PathSeeker: A Fast Mapping Algorithm for CGRAs

Mahesh Balasubramanian and Aviral Shrivastava
Make Programming Simple Lab, Arizona State University, Tempe, AZ
Email: {mbalasubramanian, aviral.shrivastava}@asu.edu

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) have gained traction over the years as a low-power accelerator due to the efficient mapping of the compute-intensive loops onto the 2-D array by the CGRA compiler. When encountering a mapping failure for a given node, existing mapping techniques either exit and retry the mapping anew, or perform backtracking, i.e., recursively remove the previously mapped node to find a valid mapping. Abandoning mapping and starting afresh can deteriorate the quality of mapping and the compilation time. Even backtracking may not be the best choice since the previous node may not be the incorrectly placed node. To tackle this issue, we propose PathSeeker – a mapping approach that analyzes mapping failures and performs local adjustments to the schedule to obtain a mapping. Experimental results on 35 top performance-critical loops from MiBench, Rodinia, and Parboil benchmark suites demonstrate that PathSeeker can map all of them with better mapping quality and dramatically less compilation time than the previous state-of-the-art approaches – GraphMinor and RAMP, which were unable to map 20 and 5 loops, respectively. Over these benchmarks, PathSeeker achieves 28% better performance at 550x compilation speedup over GraphMinor and 3% better performance at 10x compilation speedup over RAMP on a 4×4 CGRA.**

***Index Terms*—Compilers, Reconfigurable Architectures, CGRA, Mapping.**

I. Introduction

The advancement of Internet and data collecting devices have increased the demand for high-performance, low-power computing alternatives. All mobile devices collect, process, and communicate data. Analyzing the collected data to extract meaningful information is compute-intensive [1] and often limited by the thermal, power and resource constraints [2]. Coarse-Grained Reconfigurable Arrays (CGRAs) are promising accelerators that provide high efficiency at low power [3], [4]. A CGRA consists of a simple 2-D grid of Processing Elements or PEs. Each PE contains Functional Units (FUs) that can receive instructions from the instruction memory, compute arithmetic operations with the data received from the data memory or the neighboring PEs. Each PE consists of MUXes to select the inputs from its neighbors and a register file to store intermittent data. Prime examples of CGRAs are accelerators like Eyeriss [5], DianNo [6], Marvel [7], that have been used for power-efficient acceleration of machine learning models like Convolution Neural Networks (CNNs), Deep Neural Networks (DNNs) etc.

In order to achieve the high performance and highly power-efficient operation of CGRAs good compilers are needed, which will be able to obtain a good quality mapping of performance-critical loops from applications. CGRA compilers can be classified into two categories: (1) Parallel-loop compilers, (2) Modulo Scheduling-based compilers. The parallel-loop compilers like the ones for [5], [7] employ various compiler optimizations to exploit the inherent spatial and temporal parallelization strategies to map parallel loops of an application onto the PEs of the accelerator [7]. However, not all the compute-intensive loops of an application may be parallel, and those can be accelerated through modulo scheduling-based compilers. Modulo-scheduling based compilers accelerate the data flow graph of the loop body through the pipelining present in the CGRAs using software pipelining [4], [8]–[14]. This paper focuses on the modulo scheduling-based compiler techniques that can support a wide variety of application loops.

One of the biggest limitations of the existing modulo scheduling-based state-of-the-art CGRA mapping techniques is that, when trying to map loops onto the CGRA if a mapping attempt fails, these techniques either discard the current mapping and restart anew or backtrack to the previously mapped node. Techniques that restart do not learn anything from the failure, and just blindly explore the mapping space. Even the backtracking based approaches may not be effective, as they recursively unmap the last mapped node, while the last node may not be the one that is making the mapping infeasible. As a result, existing modulo scheduling-based state-of-the-art CGRA mapping techniques are unable to map some performance-critical loops even after 27 hours! This not only exacerbates the compilation time, but given reasonable limits on compilation time, it also negatively impacts the quality of the mapping achieved by these techniques.

