THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRB6/13

Graph Minor Approach for Application Mapping on CGRAs

Liang Chen and Tulika Mitra

June 2013

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Graph Minor Approach for Application Mapping on CGRAs

Liang Chen, Tulika Mitra
School of Computing
National University of Singapore

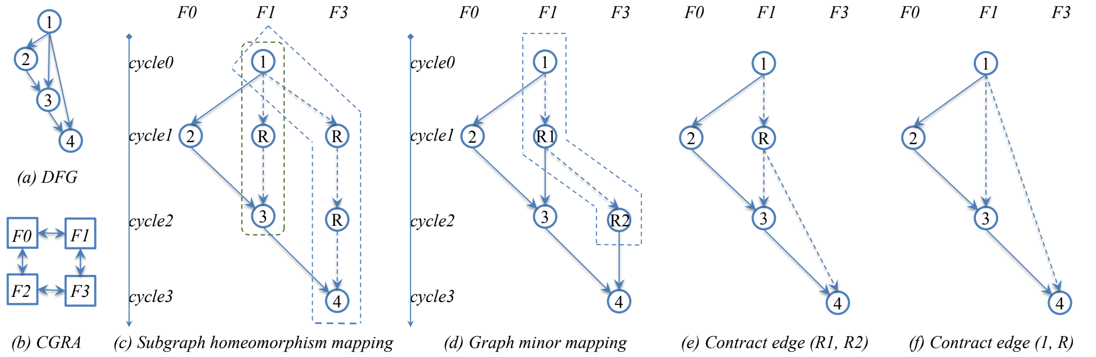
Abstract

Coarse-grained reconfigurable arrays (CGRAs) exhibit high performance, improved flexibility, low cost, and power efficiency for various application domains. Compute-intensive loop kernels are mapped onto CGRAs through modified modulo scheduling algorithms that integrate placement and routing. We formalize the CGRA mapping problem as a graph minor containment problem. We essentially test if the data flow graph representing the loop kernel is a minor of the modulo routing resource graph representing the CGRAs resources and their interconnects. We design an exact graph minor testing approach that exploits the unique properties of both the data flow graph and the routing resource graph to significantly prune the search space. We introduce additional heuristic strategies that drastically improve the compilation time while still generating optimal or near-optimal mapping solutions. Experimental evaluation confirms the efficiency of our approach.

1 Introduction

Coarse-Grained Reconfigurable Arrays (CGRA) are a promising alternatives between ASICs and FPGAs. Traditionally in embedded systems, compute intensive kernels of an application are implemented as ASIC, which has high efficiency but limited flexibility. Current generation embedded systems demand flexibility to support a diverse range of applications. FPGAs provide high flexibility, but suffer from low efficiency. To bridge this gap, CGRA architectures have been proposed such as CHESS [30], MorphoSys [39], ADRES [31], DRAA [28], FloRA [27] and many others. Most of these architectures arrange coarse-grained function units (FUs) in a mesh-like structure. The FUs in a CGRA can be reconfigured by writing to a control register (or context register) on a cycle by cycle basis.

The compute-intensive loop kernels are perfect candidates to be mapped to CGRAs containing multiple functional units targeting high instruction-level parallelism. Software pipelining techniques, example, modulo scheduling, are thus introduced to map applications onto CGRAs. A number of CGRA mapping algorithms [32, 6, 4, 22, 16, 12, 17] are inspired by compilation techniques for VLIW architectures as well as FPGA synthesis. For example, CGRA mapping algorithms adopt placement and routing techniques from FPGA synthesis domain and modulo scheduling from VLIW compilation process. It is important to note, however, that the inherent structure of the CGRAs is very different from both FPGAs and VLIW architecture. More concretely, the connectivity among the functional units in CGRAs is usually fixed unlike FPGAs where the interconnection can be reconfigured. Thus the mapping algorithms based on FPGA place and route techniques may find it challenging to identify feasible routing paths in fixed interconnect structure of CGRAs. Similarly, unlike VLIW architectures where all the FUs typically share a common register file, the FUs in most CGRAs have limited and explicit connection to the register files. Thus it is not prudent to perform register allocation as a post-processing step as is commonly done in VLIW scheduling. Instead, register allocation should be integrated in the early stage with scheduling (place and route) to achieve quality mapping.



(a) DFG

(b) CGRA

(c) Subgraph homeomorphism mapping

(d) Graph minor mapping

(e) Contract edge (R1, R2)

(f) Contract edge (1, R)

Figure 1: Subgraph homeomorphism versus Graph Minor formulation of CGRA mapping problem.

In this work, we focus on developing a comprehensive solution for the general CGRA mapping problem. Our goal is to first formalize the CGRA mapping problem. We note that a number of recent works [40, 3, 7, 17, 18] in the literature formulate the CGRA mapping problem as subgraph homeomorphism problem [15]. The idea is to test if the data flow graph (DFG) representing the loop kernel is subgraph homeomorphic to the modulo resource routing graph (MRRG) representing the CGRA resources and their interconnects. Homeomorphism formulation allows subdivision of the DFG edges when being mapped onto the MRRG, i.e., a DFG edge can be mapped as a chain of edges (path) on the MRRG. Alternatively, additional vertices on a path consisting of a chain of edges can be smoothed out to create a single edge. The additional nodes due to sub-division model the routing of data from the source to the target FUs if they are not connected directly. However, subgraph homeomorphism requires the edge mappings to be node-disjoint (except at end points) or edge-disjoint [15]. While subgraph homeomorphism provides an elegant formulation of the CGRA mapping problem, it excludes the possibility of sharing the routing nodes among single source multiple target edges [35] (also called multi-net [12]) leading to possible wastage of precious routing resources.

Figure 1 illustrates subgraph homeomorphism formulation. Figure 1(a) shows a simple DFG (for simplicity we have removed the loop back edge) being mapped onto a 2x2 homogeneous mesh CGRA shown in Figure 1(b). The DFG is homeomorphic to the subgraph of the MRRG shown in Figure 1(c) and thus the subgraph represents a valid mapping (again for simplicity we have removed additional nodes of the MRRG). In this homeomorphic mapping, edges (1,3) and (1,4) have been routed through three additional routing nodes marked by R. Notice that each routing node has degree 2 and has been added through edge subdivision (marked by dashed edges). Alternatively, the routing nodes in the MRRG subgraph can be smoothed out to obtain the original DFG. As mentioned earlier, by definition, edge subdivision cannot support route sharing.

In contrast, we model the CGRA mapping problem as graph minor containment problem which can explicitly model route sharing. A graph H is a minor of graph G if H can be obtained from a subgraph of G by a (possibly empty) sequence of edge contractions [37]. In graph theory, an edge contraction removes an edge from a graph while simultaneously merging the two vertices it previously connected. In our context, we need to test if the DFG is a minor of the MRRG, where the edges to be contracted represent the routing paths in the MRRG. Unlike edge subdivision (or its reverse operation smoothing), edge contractions are not restricted to simple paths. Thus graph minor formalism naturally allows for route sharing. Figure 1(d) shows a mapping under graph minor approach. It is a subgraph of the MRRG from which the DFG can be derived through two edge contractions as shown in Figure 1(e)-(f). In this example, we reduce the number of routing nodes from 3 (in subgraph homeomorphism mapping) to 2 (in graph minor mapping). While it is possible to support route sharing in [35, 12], we provide a clear formalization of the CGRA mapping problem under route sharing. This formalization also enables guided search in the mapping process to achieve optimal or near-optimal solutions.

We propose a customized exact graph minor containment testing approach in our context that fully exploits the structure of the DFG and the CGRA interconnects to effectively navigate and prune the mapping alternatives. We further optimize the minor testing procedure by introducing heuristics that drastically accelerate the compilation process while still preserving the quality of the solutions to large extent. Experimental evaluation confirms that our approach leads to high quality mapping both in terms of performance and resource usage with minimal compilation time.

