling time and II is given in Section VI-E.

F. Running Example

Fig 6 shows the working of CRIMSON's randomized iterative modulo schedule algorithm for the DFG and CGRA architecture shown in Fig 6(a)-(b)[2]. Instead of assigning a priority based on height like IMS, each node in DFG is assigned two times namely, Resource Constrained As Soon As Possible (RC_ASAP) and Resource Constrained As Late As Possible (RC_ALAP), which constitutes a good lower and upper bound for scheduling [16]. Similar to IMS, CRIMSON maintains an MRT to check for resource overuse during RC_ASAP and RC_ALAP assignment. The RC_ASAP is calculated from the nodes that does not have any incoming edges in the current iteration. These nodes are allotted RC_ASAP time as 0, which means, that the earliest start time of these nodes is at time 0. Based on the outgoing nodes from these start nodes and the delay of each operation, the RC_ASAP of consecutive nodes are computed in a depth-first manner (similar to IMS priority calculation). For the DFG in analysis, nodes a and f are assigned the RC_ASAP time as 0. Nodes b, c, g, and h are assigned RC_ASAP time as 1. Nodes d and i are assigned RC_ASAP time 2 and node e is assigned RC_ASAP time 3. The RC_ASAP times of each node is shown in Fig6(c) column 2. Next, starting from the last nodes of the DFG, i.e., nodes without any outgoing nodes in the current iteration, the nodes are assigned RC_ALAP in a reverse depth-first search manner, using $RC\_ALAP = RC\_ASAP + II - 1$. This ensures that $RC\_ALAP >= RC\_ASAP$. For the given DFG, e is assigned RC_ALAP time 5, node h is assigned 3. Nodes d and i are assigned RC_ALAP time 4. Nodes b, c and g are assigned RC_ALAP time 3. Finally a and f are assigned RC_ALAP time 2. The RC_ALAP times of each node is shown in Fig6(c) column 3.



Nodes	Res Cons. ASAP	Res Cons. ALAP	CRIMSON Schedule
a	0	2	0
b	1	3	1
c	1	3	1
d	2	4	2
e	3	5	0
f	0	2	0
g	1	3	1
h	1	3	2
i	2	4	2

Fig. 6: (a) The DFG of the motivation example. (b) A 2x2 CGRA architecture. (c) For each node of the DFG, resource constrained ASAP (column 2) and resource constrained ALAP (column 3) is first calculated. Then a random schedule time between RC_ASAP and RC_ALAP is chosen for each node. A valid randomized modulo schedule is shown in column 4. (d) With CRIMSON schedule a valid mapping is achieved by the mapping algorithm at II=3.

After computing the RC_ASAP and RC_ALAP, CRIMSON chooses a random time between RC_ASAP and RC_ALAP, to schedule the nodes. Like IMS, CRIMSON maintains a Modulo Resource Table (MRT) to check for resource overuse in each II modulo timeslot. After checking for resource constraints the modulo schedule time is chosen for each node. This randomization of modulo schedule time creates flexibility of movement for the nodes, which explores different modulo schedule spaces, thereby increasing the chances of finding a valid mapping by the mapping algorithm. A randomized modulo schedule for the example DFG is shown in Fig6(c) column 4, and a valid mapping for the scheduled nodes is shown in Fig 6(d) at II=3. The loop that was previously unmappable due to the restrictive scheduling of IMS Fig 3, is now mappable at II=3 due to randomization in assigning

[2] The DFG and the architecture is the same as the motivation example Fig.3(a)-(b)

Suites	Loops	#nodes	#mem. nodes	#edges
MiBench	bitcount	22	4	28
	susan	31	8	35
	sha	31	10	39
	jpeg1	43	10	48
	jpeg2	28	6	33
Rodinia	kmeans1	15	6	17
	kmeans2	16	6	17
	kmeans3	17	4	20
	kmeans4	16	4	19
	kmeans5	12	2	13
	lud1	21	4	24
	lud2	20	4	24
	b+tree	13	2	13
	streamcluster	16	4	19
	nw	20	6	21
	BFS	28	10	32
	hotspot3D	76	20	96
	backprop	39	16	44
Parboil	spmv	25	8	27
	histo	18	4	20
	sad1	25	4	30
	sad2	19	4	20
	sad3	12	4	12
	stencil	69	16	94

TABLE II: Benchmark characteristics.

modulo schedule time.