To address these concerns, in this paper, we present a novel mapping algorithm - PathSeeker. First, instead of backtracking or restarting the mapping like the previous mapping methods, PathSeeker analyzes the predecessor and successor nodes to find the reason behind the failed mapping. Second, PathSeeker explores local transformations for the predecessor and successor of the failed node to achieve a valid mapping. Finally, when local transformations do not yield a valid mapping, different PE positions of the other nodes in the time-slot of the failed node, the predecessor, and successor are iteratively explored, to find a valid mapping. We compare the mapping quality generated by PathSeeker to that of GraphMinor [11] and RAMP [12], which are state-of-the-art mapping algorithms in backtracking and restart, respectively. Experimental results on 35 application loops from the top three benchmark suites, MiBench [15], Rodinia [16], and Parboil [17] show that (i) PathSeeker can map all the 35 application loops on 4×4 CGRA, whereas GraphMinor and RAMP were not able to map 20 and 5 loops, respectively, (ii) PathSeeker achieves a better

quality of mapping at lower compilation time with 550x and 10x compilation time speedup over GraphMinor and RAMP respectively, (iii) PathSeeker scales well across different sizes of CGRA.

II. Related Works

In the context of response to a mapping failure, the existing modulo scheduling-based CGRA compiler techniques can be classified into two categories, i) restart and ii) backtrack. Genetic algorithms, simulated annealing [18], [19], minimum common subgraph (MCS) [9] or maximal clique [10], [12] based techniques can be classified as restart. Minimum common subgraph and maximal clique techniques discard the mapping on failure and search for another mapping. Simulated Annealing techniques try random time-slot and PE placements for the failed nodes, generally, having higher compilation times [20].

GraphMinor [11], RAMS [21], and BMS [20] perform backtracking on a mapping failure. RAMS and BMS form clusters from the DFG and map the clusters one-by-one. When the algorithm is unable to find a mapping for a node, the current cluster mapping is discarded, and the algorithm backtracks to the cluster that was mapped prior. However, GraphMinor maps the DFG by prioritizing nodes based on the critical path, one node at a time, and uses an exhaustive search technique. On a failure, GraphMinor backtracks to the previously mapped node in the mapped order. Essentially, GraphMinor $un-maps$ the last mapped node and tries again by mapping that node to a different place (PE). If that does not work, it continues to un-map the nodes in the reverse order in which they were mapped and keeps trying. However, the last node mapped may not be the problematic node. Even if that last node were re-mapped, it might not enable a valid mapping. HyCube [4] proposes a mapping technique for a highly connected CGRA that uses a multi-hop multi-cast path system to communicate data in a single cycle. HyCube's single-cycle communication may provide better II[1], but at the cost of scalability. Since the interconnect crossbar selection is a part of HyCube's instruction set, which negatively impacts the power and performance as the CGRA size increases.

III. Motivating Example

In GraphMinor [11], the order in which the nodes are mapped plays an important role in determining the compilation time. GraphMinor sorts the nodes of the DFG in the order of the critical paths and cycles [11]. Fig. 1(c) shows a mapping failure of node 2 due to unavailability of connected PEs (connected resource for PE1 and PE3 is PE2 that is occupied by *r*). GraphMinor backtracks to previously mapped node 6, which does not affect the mapping of 2. It checks all the different mapping for 6 and on a failure to find a valid mapping for 2, GraphMinor backtracks to nodes 0 and 5. Node 0 is the predecessor of node 2 but the problem does not lie there. GraphMinor does not find a valid mapping for 2, so it backtracks again to node 1, and node 3, for which it cannot

[1]In modulo scheduling, the interval in which successive iterations can begin execution is called the *Initiation Interval* (II) [22], which is the performance metric.