In parallel to our graph minor formalization [9] for CGRA mapping problem, [20] proposed graph epimorphism formalization for the same problem. Their approach, called EPIMap, is quite elegant and models the novel concept of re-computation in addition to route sharing. Re-computation allows for the same operation to be performed on multiple FUs if it leads to better routing. In EPIMap approach, the DFG H is morphed into another graph H' (through introduction of routing/re-computation nodes and other transformations) such that there exists subgraph epimorphism from H' to H (many to one mapping of vertices from H' to H and adjacent vertices in H' map to adjacent vertices in H). Then EPImap attempts to find the maximal common subgraph (MCS) between H' and the MRRG graph G using standard MCS identification procedure. If the resulting MCS is isomorphic to H', then a valid mapping has been obtained; otherwise H is morphed differently in the next iteration and the process repeats.

The key difference with our approach is that while we develop a customized graph minor testing procedure that exploits structural properties of our graphs, EPIMap relies on off-the-shelf MCS identification algorithm. This can potentially lead to faster compilation time for graph minor approach. Both approaches introduce heuristics to manage the computational complexity; the transformation of the DFG as well as MCS identification require heuristics in EPIMap, while graph minor approach restricts the edge mapping choices. Thus, the quality of the solutions in both approaches depend on the loop kernel and the underlying CGRA architecture. On the other hand, the re-computation concept in EPIMap enables additional scheduling and routing options that can potentially generate better quality solutions for certain kernels. Finally, graph epimorphism and graph minor are quite unrelated concepts, the discussion on this topic is out of scope here. Instead, we provide quantitative experimental comparison of the two approaches in Section 6.

2 Related work

Mapping a compute-intensive loop kernel of an application to CGRAs using modulo scheduling was first discussed in [32]. In this simulated annealing based approach, the cost function is defined according to the number of over-occupied resources. The simulated annealing approach can have long convergence time, especially for large dataflow graphs. Routing through register files and register allocation problems are further explored in [12], which extends the work in [32]. Register allocation is achieved by constraining the register usage during the simulated annealing place and route process. The imposed constraint is adopted from meeting graph [14] for solving loop cyclic register allocation in VLIW processors. Post routing, the registers are allocated by finding a Hamilton circuit in the meeting graph, which is solved as a traveling salesman problem [12]. This technique is specially designed for CGRAs with rotating register files. [22] also follows the simulated annealing framework but aims at finding better cost functions for overused resources. SPR [16] is a mature CGRA mapping tool that successfully combines the VLIW style scheduler and FPGA placement and routing algorithms for CGRA application mapping. It consists of three individual steps namely scheduling, placement, and routing. The placement step of SPR also uses the simulated annealing approach.

List scheduling has been adopted in [6, 5, 4], which analyzes priority assignment heuristics under different network traversal strategies and delay models. The heuristics utilize the interconnect information to ensure that data dependent operations can be mapped spatially close to each other. [34] also gives priorities for operations and resources to obtain a quality schedule. The priorities are given respect to the importance factors of explicit routing from producer nodes to consumer nodes. This idea is further exploited in edge-centric modulo scheduling (EMS) [35], where it is pointed out that the primary objective should be routing efficiency rather than operation assignments. The quality of a mapping using specific priorities highly depends on efficient heuristics for assigning these priority values to both operations and resources. The design of priority heuristics still remain challenging.

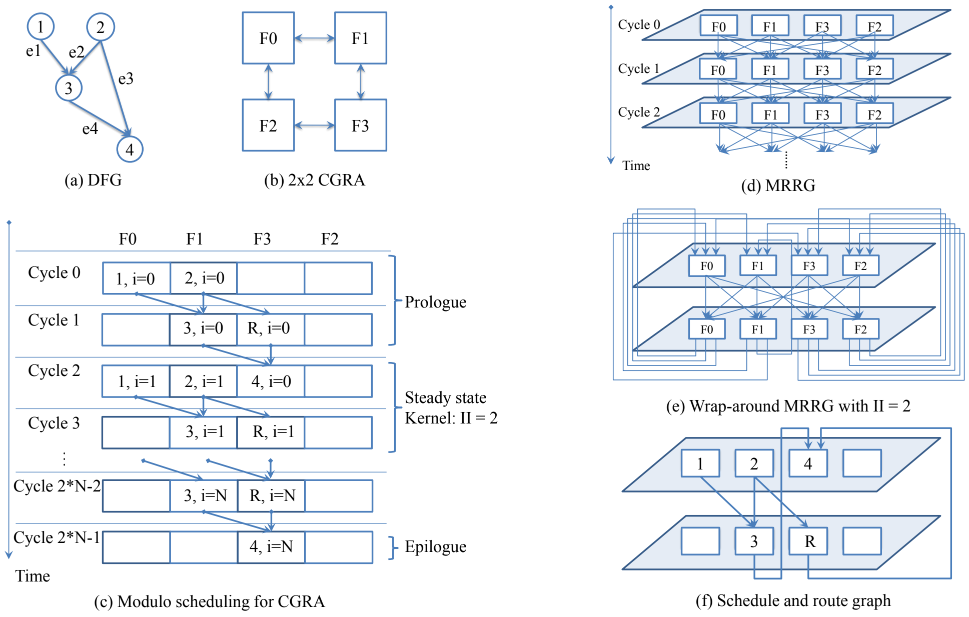
There are various approaches to CGRA mapping using techniques from graph theory domain. [10] integrates subgraph isomorphism algorithm to generate candidate mapping between a DFG and the resource graph of a coarse-grained accelerator. The reconfigurability and additional routing features are not considered. SPKM [42, 43] adopts the split and push technique [13] for planar graph drawing and focuses on spatial mappings for CGRAs, where the reconfigurability is not supported. The mapping in SPKM starts from an initial drawing where all DFG nodes reside in the same group. One group represents a single functional unit. The group is then split into two and a set of nodes are pushed to the newly generated group. The split process continues till each group contains only one node, which represents a one-to-one mapping from DFG to the planar resource graph of CGRA.

A number of CGRA mapping approaches follow the subgraph homeomorphism formalizations including [40, 3, 7, 17, 18]. The mapping algorithm in [40] is adapted from MIRS [44], a modulo scheduler capable of instruction scheduling with register constraints. The adaptations for CGRA mapping include an incorporated cost function for routing and considerations for conditional branches. [3] partitions the DFG into substructures called HyperOps and these HyperOps are synthesized into hardware configurations. The synthesis is carried out through a homeomorphic transformation of the dependency graph of each HyperOp onto the resource graph. [7] also formalizes the CGRA application mapping problem as a subgraph homeomorphism problem. However, it considers general application kernels rather than loops. Particle swarm optimization is adopted for solving CGRA mapping problem in [17, 18]. The calculation for fitness, which is used to move particles (DFG nodes) in particle swarm optimization, is specifically designed to optimize multiple objectives from the routing paths.

EPIMap [20] formalizes the CGRA mapping problem as an graph epimorphism problem with the additional feature of re-computations. The core of this approach consists of a subgraph isomorphism solver which finds the maximum common subgraph (MCS) [29] between the DFG and the resource graph of CGRA. The idea is to transform the DFG by inserting dummy routing nodes or replicated operation nodes so that the routing requirements could be satisfied through multiple runs of the core subgraph isomorphism solver. EPIMap can generate better scheduling results compared to EMS with similar compilation time. Most graph approaches solve a subset of the epimorphism problem defined in EPIMap.

3 Modulo Scheduling for CGRA

Given a loop from an application and a CGRA architecture, the goal of mapping is to generate a schedule such that the application throughput is maximized. The loop is represented as a data flow graph (DFG) where the nodes represent the operations and the edges represent the dependency among the operations. Figure 2(a) shows the DFG of a simple loop. Figure 2(b) shows a 2x2 CGRA consisting of four functional units (FUs) where the loop should be mapped to. The mapping problem consists of (a) scheduling the operations in space and time so as to satisfy the dependency constraints, and (b) explicit routing of the operands from the producers to the consumers.