If we take a closer look at the RC_ASAP and RC_ALAP times shown in Fig6(c) column 2 and 3, we can observe that there is a chance that the RC_ASAP may be the modulo schedule chosen for all the nodes, since assigning a modulo schedule time for the nodes from RC_ASAP and RC_ALAP is randomized. As seen in Fig 3(d)&(e), this schedule is not mappable. Unless there is a change to the workflow, there is a chance that finding a schedule that is unmappable and increasing the II to get a schedule process is repeated. To take care of this issue, CRIMSON proposes changes to the previous IMS-based workflow by statistically computing the feasibility of the scheduled nodes, prior to the mapping of the nodes. This makes sure that if a schedule is not mappable, a different random schedule is tried again for the same II. The number of times the mapping is tried for a given II is controlled by a threshold factor λ. With induced randomization in mapping and changes to the workflow, CRIMSON is able to achieve mapping of the application loops that were previously unmappable by IMS-based mapping techniques.

VI. Experimental Results

A. Setup

Benchmarks: We profiled top three of the widely used benchmark suites namely, MiBench [31], Rodinia [32], and Parboil [33]. The top performance-critical, non-vectorizable loops[3] were chosen for the experiments. Loops that could not be compiled or the loops that were memory bound were not considered. Experiments were designed to consider only innermost loops so that a direct comparison with IMS can be made. These benchmarks depict a wide variety of applications from security, telecomm etc. to parallel, high-perfomance computing (HPC) loops like spmv (sparse matrix-vector product). These loops on average across all the benchmark loops, corresponds to $\geq$50% of the total application execution time.

Compilation: For selecting the loops from the application and converting the loops to the corresponding DFG, we used CCF [34] - CGRA Compilation Framework (LLVM 4.0 [35] based). On top of the existing framework, to effectively compile the loops with control-dependencies (If-Then-Else structures), we implemented partial predication [36] as an LLVM pass, to convert the control-dependencies into data dependencies. Partial Predication [36] can efficiently handle loops with nested if-else structures. The loop characteristics are shown in Table II including the number of nodes in the DFG (only computing nodes are included and constants that can be passed in the immediate field of the ISA are excluded) and number of memory (load/store) nodes. CCF framework [34] produces DFG of the loop with separate address generation and actual load/store functionality. Furthermore, during the addition of routing resources after scheduling, we have implemented path-sharing technique proposed in GraphMinor [18]. Path-sharing can reduce the redundant routing nodes added. We implemented CRIMSON as a pass in the CCF framework including the λ value computation and the feasibility test. We also implemented the IMS-based state-of-the-art RAMP [20] and GraphMinor [18] as a pass in CCF. As observed in Table I, RAMP has demonstrated equal or better results when compared to GraphMinor. Hence, we compare CRIMSON against RAMP. We compiled the applications of the benchmark suites using optimization level 3 to avoid including loops that can be vectorized by compiler optimizations. We considered 2D torus mesh CGRA of sizes 4×4, 5×5, 6×6, 7×7, and 8×8.

B. CRIMSON is able to schedule and map loops that could not be mapped using RAMP

From Table III, we can infer that for loops, $jpeg1$, $jpeg2$, $hotspot3D$, $backprop$, and $stencil$, IMS-based state-of-the-art heuristic RAMP, was not able to find a valid mapping for a 4×4 CGRA (denoted by "**X**" in Table III). From the motivating example Fig 3, IMS produces almost the same modulo schedule time for most of the nodes for any increase in II. CRIMSON, on the other hand, facilitates the exploration of different modulo scheduling times for nodes of the DFG, resulting in a valid mapping. It is observed that even at a lower CGRA size 4×4, CRIMSON was able to map these particular loops. From Table III, when running on RAMP, loops that were not mappable on a 4×4 CGRA, were eventually mapped when allocated enough resources. For example, $stencil$ which was unmappable by RAMP on a 4×4 CGRA was mapped on a 5×5 CGRA due to allocation of additional resources. Therefore it can be said that the motivating example can also be mapped when allocated enough resources. From the motivating example, if Fig 3(b) CGRA architecture was a 3×3 CGRA, then the IMS-based mapping algorithm would have used the extra resources provided to route the operation r. But this conclusion was not applicable to all the loops, meaning, loops such as $hotspot3D$ and $jpeg2$ were unable

[3] Maximum up to 5 loops per benchmark, with each contributing $>$7% of the execution time of the application when executed with standard inputs that are shipped with the benchmark suites.