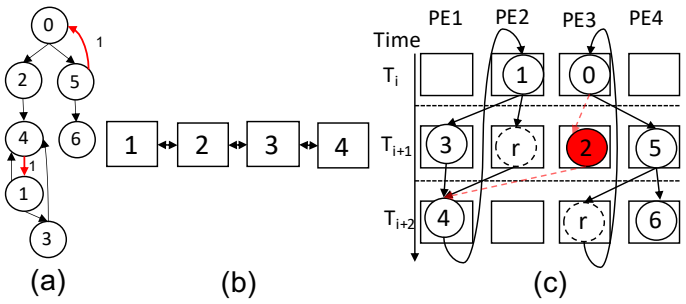

Fig. 1. a) DFG of an application loop. (b) A 1×4 CGRA target architecture. (c) Failure to map node 2 by GraphMinor

find a valid mapping for node 2. After backtracking through all the mapped nodes, GraphMinor reaches 4 wherein the actual problem lies. Now remapping node 4 will yield in a mapping. GraphMinor fails to identify that the problem was one step away – remapping node 4 would have fixed the mapping in the first attempt. Due to the exhaustive nature of the GraphMinor algorithm, the compilation time increases exponentially with an increase in DFG size (compute intensive section with more nodes and recurrences) and CGRA size.

The aim of RAMP [12] is to find the optimal routing between two operations that are scheduled more than one cycle apart by the scheduler. The core algorithm for finding PE for a node in the DFG is the maximal clique algorithm[2]. RAMP takes three steps to resolve all the routing issues and find a maximal clique. On every attempt, RAMP tries to find a maximal clique, and an increase in DFG size due to the addition of routing resources also increases the mapping search time. Another drawback of the RAMP is that it does not isolate the failed mapping. The maximal clique algorithm is time-consuming, $\mathcal{O}(N^8)$ [12], [23], where N is a product of nodes in DFG and CGRA size. Restarting the algorithm on every failure will possibly lead to a longer compilation time.

As illustrated, the existing mapping approaches do not solve the mapping failure by targeting the failed node and hence suffers from poor performance or increased compilation time to converge at a valid mapping. Given the NP-completeness nature of the mapping, we will never know if these techniques will produce a mapping within a finite time. The objective of PathSeeker is to achieve a good quality mapping within a limited amount of time for all the application loops considered across various sizes of CGRA.

IV. PathSeeker

A driver function[3] calls the $PathSeeker$ mapping routine to map the scheduled operations. The PathSeeker algorithm is shown in Algorithm 1. Lines 1-3 initialize an empty queue and pushes the node v into the queue. For a given node v chosen

[2]We do not illustrate the RAMP working due to the complex nature of the maximal clique algorithm and the optimal routing algorithm. We point the readers to [23] for the maximal clique algorithm and [12] for the optimal routing algorithm.

[3]For brevity, the driver algorithm is not shown, but is very similar to the driver function $ScheduleAndMap$ of GraphMinor [11] using Modulo Resource Routing Graph (MRRG).

Algorithm 1: $PathSeeker$ (List $AList$, Node v)

```
1  Initialize empty queue;
2  visited[v] = true;
3  queue.push(v);
4  while (queue ≠ empty) do
5      v = queue.front();
6      queue.pop(v);
7      if is_already_mapped(v) then
8          continue;
9      P ← Get_Mapped_Pred(v);
10     S ← Get_Mapped_Succ(v);
11     Γ ← Get_Connected_PEs(v, P, S);
12     if Γ.size() = 1 then
13         PE ← Γ(0);
14     else if Γ.size() > 1 then
15         PE ← Γ(Rand(Γ.size()));
16     else
17         if Localized_Search(v, P, S) ≠ true then
18             if Recovery_Level_One(v, P, S) ≠ true
                then
19                 if
                    Recovery_Level_Two(v, P, S) ≠ true
                    then
20                     return failure;
21     SetMappablePositions(v, Γ) ;
22     SetCurrentPosition(v, PE) ;
23     for i in AList[v] do
24         if visited[i] ≠ 0 then
25             visited[i] = true;
26             queue.push(i);
27 return success
```