(a) DFG

(b) 2x2 CGRA

(d) MRRG

(c) Modulo scheduling for CGRA

(e) Wrap-around MRRG with II = 2

(f) Schedule and route graph

Figure 2: Modeling of loop kernel mapping on CGRAs: An illustrative example.

3.1 Modulo Scheduling

Modulo scheduling is a software pipelining technique used to exploit instruction-level-parallelism in the loops by overlapping consecutive iterations [36]. The schedule produced includes three phases: the prologue, the kernel, and the epilogue. The kernel corresponds to the steady state execution of the loop and comprises of operations from consecutive iterations. The schedule length of the kernel, which is also the interval between successive iterations, is called the initiation interval (II). If the number of loop iterations is high, then the execution time in the kernel is dominant compared to the prologue and the epilogue. Thus, the goal for modulo scheduling is to minimize the II value. Initially, the scheduler selects the minimal II (MII) value between resource-minimal II and recurrence-minimal II, and attempts to find a feasible schedule with that II value. If the scheduling fails, then the process is repeated with an increased II value.

Figure 2(c) shows the modulo-scheduled version of the loop in Figure 2(a) to the CGRA architecture in Figure 2(b) with prologue, kernel, and epilogue where II=2. Notice that *operation 4* from the i^{th} iteration is executing in the same cycle with *operation 1* and *operation 2* from the $(i+1)^{th}$ iteration in the steady state. Also, we need to hold the output of *operation 2* in an routing node (R) till it gets consumed by *operation 4*. This explicit routing between FUs is what sets apart modulo scheduling in CGRA from conventional modulo scheduling. In conventional modulo scheduling, function units are fully connected as routing is guaranteed through the central RF. In CGRAs, the modulo scheduler has to be aware of the details of the underlying interconnections among the FUs and the register files (RF), to route data.

3.2 Modulo Routing Resource Graph (MRRG)

Mei et al. [32] defined a resource management graph for CGRA mapping, called Modulo Routing resource graph (MRRG), which has been used extensively in subsequent research [22, 34, 35, 16]. The MRRG captures the interconnections among the FUs and the RFs. In MRRG, the resources are presented in a time-space view. The nodes represent the ports of the FUs and the RFs, and the edges represent the connectivity among the ports.

We adopt a simplified form of MRRG proposed in [34] where a node corresponds to FU or RF rather than the ports. Our mapping technique integrates register allocation with scheduling. We model each RF as one node per cycle in the MRRG. The individual registers within RF are treated as identical elements and represented by the capacity of the RF as in *compact register file model* [12]. The usage of registers is tracked and constrained during the mapping procedure. The number of read and write ports per RF is also included as a constraint. Modeling the RF as a single node reduces the complexity of the MRRG and accelerates the mapping algorithm.

The MRRG is a directed graph G_{II} where *II* corresponds to the initiation interval. Given a graph G, we denote the vertex set and the edge set of G by $V(G)$ and $E(G)$, respectively. Each node $v \in V(G_{II})$ is a tuple *(n, t)*, where *n* refers to the resource (FU or RF) and *t* is the cycle. Let $e = (u, v) \in E(G_{II})$ be an edge where *u = (m, t)* and *v = (n, t+1)*. Then the edge *e* represents a connection from resource *m* in cycle *t* to resource *n* in cycle *t+1*. If resource *m* is connected to resource *n* in the CGRA, then node *u = (m, t)* is connected to node *v = (n, t+1)* for $t \geq 0$.

For example, Figure 2(d) shows the MRRG corresponding to the CGRA shown in Figure 2(b). The resources of the CGRA are replicated every cycle along the time axis, and the edges always point forward in time. During modulo scheduling, when a node *v=(n, t)* in the MRRG becomes occupied, then all the nodes *v'=(n, t+k×II)* (where $k > 0$) are also marked occupied. For example, in the modulo schedule with II=2 shown in Figure 2(c), as F1 is occupied by *operation 1* in cycle 0, it is also occupied by *operation 1* every $2 \times k$ cycle. In most CGRA mapping techniques, this modulo reservation for occupied resources is done through a modulo reservation table introduced in [32].

3.3 MRRG with Wrap-around Edges

The goal of CGRA modulo scheduler is to generate II different configurations for the CGRA where each configuration corresponds to a particular cycle in the kernel. These configurations are stored in a configuration RAM and provide configuration context to the CGRA every cycle. As these configurations are repeated every II cycles, the output from the resources involved in the last cycle configuration are consumed by resources involved in the first cycle configuration. Thus instead of using MRRG where the time axis grows indefinitely till the steady state is achieved, we could restrict the time axis to the target II. We then need to add wrap around edges from the last cycle to the first cycle as shown in Figure 2(e) (similar graph is also used in [16]). The modulo scheduled kernel in Figure 2(c) can now be simplified to the graph in Figure 2(f). We refer to this simplified graph as *schedule and route graph (SRG)*, which captures the scheduling plus routing information and is a subgraph of the MRRG. So instead of using a modulo reservation table, we can directly use MRRG with wrap around edges, which provides us an integrated view during mapping. *In the following, the term MRRG will be used to refer to MRRG with wrap around edges.*

4 CGRA Mapping Problem Formalization

We first present the formalization of the CGRA mapping problem in the form of subgraph isomorphism when no data routing is required and subgraph homeomorphism when routes are not shared. We then model the application mapping problem on CGRAs as a graph minor problem [37] between the DFG and the MRRG in the presence of route sharing. Meanwhile, we point out the necessary restrictions imposed in the formalization. We also provide the NP-completeness proof for the CGRA mapping problem under our graph minor formalization.

4.1 Subgraph Isomorphism and Subgraph Homeomorphism Mapping

Let H be a directed graph representing the DFG and G_{II} be a directed graph representing the MRRG with initiation interval II. We are looking for a mapping from the input graph *H* to the target graph *G*. In the ideal scenario of full connectivity among the FUs, all the data dependencies in the DFG can be mapped to direct edges in the MRRG. That is, for any edge $e = (u, v) \in E(H)$, there is an edge $e = (f(u), f(v)) \in E(G)$ where f represents the vertex mapping function from the DFG to the MRRG. This matches the definition of subgraph isomorphism in graph theory. Thus the CGRA application mapping problem can be solved using techniques for subgraph isomorphism from the graph theory domain [41, 11].

In reality, however, data may need to be routed through a series of nodes rather than just using direct links. For example, the edges $(1, 3)$ and $(1, 4)$ in Figure 1(a) are routed through additional nodes. If an edge $e = (u, v) \in E(H)$ in the DFG can be mapped to a path from *f(u)* to *f(v)* in the MRRG *G*, it matches the subgraph homeomorphism definition [15]. The subgraph homeomorphism techniques for CGRA mapping problem has been explicitly adopted in [40, 3, 7, 17, 18]. Subgraph homeomorphism, however, requires the edge mappings to be node-disjoint (or edge-disjoint) [15], which means the nodes (or the edges) in the mapping paths for the edges carrying the same data cannot be shared. We now show how the CGRA mapping problem can be formalized in the presence of route sharing.

4.2 Graph Minor

We now present graph minor [37] based formulation of the application mapping problem on CGRAs with route sharing. In graph theory, an undirected graph *H* is called a minor of the graph *G* if *H* is isomorphic to a graph that can be obtained by zero or more edge contractions on a subgraph of *G*. An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. More formally, a graph *H* is a minor of another graph *G* if a graph isomorphic to *H* can be obtained from *G* by contracting some edges, deleting some edges, and deleting some isolated vertices. The order in which a sequence such operations is performed on *G* does not affect the resulting graph *H*.

A *model* of H in G is a mapping ϕ that assigns to every edge $e \in E(H)$ an edge $\phi(e) \in E(G)$, and to every vertex $v \in V(H)$ a non-empty connected subgraph $\phi(v) \subseteq G$ such that

1. the graphs $\{\phi(v)|v \in V(H)\}$ are mutually vertex-disjoint and the edges $\{\phi(e)|e \in E(H)\}$ are pairwise distinct; and

2. for $e = \{u, v\} \in E(H)$, the edge $\phi(e)$ connects subgraph $\phi(u)$ with subgraph $\phi(v)$.