Suites	Loops	4x4			5x5			6x6			7x7			8x8		
		MII	RAMP	CRIM.	MII	RAMP	CRIM.	MII	RAMP	CRIM.	MII	RAMP	CRIM.	MII	RAMP	CRIM.
MiBench	bitcount	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	susan	2	3	4	2	2	2	2	2	2	2	2	2	2	2	2
	sha	3	3	4	2	**X**	3	2	3	2	2	2	3	2	2	4
	jpeg1	3	**X**	6	2	**X**	4	2	2	2	2	2	2	2	2	2
	jpeg2	2	**X**	5	2	**X**	3	2	**X**	2	2	2	2	2	2	2
Rodinia	kmeans1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	kmeans2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	kmeans3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	kmeans4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	kmeans5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	lud1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	lud2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	b+tree	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	streamcluster	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	nw	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2
	BFS	2	2	3	2	3	3	2	2	3	2	2	2	2	2	3
	hotspot3D	5	**X**	10	4	**X**	7	4	**X**	7	3	**X**	6	3	**X**	4
	backprop	5	**X**	7	4	4	4	3	3	3	3	3	3	3	3	4
Parboil	spmv	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2
	histo	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	sad1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	sad2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	sad3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	stencil	4	**X**	6	3	4	5	3	3	3	3	3	4	2	2	2

TABLE III: A comprehensive table showing the MII and II achieved by RAMP, an evaluated IMS-based heuristic, and CRIMSON (CRIM.) for 24 benchmark application loops from three major benchmark suites at 0.005 $exploration\_factor$. The "**X**" in the table denotes that there was no mapping obtained from RAMP for an increasing II up to 50. MII in the table denotes the minimum II, which is the maximum of either ResMII or RecMII.

to find a valid mapping even when additional resources were allocated. RAMP was not able to achieve a mapping even at 8×8 CGRA for $hotspot3D$ whereas RAMP was not able to achieve a mapping till 6×6 for $jpeg2$. While RAMP is able to map most of the loops at a higher CGRA size, CRIMSON with effective randomized modulo scheduling was able to map all the loops at size 4×4. Additionally, for $sad1$ and $sad3$ loops, for which GraphMinor was not able to find a mapping, CRIMSON was able to achieve a mapping at 4×4 CGRA size.

C. CRIMSON has nearly identical II for loops that could be mapped using RAMP

From Table III we can observe that for loops mapped using RAMP, the II obtained from CRIMSON was comparable to RAMP across five different CGRA sizes ranging from 4×4 to 8×8. We can see an occasional spike in the II in CRIMSON for $susan$ at 4×4 and $stencil$ on 5×5, which is due to premature II increase by CRIMSON based on the λ value. To emphasize, λ is the maximum number of randomized schedules that are explored at the same II. A new schedule may be requested (i) on a failure to find a randomized schedule, (ii) on a failure of the feasibility test or, (iii) a failure to map. The λ value is not reset for a given II. After exhausting the λ limit, the II is increased and a new RC_ASAP and RC_ALAP is computed along with a new λ value. The λ value is computed by Equation 1 for each II. The λ value is determined by the user defined *exploration_factor*, which is the percentage of schedule space to that should be explored. If the *exploration_factor* is set too low, less modulo schedules are explored per II, thereby making it difficult to obtain a valid mapping and increasing the II prematurely. If the *exploration_factor* is set too high the time to obtained a valid schedule/mapping increases, which negatively affects the compilation time of CRIMSON. Table III comprehensively conveys that CRIMSON has a nearly identical performance compared to RAMP for all the loops across different CGRA architectures that RAMP was able to map and CRIMSON is better than RAMP by mapping the five loops that were not mappable by RAMP and seven loops that were not mappable by GraphMinor on a 4×4 CGRA. The II obtained from CRIMSON is not always equal to or better than state-of-the-art RAMP and is dependent on the λ value.