Algorithm 2: $Localized\_Search$(Node v, Predecessor P, Successor S)

```
1  timeslot ← Get_Modulo_Schedule_Time(v)
2  mapped_pred_succ ← P.size() + S.Size()
3  succ_map_set ← ∅
4  pred_map_set ← ∅
5  v_pe ← Get_Free_PEs(timeslot)
6  for i in v_pe do
7      for j in S do
8          s_pe ← GetMappablePositions(j)
9          for jj in s_pe do
10             for k in P do
11                 p_pe ← GetMappablePositions(k)
12                 for kk in p_pe do
13                     if connectedPEs(i, kk, jj) then
14                         Γ.insert(i)
15                         succ_map_set.insert(k)
16                         pred_map_set.insert(j)
17                         store_connected_pes(i, kk, jj)
18 if Γ.size() = 0 then
19     return false
20 Update Γ and PE values
21 return success
```

from lines 5, lines 7 and 8 check if the node has been already mapped. The algorithm continues further only if the node is not mapped, otherwise the next node in the queue is selected. Lines 9 and 10, $Get\_Mapped\_Pred()$ and $Get\_Mapped\_Succ()$ routines, return only the predecessors and successors of the current node that are already mapped. $Get\_Connected\_PEs()$ function, in line 11, returns all the possible free PEs that are connected to the mapped predecessor and successors from the Modulo Resource Routing Graph (MRRG). PathSeeker starts the mapping in a reverse breadth-first search graph traversal (using an adjacency list $AList$) to aid the mapping of predecessors easily. This design decision was taken by analyzing the loops considered for the experiments. Since the nodes are already scheduled to a time-slot before mapping, taking a reverse breadth first search (BFS) approach will aid the mapping of predecessor node with fewer mapping failures. On the contrary, when we analyzed the breadth first search with predecessors mapped first followed by the successor nodes, due to the random placement of the predecessors, there was a high chance that the predecessor nodes are placed in non-connected PEs, which resulted in a successor node mapping failure.

Based on the size of the Γ from line 11, the placement (PE) for the node v is chosen. If Γ size is one then that PE is chosen in line 13, or if more than one possible PE positions are available a random PE is chosen from line 15. An empty Γ means that there were no possible placements available for the node v. At this juncture, PathSeeker employs a three-tier recovery approach to find a valid placement for node v. In line 17, the $Localized\_Search()$ routine is invoked. On a failure of this routine, in line 18 $Recovery\_Level\_One()$ routine is called. On a Level One failure, $Recovery\_Level\_Two()$ routine is employed, in line 19. The Level One and Level Two routines use complex time-slot level remapping procedures to find a valid mapping for node v. All the three recovery methodologies are explained in detail in the following subsections. In an event of all three recovery failures, the PathSeeker algorithm is restarted for the given II. When a valid mapping is found, all the possible placements i.e., Γ, and the selected PE are stored for recovery purposes by $SetMappablePositions()$ and $SetCurrentPosition()$ routines, respectively in lines 21 and 22. Line 23-26 selects the next adjacent node to v and adds it to the queue to continue the mapping procedure.