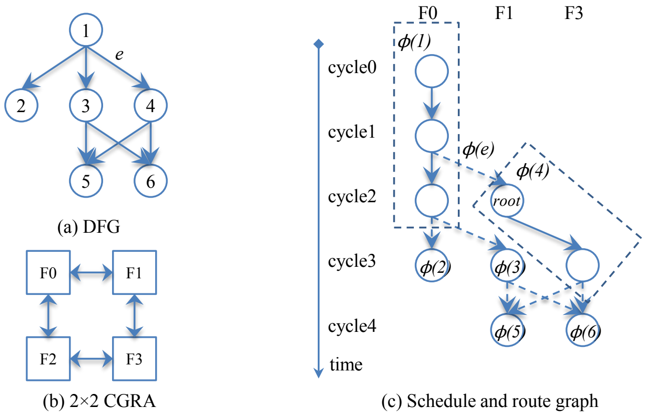
Thus, H is isomorphic to a minor of G if and only if there exists a model of H in G [2].

4.3 Adaptation of Graph Minor for CGRA Mapping

We need to adapt and restrict the definition of graph minor for our problem. First, graph minor is usually defined for undirected graphs. For directed graphs, the definition of edge contraction is similar to the undirected case [38]. Figure 1(e)-(f) show examples of directed edge contractions.

We call the subgraph $M \subseteq G$ defined by the union of $\{\phi(v)|v \in V(H)\}$ and $\{\phi(e)|e \in E(H)\}$ as the schedule and route graph (SRG) of H in G. The SRG M is essentially the model of H in G. The edge set of M is partitioned into the *contraction edges* (the edges in $\{\phi(v)|v \in V(H)\}$) and the *minor edges* (the edges in $\{\phi(e)|e \in E(H)\}$). The minor edges support the data dependencies in the dataflow graph, while the contraction edges represent data routing through additional nodes. For example, in Figure 1(d), $\phi(1)$ is the subgraph inside the dashed region rooted at node 1. The dashed edges are the contraction edges, while the solid edges are the minor edges.

Minor edge constraint In graph minor definition, for $e = (u, v) \in E(H)$, the minor edge $\phi(e)$ connects $\phi(u)$ with $\phi(v)$. In other words, it is sufficient for $\phi(u)$ to connect any node in the subgraph $\phi(u)$ with any node in the subgraph $\phi(v)$. However, for our problem, we need to define one particular node in the subgraph $\phi(v)$ where the actual operation $\phi(v)$ takes place and



(a) DFG

(b) 2×2 CGRA

(c) Schedule and route graph

Figure 3: Minor relationship between DFG and MRRG

it has to receive all the required inputs. The remaining nodes in $\phi(v)$ are used to route the result of the operation. More concretely, for our mapping, each subgraph $\phi(v) \subseteq G$ is a tree rooted at the node where the computation takes place. Let $root(\phi(v))$ be the root of the tree $\phi(v)$. Then we introduce the restriction that for $e = (u, v) \in E(H)$, the minor edge $\phi(e)$ connects $\phi(u)$ with $root(\phi(v))$. For example, the DFG in Figure 3(a) has an edge e that connects the DFG nodes 1 and 4, and it is mapped to a 2×2 CGRA shown in Figure 3(b). Then in the SRG, $\phi(1)$ has to connect to the root of $\phi(4)$ through a direct link $\phi(e)$ as shown in Figure 3(c).

Timing constraint The wrap-around nature of the MRRG introduces another restriction to the mapping. For an SGR M to be a valid mapping, it has to satisfy the timing constraints which we will detail in the followings.

For simplicity, let us first ignore the recurrence edges in the DFG. Then the DFG H is a directed acyclic graph. Let $u \in V(H)$ be a node in the DFG without any predecessor and $root(\phi(u)) = (m, t) \in M)$ where $0 \leq t < II$ and M is the SRG, a subgraph of the MRRG. That is, u has been mapped to the FU m in configuration (or cycle) t in the MRRG. We define the timestamp of u as $cycle(u) = t$ if $root(\phi(u)) = (m, t)$.

Let $v \in V(H)$ be a DFG node with $u \in V(H)$ as its predecessor node and $route(u, v)$ be the number of nodes (possibly zero) in the connecting path between $root(\phi(u))$ and $root(\phi(v))$ in the SRG M. Clearly, for a mapping M to be valid, the following timing constraint must be satisfied for each internal DFG node v. This constraint ensures identical cycle along all input edges of v.

$$\forall u, u' \in pred(v): \quad cycle(u) + route(u, v) = cycle(u') + route(u', v)$$

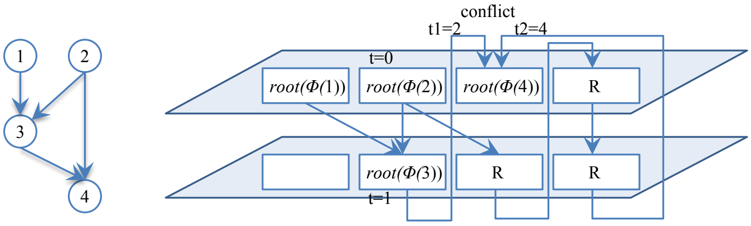
We also define

$$\forall u \in pred(v): \quad cycle(v) = cycle(u) + route(u, v) + 1$$

where $pred(v)$ is the set of all predecessors of v in the DFG. Note that we are not doing modulo operation (w.r.t. II) while computing the cycle values. Figure 4 shows this timing computation. In the SRG, $root(\phi(2))$ is in cycle 0 and $root(\phi(3))$ is in cycle 1. However, $root(\phi(2))$ has to go through three routing nodes to reach $root(\phi(4))$ and $root(\phi(3))$ can directly pass the data to $root(\phi(4))$ in the next cycle. This violates the timing constraint and leads to an invalid mapping.

For a recurrence edge $e = (u, v) \in V(H)$ with recurrence distance d in the DFG, we need to introduce additional timing constraint

$$route(u, v) = II \times d + cycle(u) - cycle(v)$$



(a) DFG

(b) Invalid schedule and route graph

Figure 4: Invalid mapping under timing constraint

Attribute constraint Each node in the DFG and the MRRG has an attribute that specifies the functionality of the node. For example, a node in the DFG can have memory operation as its attribute, while a node in the MRRG can have an attribute that signifies that it can support memory operations. Attribute constraint ensures that a DFG node is mapped to a MRRG structure with matching attribute. For example, the root of the tree subgraph for mapping a memory operation can only be a functional unit supporting memory accesses. However, any other nodes in the tree subgraphs can be functional unit or register.

Register file constraint The mapping must ensure availability of register file read/write ports and capacity in the corresponding cycle if a link from/to the register file is used in any node mapping.

Restricted Graph Minor We can now define application mapping on CGRAs as finding a valid subgraph (schedule and route graph) *M* of the MRRG such that the DFG can be obtained through repeated edge contractions of *M*. We call the DFG a restricted minor of the MRRG and the subgraph M represents the mapping. Alternatively, the DFG H is a minor of G if and only if there exists a model of H, represented by the schedule and route graph M, in G.

Lemma 1. *The restricted graph minor problem for directed graphs is NP-complete.*

Proof. We first show that the restricted graph minor problem for directed graphs is in the set of NP. Given a mapping in the form of SRG $M \subseteq G$, we can check in polynomial time (a) the graphs $\{\phi(v)|v \in V(H)\}$ are mutually vertex-disjoint and the edges $\{\phi(e)|e \in E(H)\}$ are pairwise distinct, (b) for $e = (u, v) \in E(H)$, the edge $\phi(e)$ connects subgraph $\phi(u)$ with $root(\phi(v))$, and (c) the timing constraints as defined earlier are satisfied. That is DFG H is a minor of the SRG M.