D. Scheduling time comparison between CRIMSON and IMS.

The scheduling time for IMS [22] and CRIMSON are shown in Fig. 7, which is reported based on the execution of both the algorithms on Intel-i7 running at 2.8GHz with 16GB memory. As shown in Fig 7, the x-axis is the scheduling time i.e, time to obtain a valid schedule that is mappable, in μs (microseconds) and the y-axis corresponds to the benchmark loops. The 19 benchmarks shown in Fig 7 are those in which a mappable schedule was obtained by IMS. From Fig 7, we can see that the scheduling time of CRIMSON is slightly higher than that of IMS. This is due to the additional computation of RC_ASAP and RC_ALAP, and the feasibility test (Fig 5). For the loops shown, the *exploration_factor* was kept at 0.005.

E. Trade-off analysis between scheduling time and II at different λ values.

From Equation 1, we can see that the λ value depends on the *exploration_factor*. This factor is defined as the percentage of modulo schedule space to be explored when there is an infeasible schedule or a mapping failure. The exploration_factor was

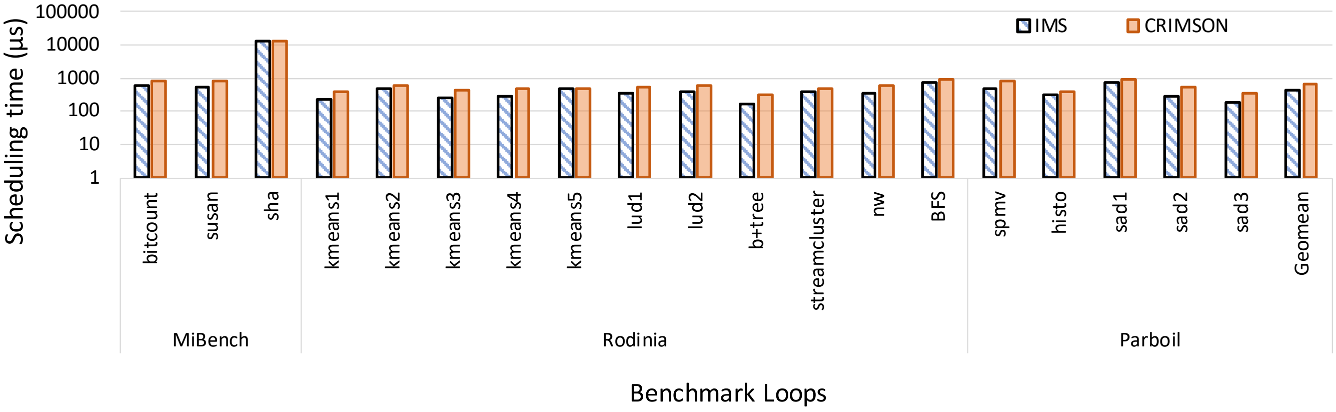


Fig. 7: Scheduling time comparison of CRIMSON and IMS.

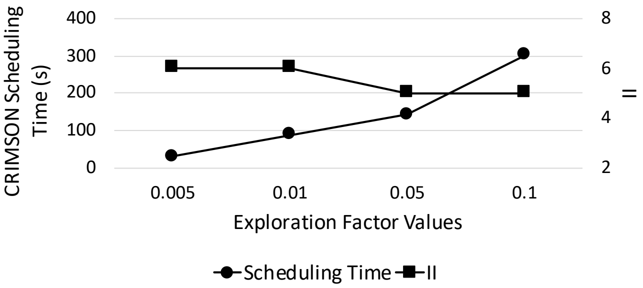


Fig. 8: Scheduling time vs. II trade-off trend for stencil.