On a failure to map a node PathSeeker invokes the $Localized\_Search()$ algorithm shown in Algorithm 2. The algorithm searches through the possible positions of the predecessors and the successors to find a valid placement for the failed node. Lines 6-17 search through both predecessors' and successors' possible positions when there are mapped predecessors and successors for the failed node. This localized

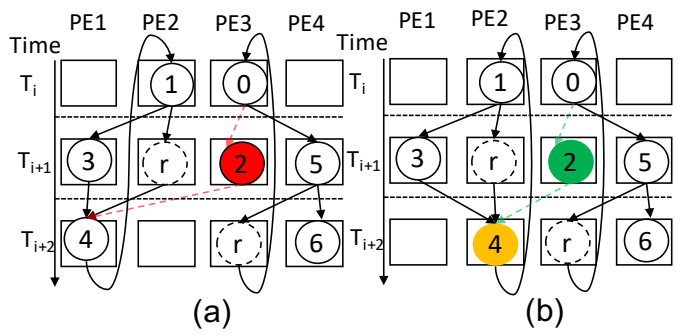

Fig. 2. (a) Mapping failure due to GraphMinor. (b) PathSeeker's localized modifications results in a valid mappin faster.

search routine does not modify any other nodes that are already mapped onto the CGRA. The $GetMappablePositions()$ function in lines 8 and 11 retrieves the possible PE positions stored in line 21 in Algorithm 1. On finding a successful connected PE, the PE position for node v (failure node) is store into Γ array. A valid mapping is obtained for node v only if all the predecessors and successors have a connected PE to v. Lines 16 and 17 updates the predecessor nodes and successor nodes, which is used to check if all the predecessors and successors were able to successfully find a connected PE. On failure of this localized search (when Γ's size is 0), PathSeeker invokes the $Recovery\_Level\_One()$ routine, from Algorithm 1, line 18. Lines 10 and 12 are modified when the failed node does not have a predecessor mapped.

The $Recovery\_Level\_One()$ routine employs the novel timeslot level remapping. The remapping starts by collecting all the nodes mapped to current timeslot as that of the failed node. Next, the remapping algorithm iterates over the mappable positions of each node and remaps them. On remapping each node, the valid position for the failed node is checked. This local rearrangement of the already mapped nodes to the timeslot is the novelty of PathSeeker, and it helps to change the course of the mapping. On a successful mapping of the failed node, the current remapping of the nodes is finalized and their positions are updated.

Algorithm 1 line 19, calls the $Recovery\_Level\_Two()$ routine on a Level One failure. Developed from Level One, the $Recovery\_Level\_Two()$ algorithm, not only remaps the nodes in the failure node's (node v) timeslot, but also remaps the nodes present in the successors' and the predecessors' timeslots. On a Recovery Level Two failure, the PathSeeker algorithm restarts with a new design space to be explored.

V. Running Example

Fig. 2(a)&(b) shows the working of the PathSeeker technique on a failure to map node 2. Fig. 2(a) shows the failure to map node 2 encountered by GraphMinor. For the failure in Fig. 2(a), PathSeeker's $Localized\_Search$ function is invoked first which gets the predecessors and successors of failed node 2, i.e., node 0 and node 4, respectively. PathSeeker iterates through all the possible positions to find a valid mapping of the successor, and consecutively the predecessor. There is just one possible position for node 4, i.e., $PE3$, which meets all the dependencies. A valid mapping by PathSeeker for this failure case is shown in Fig. 2(b). PathSeeker calls the $Localized\_Search$ to modify the path mapping of 4 to find a valid mapping for 2. It can be observed that PathSeeker's $Localized\_Search$ does not modify the placements of other mapped nodes and instead only explores within the existing mapping. In a hypothetical case where there is no possible mapping available for node 2, Level One recovery routine will be called to $remap$ the nodes in time-slot T_{i+1}. On a failure to find a valid mapping from Level One recovery, Level Two recovery function will be called to remap the nodes in time-slot T_{i+2} and subsequently the nodes in time-slot T_i, which are the successor and predecessor time-slots of node 2. While previous techniques explore the design space on a node-by-node basis, PathSeeker explores the mapping space on a time-slot level.