We now show that for general directed graphs, the restricted graph minor problem can be reduced to the Hamiltonian cycle problem, which is an NP-complete problem. The Hamiltonian cycle problem is to find a cycle in a directed graph *G* visiting each node exactly once. We can construct a graph *H* which is a directed cycle with $|V(G)|$ nodes. Finding the Hamiltonian cycle in *G* can now be reduced to finding a restricted graph minor between *H* and *G*. As $|V(G)| = |V(H)|$, each subgraph $\phi(v)$ can only consist of a single vertex and each edge mapping $\phi(e)$ where $e = (u, v) \in E(H)$ directly connects vertex $\phi(u)$ to vertex $\phi(v)$. This matches the exact definition of Hamiltonian cycle. Thus the restricted graph minor problem for directed graphs is NP-complete. □

5 Graph Minor Mapping Algorithm

Our solution for restricted graph minor containment problem is inspired by the tree search method (also called state space search) widely used to solve a variety of graph matching problems [33]. The contribution of our solution is the introduction of customized and effective pruning constraints in the search method that exploit the inherent properties of the data flow graph and the CGRA architecture. We first present the exact restricted graph minor containment algorithm followed by description of additional strategies to accelerate the search process.

5.1 Algorithmic framework

Our goal is to map a DFG H to the CGRA architecture. Similar to the traditional modulo scheduling, we start with the minimum possible II, which is the maximum of the resource constrained II and the recurrence constrained II, that is, $II = \max(ResMII, recMII)$. Given this II value, we create the MRRG G_{II} corresponding to the CGRA architecture. If H is a minor of G_{II}, then the DFG can be mapped with initiation interval II. To check graph minor containment, we check if there exists a model or mapping of H in the form of a valid SRG $M \subseteq G_{II}$. If such SRG M does not exist, we increment the II value by one, create the MRRG corresponding to this new II value, and perform graph minor testing for this new MRRG. This process is repeated till we have generated an MRRG with sufficiently large value of II so that the DFG can satisfy the graph minor test. Algorithm 1 provides a high-level view of our mapping framework.

The core routine of the mapping algorithm *Minor()* performs graph minor testing. We consider all possible mapping between the DFG and the MRRG; thus our algorithm is guaranteed to generate a valid mapping if it exists. Clearly, the number of possible mappings between the DFG and the MRRG is exponential in the number of nodes of the DFG. That is, our search space is large. Our goal is to either (a) quickly identify a mapping such that the DFG passes the restricted minor test, or (b) establish that no such mapping exists. As mentioned earlier, we employ powerful pruning strategies to efficiently navigate this search space. We also carefully choose the order in which we attempt to map the nodes and the edges so as to achieve quick success in finding a valid mapping or substantial pruning that helps establish the absence of any valid mapping.

The procedure *Minor()* starts with an empty mapping. As mentioned earlier, restricted graph minor mapping for our problem requires mapping each vertex $v \in V(H)$ in the DFG to a tree $\phi(v) \subseteq G$ in the MRRG. Each edge $e = (u, v) \in E(H)$ is simply mapped to an edge $\phi(e) \in E(G)$ that connects some node in $\phi(u)$ to $root(\phi(v))$. Following this definition, we attempt to map the nodes one at a time in some pre-defined priority order, which will be detailed in Section 5.2.

Note that a node $v \in H$ maps to a tree subgraph $\phi(v) \subseteq G$. Clearly, there exist many such possible valid mappings. However, the *min_map()* function in Algorithm 1 returns a set Γ of minimal valid mappings $\phi(v)$. We hope to succeed with this minimal mappings. Subsequently, in the future, if a direct successor of v cannot be mapped, then more elaborate tree subgraph mappings of v are explored through *expand_map()* function.

Each feasible mapping ensures that all the minor edges from mapped direct predecessors and successors of v can be mapped. In particular, while mapping node v, we identify all its mapped direct predecessors P and successors S. We ensure that minor edge constraint can be satisfied between each node $p \in P$ and v as well as between v and each node $s \in S$. In particular, if node v does not have any mapped successor (i.e., $S = \perp$), then we generate $\phi(v)$ containing a single node, i.e., the feasible node mappings with minimal resource requirement. If node v has mapped successors, then we attempt to generate $\phi(v)$ containing additional routing nodes to ensure that each mapped successor $s \in S$ can be reached from some node in $\phi(v)$. In other words, edge mapping is automatically performed through minor edge constraint checking and we do not need to explicitly map the edges.

In addition, we check for timing constraint between v and its predecessors/successors, the attribute constraint, and the register constraint. Finally, we also apply aggressive pruning constraints to eliminate mappings that are guaranteed to fail in the future. We may succeed in mapping all direct successors of v with this minimal mapping. If not, then we expand $\phi(v)$ by

Algorithm 1: Graph Minor Mapping Algorithm

```
begin
    order_list := DFG_node_ordering(H);
    II := max(resMII, recMII);
    while do
        /*Create MRRG with II*/;
        $G_{II}$ := Create_MRRG(G, II); $M$ := $\perp$;
        for all $v \in V(H)$ and $e \in E(H)$ do
            $\phi(v) := \perp;\ \phi(e) := \perp;$
        add all $\phi(v)$, $\phi(e)$ to $M$;  /* empty mapping */
        if Minor(H, $G_{II}$, M) then
            return(M);
        II++;
Function Minor(H,G,M)
begin
    if no unmapped node in $H$ then
        return(success);
    $v$ := next unmapped node in $H$ according to order_list;
    $P$ := $\{p\ |p \in pred(v) \wedge \phi(p) \neq \perp\}$;  /*mapped predecessors of $v$ */
    $S$ := $\{s\ |s \in succ(v) \wedge \phi(s) \neq \perp\}$;  /*mapped successors of $v$ */
    /*All node mappings satisfying minor edge, timing, attribute, pruning constraints */
    $\Gamma$ := min_map($v, P, S$);
    for each $\phi(v) \in \Gamma$ do
        update $M$ with $\phi(v)$;
        if Minor($H, G, M$) then
            return(success);   /* mapping completed */
    if $\Gamma = \perp$ then
        /* No feasible node mapping; expand predecessors */
        for each possible expansion do
            expand_map(v,P,M);
            /* attempt mapping $v$ again */
            if Minor($H, G, M$) then
                return(success);
    /* No node mapping; backtrack to the predecessor */
    return(failure);
```

adding one node at a time . If we cannot map subsequent nodes, even after expansions of $\phi(v)$, then we backtrack and attempt a different minimal mapping from Γ.

The mapping process continues till we have either mapped all the DFG nodes (i.e., the DFG is a restricted minor of the MRRG) or we have discovered that no such mapping is possible (i.e., the DFG is not a restricted minor of the MRRG) and we have to increment the II value.

5.2 DFG node ordering

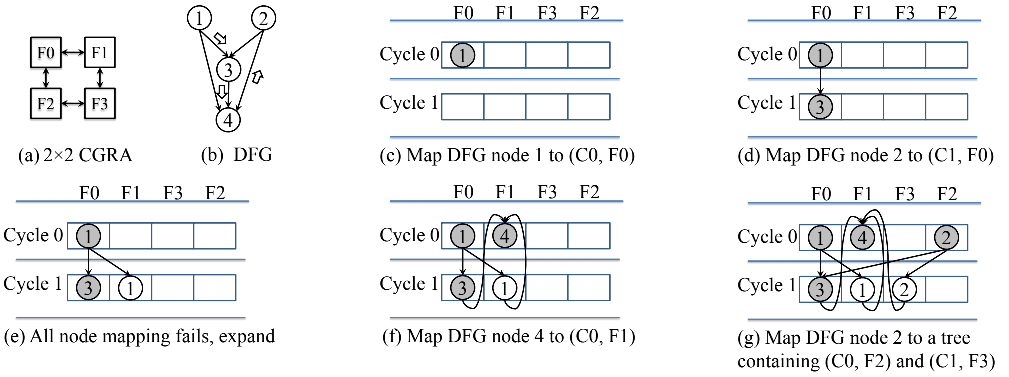
An appropriate ordering of the DFG nodes during mapping is crucial to quickly find a feasible solution. We impose the constraint that the nodes along the critical path have higher priority, i.e., they appear earlier. This is because if the critical path cannot be mapped with the current II value, then we can terminate the search process and move on to the next II value.