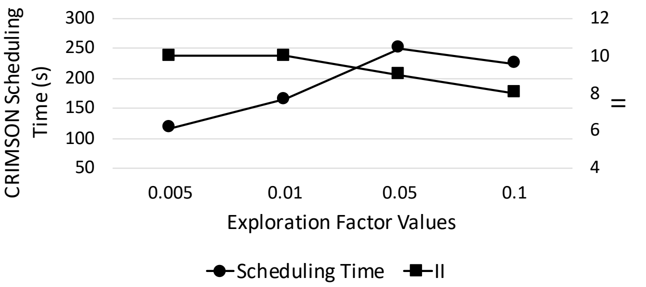


Fig. 9: Scheduling time vs. II trade-off trend for hotspot3D.

changed from 0.5% (0.005) to 10% (0.1) and the corresponding scheduling time and II were recorded. The scheduling time numbers are recorded from executing CRIMSON on Intel-i7 running at 2.8GHz and 16GB memory and the compilation was performed for a 4×4 CGRA. A 4×4 CGRA was chosen because the II obtained by CRIMSON was much greater than the MII and the effect of λ can be shown clearly. In Fig 8 and Fig 9, the left y-axis (primary axis) denotes the CRIMSON scheduling time, in seconds, and the right y-axis (secondary axis) denotes the II obtained. The x-axis denotes the different *exploration_factors*. From Equation 1 it is to be noted that as the exploration_factor increases, the λ value increases. From Fig 8 and Fig 9, it is evident that as exploration_factor increases the CRIMSON scheduling time increases, due to elaborate exploration of the schedule space at a given II. For lower value of the exploration_factor, superficial exploration of modulo schedule space prematurely increases the II but at lower scheduling time. We can also note from Fig 9 at 0.1 that the above statement is not always true. At 0.1 the II decreases with the decrease in the scheduling time because a feasible and a mappable schedule was obtained earlier in the modulo schedule space exploration due to the innate randomness of the CRIMSON scheduling algorithm.

VII. Conclusion

This paper presented some of the major challenges encountered in the state-of-the-art mapping techniques with respect to scheduling and mapping of compute-intensive loops onto the CGRA. The previous mapping techniques use IMS scheduling that rarely showed a change in the modulo schedules for increased II, which obstructed the mapping algorithm to map the application loops onto the CGRA architecture. Additionally, previous mapping techniques assumed that the obtained IMS schedule is mappable and started to map the scheduled nodes. On a failure to map, due to the limited connectivity of the PEs or addition of routing nodes, the mapping algorithms increase the II and call IMS again to get a schedule that almost never changes. To mitigate these challenges, this paper introduced CRIMSON, that comprehensively modeled RC-ASAP and RC-ALAP, picking a random modulo schedule time between these upper and lower boundaries. CRIMSON generated different schedules, thereby exploring different schedule spaces, on each invocation for a given or increased II. CRIMSON also introduced a novel feasibility test that pruned schedules that are unmappable. On evaluating the top 24 performance-critical loops from MiBench, Rodinia and Parboil, CRIMSON was able to map 5 application loops that were unmappable by RAMP and 7 application loops that were unmappable by GraphMinor. The II achieved by CRIMSON was comparable to the II achieved by RAMP for the application loops that were mappable by RAMP.

Acknowledgment

This work was partially supported by funding by National Science Foundation grants CNS 1525855, and CCF 1723476 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

References

[1] Lirong Zheng, Hui Zhang, Weili Han, Xiaolin Zhou, Jing He, Zhi Zhang, Yun Gu, Junyu Wang, et al. Technologies, applications, and governance in the internet of things. *Internet of things-Global technological and societal trends. From smart environments and spaces to green ICT*, 2011.

[2] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[3] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 66–81. Springer, 2008.

[4] Chris Nicol. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. *Wave Computing White Paper*, 2017.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[7] Xitian Fan, Huimin Li, Wei Cao, and Lingli Wang. DT-CGRA: Dual-track Coarse-Grained Reconfigurable Architecture for Stream Applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[8] Devereaux Conrad Chen. *Programmable arithmetic devices for high speed digital signal processing*. PhD thesis, Citeseer, 1992.

[9] Reiner W Hartenstein and Rainer Kress. A Datapath Synthesis System for the Reconfigurable Datapath Architecture. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*, pages 479–484. IEEE, 1995.

[10] Ethan Mirsky, Andre DeHon, et al. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.

[11] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE transactions on computers*, 49(5):465–481, 2000.

[12] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems*, 82(2):389–397, 1999.

[13] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using epimorphism to map applications on CGRAs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1284–1291. ACM, 2012.

[14] B Mei, M Berekovic, and JY Mignolet. ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine-and coarse-grain reconfigurable computing*, pages 255–297. Springer, 2007.