VI. Experimental Results

Setup: We profiled applications from three widely used benchmark suites[4] MiBench [15], Rodinia [16], and Parboil [17]. These benchmarks depict a wide variety of application domains comprising of embedded system applications like automotive, industry, office, network, security, and telecommunication, heterogeneous applications like data mining, pattern recognition, image processing, graph algorithms, and high performance computing application like spare matrix-dense vector multiplication (spmv). **Compilation:** The extraction of loops and converting them to Data Flow Graph (DFG) were performed using CCF [24], an LLVM 4.0 [25] based CGRA compilation and simulation framework. We have implemented partial predication [26], for compiling loops with conditionals. We have also implemented path-sharing, proposed in GraphMinor [11]. RAMP [12], GraphMinor [11], and PathSeeker (proposed technique) mapping algorithms as passes in CCF. We compiled the application loops with optimization level 3, to avoid those loops that are vectorizable by the compiler. We scaled the three mapping algorithms across five CGRA sizes, namely 4×4, 5×5, 6×6, 7×7, and 8×8 for scalability. We avoided loops with system calls as they cannot be accelerated on CGRA.

A. PathSeeker maps all the loops on 4×4 CGRA at a lower II.

Fig.3 and Fig. 4 shows the performance comparison of PathSeeker with GraphMinor and RAMP. The values were recorded by executing PathSeeker, RAMP and GraphMinor on an Intel-i7 running at 2.8 GHz with 16 GB memory. A 4×4 CGRA was used for this experiment. The compilation time threshold was kept at 100,000 seconds[5]. It can be inferred from Fig.3 and Fig. 4 that PathSeeker, with its novel remapping scheme was able to map all the 35 loops considered, whereas GraphMinor and RAMP were not able to map 20 and 5 loops, respectively. The loops for which a valid mapping cannot be

[4]Top two performance-critical loops were chosen from each application, with each contributing $> 7\%$ of the execution time of the application when executed with standard inputs that were shipped with the benchmark suites.

[5]A 100,000 seconds threshold time is 27 hours, more than a day to find a valid mapping.

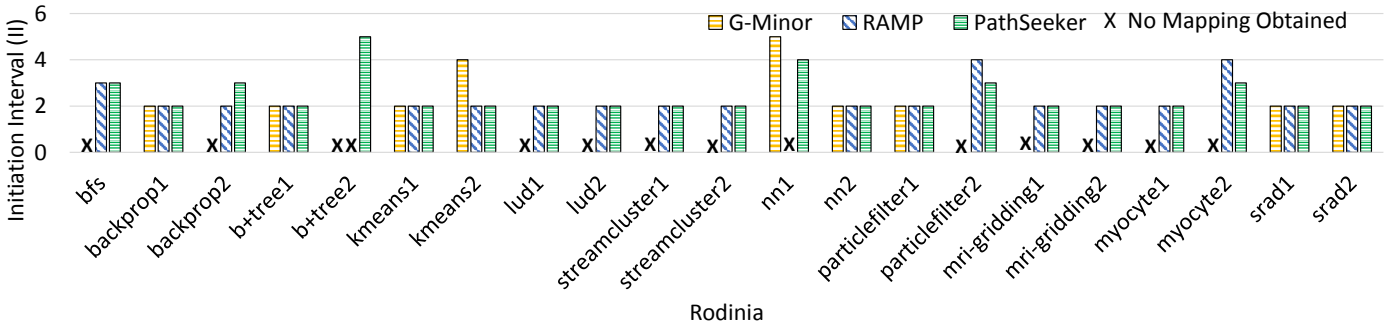

Fig. 3. Performance (II) comparison of PathSeeker with GraphMinor (G-Minor) and RAMP for application loops from Rodinia on a 4×4 CGRA.