In addition, we employ an ordering that helps us validate the timing constraints as discussed in Section 4.3. A node *v* can be mapped only when at least one of its direct predecessor or successor has been mapped. That is *v* should appear in the ordering after at least one of its direct predecessor or successor nodes. The only exception is the first node in the ordering. The advantage of this ordering is that the timestamps $cycle(v)$ are generated appropriately for the nodes so that timing conflicts can be avoided early. When the DFG contains disjoint parts, a new timestamp is regenerated and propagated for every disjoint component during the mapping process. The relative ordering of the components is not important for our problem.

Figure 5(b) shows a DFG and the ordering of the nodes through the arrow signs. We start with the input node 1 on the critical path. We proceed along the critical path to node 3 and node 4. Notice that we could not include node 2 after node 1 because none of its direct predecessors or successors would have appeared in the ordering by then. After node 4, we include node 2 in the ordering.

5.3 Mapping Example

Suppose we have a DFG as shown in Figure 5(b) and we are attempting to map it to a 2×2 CGRA array. Let us assume that we are currently considering II=2. For simplicity of exposition, we only draw the occupied edges in the MRRG. The entire mapping process is illustrated in Figures 5(c-g).



(a) 2×2 CGRA (b) DFG (c) Map DFG node 1 to (C0, F0) (d) Map DFG node 2 to (C1, F0)

(e) All node mapping fails, expand (f) Map DFG node 4 to (C0, F1) (g) Map DFG node 2 to a tree containing (C0, F2) and (C1, F3)

Figure 5: An example of mapping process during the restricted graph minor test.

The process starts with mapping node 1. Node 1 is the initial node and it has no mapped direct successor. So the first tree subgraph generated by *min_map()* function contains only one node as shown in Fig 5(c): F1 in cycle 0 denoted as (C0, F0). Then we pick the next node in the priority list which is node 3. Again, this node has no mapped direct successors; so its tree mapping also contain only one node. However, we need to make sure that $\phi(1)$ is directly connected with $root(\phi(3))$ according to the edge constraint imposed by the edge e = (1, 3) in DFG. Mapping node 3 to (C1, F0), as shown in Figure 5(d), can satisfy the constraint.

The next node in the priority list to be mapped is node 4. However, this time we fail to find any feasible node directly connected to the mapped direct predecessors $\phi(1)$ and $\phi(3)$. As mapping for node 4 fails, we expand its predecessor's mapping. An extra node (C1, F1) is added to $\phi(1)$ in Figure 5(e). Notice that to distinguish between root nodes and other nodes, the root nodes have been shadowed. Now node 4 can be mapped to (C0, F1) in Figure 5(f).

The final node in the list is node 2. This time, node 2 has two mapped successors, node 3 and node 4. Thus, we find a tree subgraph $\phi(2)$ containing (C0, F2) and (C1, F3) (see Figure 5(g)) that satisfy both the minor edge constraint (direct links to root nodes of $\phi(3)$ and $\phi(4)$) and the timing constraints at node 3 and 4. As all the nodes and the minor edges have been mapped successfully, DFG is a minor of $MRRG$ with $II = 2$.

5.4 Pruning constraints

Pruning constraints are important to reduce the compilation time. Pruning constraints look ahead and quickly identify if the current mapping can be extended to a successful final mapping. This lookahead helps to eliminate mappings that are guaranteed to fail in the future. Note that the pruning constraints do not affect the optimality of the solution.

Available resource constraint This constraint simply checks that the number of available FUs of each type in the MRRG is larger than or equal to the number of unmapped DFG nodes of the same type. For example, the number of remaining available memory FUs must be larger or at least equal to the number of unmapped memory operations in the DFG. Such global variables are used to record information about the available FUs and the unmapped DFG nodes and are updated every time the partial mapping changes. Thus both time and space complexity of this constraint are *O(1)*.

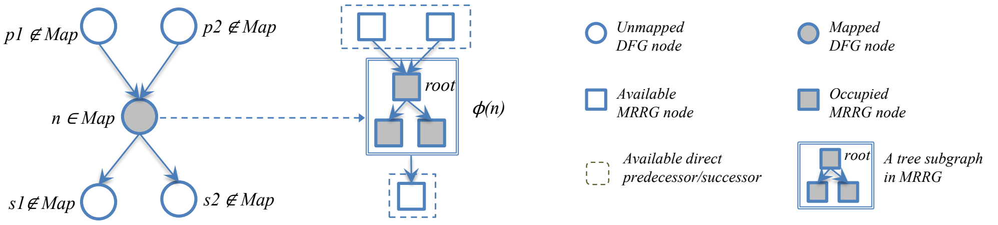


Figure 6: Illustrations of degree pruning constraint.

Degree constraint This constraint considers the local structures between the DFG H and the MRRG G. Let $\phi(n) \subseteq G$ be the tree subgraph representing the mapping of node $n \in V(H)$. The number of unmapped direct predecessors of n in the DFG must be smaller than or equal to the number of available direct predecessors of $root(\phi(n))$ in the MRRG.

On the other hand, if n has any unmapped direct successor, then the number of available direct successors of $\phi(n)$ must be at least one. This is because the data from $\phi(n)$ can be routed through any available outgoing node. For example in Figure 6, DFG node n is mapped to $\phi(n)$ in the MRRG. It has two unmapped direct predecessors and two unmapped direct successors. So $root(\phi(n))$ must have at least two available direct predecessors and there must be at least one available direct successor of $\phi(n)$ in the MRRG. Notice that the available direct successors of $\phi(n)$ are those available MRRG nodes directly connected from any node in $\phi(n)$.

The degree pruning constraint checks for all the DFG nodes in the current mapping. Since the degree of the nodes in both graphs are limited, the time complexity for this pruning constraint is $O(cN)$, where N is the number of DFG nodes and c is the average number of producer nodes in $\phi(n)$ across all mapped DFG nodes n.

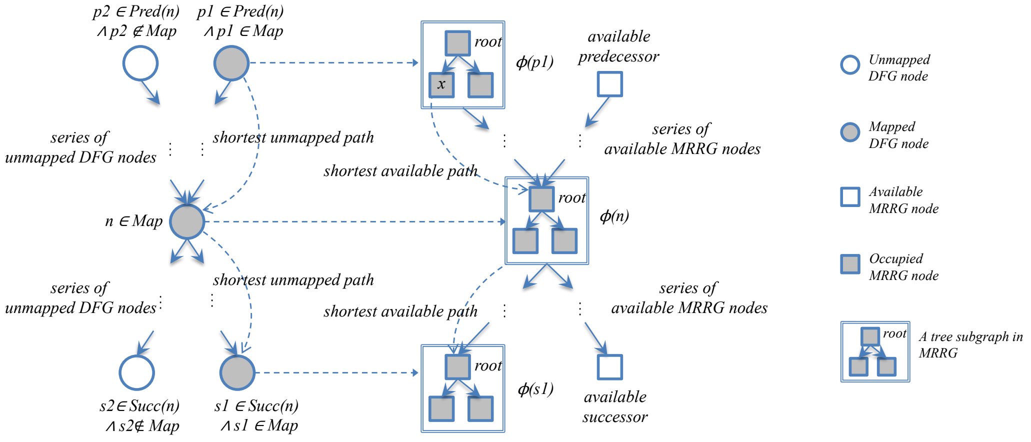


Figure 7: Illustration of predecessor and successor constraints.