[15] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146. ACM, 2006.

[16] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[17] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *ACM Sigplan Notices*, volume 44, pages 21–30. ACM, 2009.

[18] Liang Chen and Tulika Mitra. Graph minor Approach for Application Mapping on CGRAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21, 2014.

[19] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. REGIMap: Register-aware Application Mapping on Coarse-Grained reconfigurable architectures (CGRAs). In *Proceedings of the 50th Annual Design Automation Conference*, page 18. ACM, 2013.

[20] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: Resource-Aware Mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[21] Shouyi Yin, Xianqing Yao, Dajiang Liu, Leibo Liu, and Shaojun Wei. Memory-aware Loop Mapping on Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1895–1908, 2016.

[22] B Ramakrishna Rau. Iterative Modulo Scheduling. *International Journal of Parallel Programming*, 24(1), 1996.

[23] Benmao Cheng, Hong Wang, Shiyuan Yang, Daoheng Niu, and Yang Jin. A Novel Testability-oriented Data Path Scheduling Scheme in High-Level Synthesis. *Tsinghua Science & Technology*, 12:134–138, 2007.

[24] Li Zhou, Dongpei Liu, Min Tang, and Hengzhu Liu. Mapping Loops onto Coarse-Grained Reconfigurable Array Using Genetic Algorithm. In *Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013*, pages 801–808. Springer, 2013.

[25] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for CGRAs. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8. IEEE, 2013.

[26] Ganghee Lee, Kiyoung Choi, and Nikil D Dutt. Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(5):637–650, 2011.

[27] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[28] Giovanni Ansaloni, Kazuyuki Tanimura, Laura Pozzi, and Nikil Dutt. Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(12):1803–1816, 2012.

[29] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–26, 2019.

[30] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[31] Matthew Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[32] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[33] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[34] Shail Dave and Aviral Shrivastava. CCF: A CGRA Compilation Framework. 2018.

[35] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[36] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(2):8, 2013.

Mahesh Balasubramanian is a Ph.D. student at Arizona State University. He received his Masters in Electrical Engineering from the University of Texas at San Antonio and bachelors from Anna University, Chennai, India. His research interests lie in area of parallel computing and co-processor accelerators like CGRAs particularly for general purpose and HPC applications. He is a recipient of CIDSE Doctoral Fellowship, Engineering Grad Fellowship, and Ferdinand A. Stanchi Fellowship at Arizona State University.

Aviral Shrivastava is a Professor in the School of Computing Informatics and Decision Systems Engineering at the Arizona State University, where he has established and heads the "Make Programming Simple (MPS)" lab. He received his Ph.D. and Masters in Information and Computer Science from the University of California, Irvine, and bachelors in Computer Science and Engineering from Indian Institute of Technology, Delhi. He is a 2011 NSF CAREER Award Recipient, and recipient of 2012 Outstanding Junior Researcher in CSE at ASU. His works have received several best paper nominations, including at DAC 2017, and a best student paper award at VLSI 2016. His students have received outstanding Ph.D. student award in CSE @ ASU in 2017, and outstanding MS student award in CSE at ASU in 2012 and 2010. Prof. Shrivastava's research lies in the broad area of "Software for Embedded and Cyber-Physical Systems." More specifically, Prof. Shrivastava is interested in topics around i) Compilers and microarchitectures for heterogeneous and many-core computing, ii) protecting computation from soft errors, and iii) Precise timing for Cyber-Physical Systems. His research is funded by NSF, DOE, NIST, and several industries including Microsoft, Raytheon Missile Systems, Intel, Nvidia, etc. He serves on the organizing and program committees of several premier embedded system conferences, including DAC, ICCAD, ISLPED, ESWEEK, and LCTES. He is currently serving as deputy editor in chief of IEEE Embedded Systems Letters, associate editor for ACM Transactions Embedded Computing Systems (ACM TECS), IEEE Transactions on Computer-Aided Design (IEEE TCAD), and Springer International Journal on Parallel Processing (Springer IJPP), and Springer Design Automation for Embedded Systems (Springer DAEM). He has served as the program chair of CODES+ISSS 2017 and 2018, and LCTES 2019, and is currently the virtual conference chair of ESWEEK 2020, and track chair for RTSS 2020