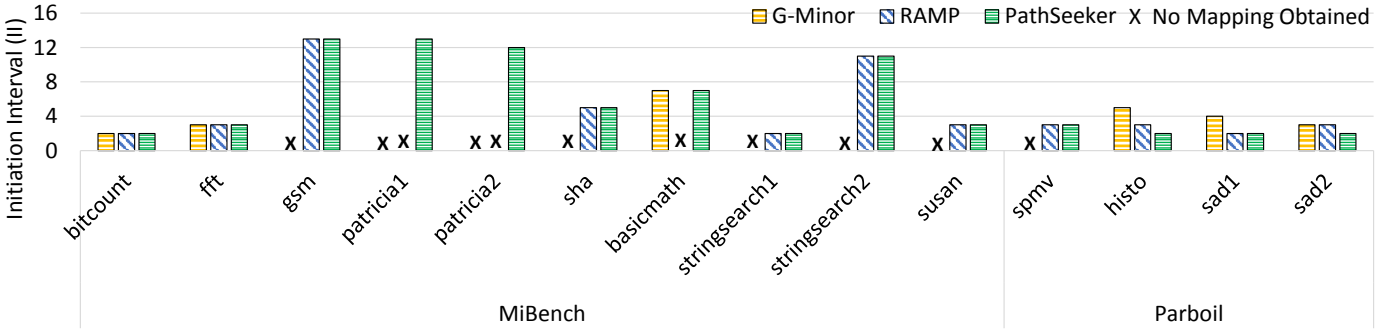

Fig. 4. Performance (II) comparison of PathSeeker with GraphMinor (G-Minor) and RAMP for application loops from MiBench and Parboil on a 4×4 CGRA.

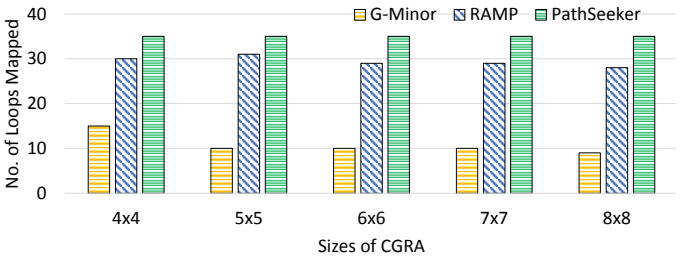

Fig. 5. PathSeeker achieved a valid mapping for the all the 35 loops across various sizes of CGRA.

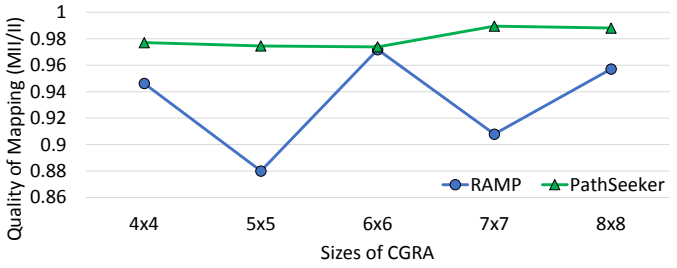

Fig. 6. PathSeeker achieves a superior mapping quality (II closer to MII) compared to RAMP.

obtained within 100,000 seconds are denoted by "**X**" in the Fig. 3 and Fig. 4.

The II obtained from GraphMinor and RAMP are not always optimal (lower II is better). This can particularly be noted in loops such as $kmeans2$, $nn1$, $histo$ and $sad1$ where GraphMinor had higher II, and $particlefilter2$, $myocyte2$, $histo$, and $sad2$ for which RAMP had higher II. Considering the loops for which the GraphMinor has obtained a valid mapping, PathSeeker showed a 28% lower II. Compared to RAMP, PathSeeker achieved a comparable performance in all the loops and had better performance in five loops mentioned above.

B. PathSeeker maps all the loops across all the CGRA sizes with a better quality.

We performed the scalability experiment for CGRA sizes of 5×5, 6×6, 7×7, and 8×8. Fig 5, shows the scalability of PathSeeker with respect to GraphMinor and RAMP. We can observe that as the size of the CGRA increases the number loops mappable by GraphMinor and RAMP reduces. PathSeeker, on the other hand, is able to achieve a valid mapping for all the 35 loops considered. Due to the backtracking mechanism, GraphMinor was not able to find a mapping within the threshold of almost 75% of the loops. Fig 5 clearly shows that arbitrary backtracking to the previously mapped nodes on encountering a mapping failure is not a scalable solution.