Predecessor and successor constraint We further exploit structural patterns formed by each mapped DFG node n and its predecessors/successors as shown in Figure 7. Basically, we try to check the timing constraint inherently imposed by these patterns. During the mapping, we first calculate the shortest path lengths in both DFG and MRRG. The shortest paths defined here only consists of unmapped DFG nodes or available MRRG nodes except the two end nodes. For any mapped predecessor p of n, if p and n are connected through the shortest unmapped path $r = (p \rightsquigarrow n)$, then $\phi(p)$ and $\phi(n)$ should also be connected by a shortest available path $R = (x \rightsquigarrow root(\phi(n)))$, $x \in \phi(p)$, in MRRG. And we have

$$cycle(x) - cycle(root\phi(n)) \geq max(length(R), length(r))$$

which uses the fact that the timestamp differences must be at least equal to the length of the shortest path connecting the corresponding nodes either in the MRRG or in the DFG. Similar constraints are also applied to the patterns formed by n and its successors.

We also consider the relationships between a mapped DFG node n and its unmapped predecessors/successors. However, these predecessors/successors have not been mapped yet, there is no explicit structural information to be used for pruning purpose. Instead, we calculate the number of available MRRG nodes those could be connected to $root(\phi(n))$ (or reached from $\phi(n)$) through available MRRG paths. The number must be at least equal to the number of unmapped predecessors (or successors) of n, which can be connected to (or from) n through unmapped DFG paths.

To obtain the reachability information in both the DFG and the MRRG during the mapping, two reachability matrices are built using an efficient algorithm by Italiano et al. [24]. The algorithm has a time complexity $O(K)$ with $O(K^2)$ space overhead, where K is the number of nodes in the input graph. Each element (u, v) in one matrix is represented as the shortest path length between the node u and node v. To build the reachability matrix for M MRRG nodes, the time complexity is $O(M^2)$. As the computation for reachability matrices is the most time consuming step, the overall time complexity for the pruning constraint is $O(M^2)$.

Feasibility constraint In the final pruning constraint, we exploit the structural patterns specified by the unmapped DFG nodes. As shown in Figure 8, for each unmapped DFG node, we find all its mapped predecessors and successors reachable through unmapped paths. There must be at least one MRRG node that has the same connectivity to all the subgraphs the corresponding predecessors and successors have been mapped to. More specifically, let n is such an unmapped

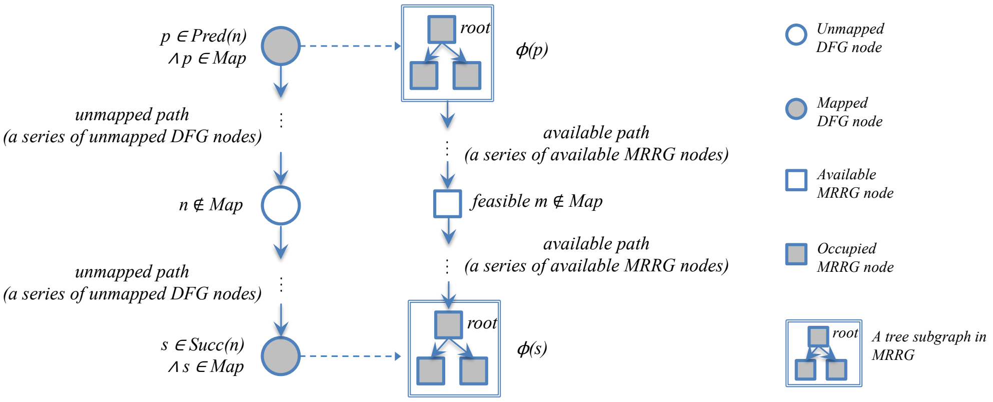


Figure 8: Illustration of feasibility constraint.

DFG node, p is a mapped predecessor of n ($p \in pred(n)$) and p is connected to n through an unmapped path. Then in the MRRG, there must be at least one available MRRG node m such that m could be connected from $\phi(p)$ through an available path. As this pruning constraint also depends on the reachability matrices, the complexity is $O(M^2)$.

5.5 Acceleration strategies

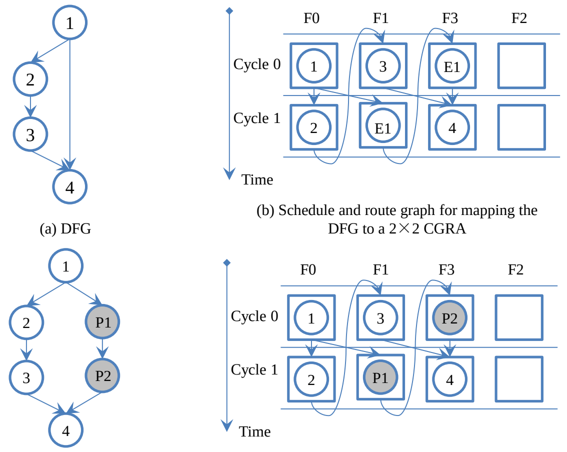
We now introduce additional strategies to further accelerate compilation time. These strategies are integrated in the preprocessing step and the constraints integrated in the algorithm infrastructure. All the strategies are designed in such a way that they do not impact the optimality of the mapping; but only improves the compilation time.

5.5.1 Dummy nodes in the DFG

The first acceleration strategy is to introduce dummy nodes in the DFG during the preprocessing step. These dummy nodes are *only* used for routing, which means they can be mapped to non-computation nodes in the MRRG, e.g., register file nodes. Basically, the idea is based on the observation that expanding the tree mapping $\phi(v)$ for any node v is quite expensive. This is because $\phi(v)$ is expanded only after all attempts to map subsequent nodes have failed. Also the expansion is carried out incrementally, i.e., $\phi(v)$ is expanded by only one node at a time. While this ensures good scheduling quality, the compilation time can potentially increase. The goal of introducing dummy nodes is to avoid the expansions as much as possible without affecting the quality of the solution.

Motivating example Figure 9 shows an example of how dummy nodes can avoid expansion of node mapping. Suppose we are trying to map the DFG shown in Figure 9(a) to a 2×2 CGRA. The mapping order is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. The first three nodes 1, 2, and 3 can be mapped successfully in the first attempt. However, when we try to map node 4, the mapping attempt fails (Γ is empty in this case) and we have to expand $\phi(1)$ twice in order to find the final feasible mapping for node 4. The final schedule and route graph is shown in Figure 9(b) with the expansion nodes for $\phi(1)$ denoted as $E1$. The detailed search process is also listed in Figure 9(c).

To avoid the mapping failures and expansions, we can add two dummy nodes $P1$ and $P2$, as shown in Figure 9(d). Suppose the mapping order for the new DFG is $1 \rightarrow 2 \rightarrow 3 \rightarrow P1 \rightarrow P2 \rightarrow 4$. After mapping the three nodes 1, 2 and 3, we will continue to map $P1$ and $P2$ without any failure and expansion. Finally, node 4 will be mapped successfully at the first attempt. The



(a) DFG

(b) Schedule and route graph for mapping the DFG to a 2×2 CGRA

(d) DFG with padding

(e) Schedule and route graph for mapping the padded DFG to a 2×2 CGRA

Step	Action	Progress
1	Map 1 to (C0, F0).	Mapping succeed.
2	Map 2 to (C1, F0).	Mapping succeed.
3	Map 3 to (C0, F1).	Mapping succeed.
4	Expand ϕ(1) to (C1, F1).	Mapping fails; expansion is carried out.
5	Expand ϕ(1) to (C0, F3).	Mapping fails; expansion is carried out.
6	Map 4 to (C1, F3).	Mapping succeed.

(c) Detailed information of the mapping process

Step	Action	Progress
1	Map 1 to (C0, F0).	Mapping succeed.
2	Map 2 to (C1, F0).	Mapping succeed.
3	Map 3 to (C0, F1).	Mapping succeed.
4	Map P1 to (C1, F1).	Mapping succeed.
5	Map P2 to (C0, F3).	Mapping succeed.
6	Map 4 to (C1, F3).	Mapping succeed.