The Minimum II (MII) is the minimum possible II that can be achieved for a given loop DFG and the CGRA architecture. The quality of mapping of a mapping algorithm is the ratio of $\frac{MII}{II}$, which indicates how close the obtained II is to the MII. Fig 6, shows the quality of mapping of RAMP and PathSeeker across all the five CGRA sizes. GraphMinor was not considered due to its inability to find a mapping for more than 70% of the loops. The mapping quality achieved by PathSeeker is better and consistent across all the CGRA sizes, compared to RAMP.

C. PathSeeker is a fast mapping algorithm.

Fig. 7, shows the scaling of average compilation times of RAMP and PathSeeker, considering only the loops that were mappable by RAMP. The y-axis of Fig. 7 shows the average

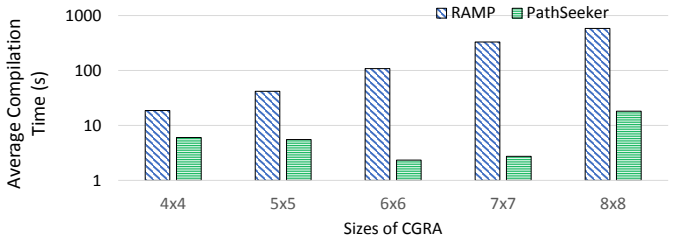

Fig. 7. PathSeeker achieves a mapping for all the loops across various sizes of CGRA at a lower compilation time.

compilation time across all the benchmark loops for which RAMP was able to achieve a valid mapping, in log scale, and the x-axis shows the various sizes of the CGRA. As shown in Fig. 7, the compilation time of RAMP increases exponentially, due to the restart mechanism on encountering a failure and its algorithmic complexity. In comparison, the compilation time of PathSeeker scales linearly, due to the initial randomized placement of the nodes and localized modifications of the mapping pertaining to the failed nodes.

VII. Summary

This paper presented a novel CGRA mapping scheme, PathSeeker, that was able to map all the loops in a smaller CGRA size, with better II and lower compilation time. Existing techniques, such as GraphMinor and RAMP, resort to backtracking to a previously mapped node or restarting the mapping process, when encountering a mapping failure. This leads to a significant increase in the compilation time and poor II. PathSeeker's novelty lies in employing localized search strategies and time-slot level remapping to rectify a mapping failure. PathSeeker was able to map all the 5 top performance-critical loops across three widely used benchmark suite loops on a 4×4 CGRA, whereas GraphMinor and RAMP were not able to map 20 and 5 loops on the same CGRA size, respectively. On comparing the loops that were mappable by GraphMinor and RAMP, PathSeeker achieved a 28% lower II compared to GraphMinor and 3% lower II compared to RAMP on a 4×4 CGRA. PathSeeker was able to get a 550x and 10x compilation time improvement compared to GraphMinor and RAMP, respectively.

Acknowledgment

This work was partially supported by National Science Foundation grant CPS 1645578.

References

[1] National Research Council et al. *Frontiers in massive data analysis*. National Academies Press, 2013.

[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] 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.

[5] 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.

[6] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[7] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[8] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques*, 150(5):255, 2003.

[9] 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.

[10] 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.

[11] 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.

[12] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Panagiotis Theocharis and Bjorn De Sutter. A bimodal scheduler for coarse-grained reconfigurable arrays. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–26, 2016.

[21] 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.

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

[23] Ashay Dharwadker. The clique algorithm, 2006.

[24] Shail Dave and Aviral Shrivastava. Ccf: A cgra compilation framework, 2018.

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

[26] 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.