(f) Detailed information of the mapping process for the padded DFG

Figure 9: A motivating example for dummy node insertion.

final schedule and route graph is shown in Figure 9(e) and the detailed mapping process is listed in Figure 9(f).

Clearly, dummy node insertion is useful in guiding the mapping process. So we add dummy nodes as part of DFG pre-processing step. We first assign scheduling levels to each DFG node using as soon as possible (*ASAP*) scheduling policy and as late as possible (*ALAP*) scheduling policy. The number of dummy nodes inserted to a DFG edge $e = (u, v) \in E(H)$ is equal to the difference between the *ASAP* level of v and the *ALAP* level of u. This is somewhat similar in concept to node balancing in [20]. However, the difference is that we insert dummy nodes to accelerate the search process to obtain a feasible schedule. In the previous approach [20], adding more balancing nodes is a requirement to obtain a valid schedule.

5.5.2 Fast implementation of pruning constraints

For large DFGs, the pruning constraints can increase the compilation time. The most expensive part is the reachability matrices computation in the pruning constraints and the running time to generate a full reachability matrix for the MRRG is $O(M^2)$. Note that M is the number of nodes in the MRRG, which could increase to a large value when II increases contributing significantly to compilation time. To reduce this overhead, we bypass updating the reachability matrix of the MRRG at each step.

We do, however, generate the reachability information for the DFG statically in the beginning and for the MRRG at its generation step for each II value. We believe that the two static matrices provide limited but enough information for the pruning purposes. The static reachability matrices now record the reachability information between any two arbitrary nodes in the absence of any mapping, e.g., the element (x, y) in the MRRG matrix records the static shortest path length between MRRG nodes x and y. However, the pruning constraints require the reachability information to be updated every time a node mapping is carried out. With only statical reachability matrix, the pruning constraints have to be redesigned as follows.

Fast implementation of predecessor and successor constraints Unlike the original constraint, the fast implementation only focuses on the structural patterns related to current mapping as shown in Figure 10. Suppose the candidate DFG node n is mapping to $\phi(n)$ in the MRRG. For every mapped predecessor p of n, we can have the length value for the static shortest path $r_s = (p \rightsquigarrow n)$, from static DFG matrix. Let $R_S = (x \rightsquigarrow \phi(n), x \in \phi(p)$, be the static shortest path between $\phi(p)$ and $\phi(n)$ in MRRG. x can be identified by checking the static MRRG matrix

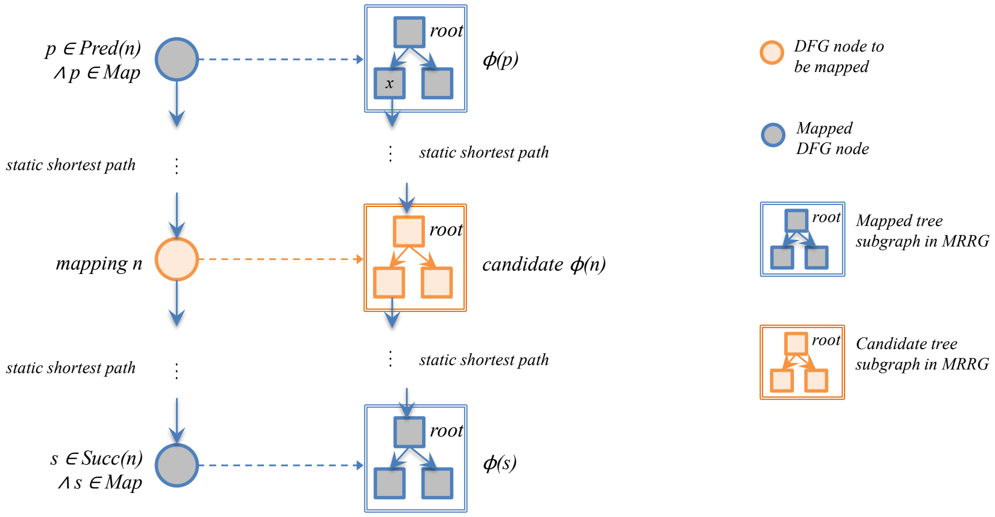


Figure 10: Illustration of fast implementation of predecessor and successor constraint

for all the nodes in $\phi(p)$. Utilizing the same fact used in the original constraint, we have

$$cycle(x) - cycle(root(\phi(n))) \geq max(length(R_S), length(r_s))$$

Similarly, constraints are also imposed for the structural patterns formed by the candidate node and its mapped successors. The fast implementation reduces the runtime complexity from $O(M^2)$ to $O(cN)$. c is the average number of nodes in $\phi(n)$ for each DFG node n.

Fast implementation of feasibility constraint The basic idea for designing fast implementation of feasibility constraint is to consider the local effects brought by consuming one MRRG node for the remaining unmapped DFG nodes. Suppose the candidate MRRG node to be used for mapping is m, then the consumption will affect the potential mappings those also require m. As shown in Figure 11, if m is directly linked from any node in $\phi(p)$, p is a mapped DFG node, then the consumption of m can affect the mapping for the unmapped child $child\_p$ of p. In other words, we need to ensure that apart from m there is another available MRRG node m' can be used to map $child\_p$ satisfying certain timing constraints. Regarding to every mapped successor s of $child\_p$, we can have the static shortest path $r_s = (child\_p \rightsquigarrow s)$. Let R_S be the static shortest path connecting m' and $root(\phi(s))$, $R_S = (m' \rightsquigarrow root(\phi(s)))$. Following the same fact used before, we have

$$cycle(m') - cycle(root(\phi(s))) \geq max(length(R_S), length(r_s))$$

If m is also a direct predecessor of the root node of $\phi(s')$, s' is a mapped DFG node. Similar checking constraints are also used considering the unmapped parent node of s'. The time complexity of this pruning constraint is also $O(cN)$.

5.6 Integration of Heuristics

Our modulo scheduling algorithm (Algorithm 1) can achieve the optimal II by definition. This is because it checks if the DFG is a minor of the MRRG for each value of II, starting with the minimum possible value. However, even with the pruning and acceleration strategies, the runtime of the optimal algorithm can be prohibitive when both the number of DFG nodes and the number of CGRA functional units are quite large. Therefore, we integrate some heuristics in the algorithm infrastructure to speed up the search process. This may introduce sub-optimality,

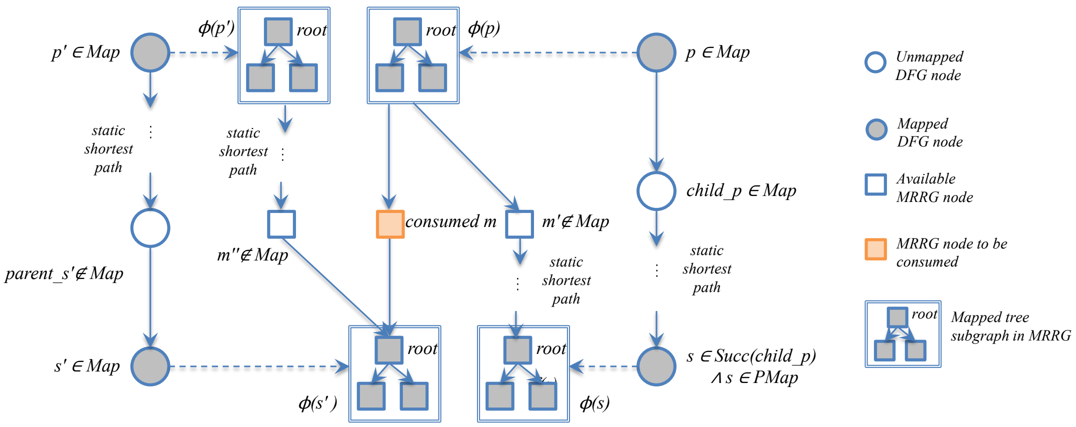


Figure 11: Illustration of fast implementation of feasibility constraint

i.e., the search process may miss a valid mapping at lower II value even though it exists. But the compilation time improves significantly.

The first heuristic avoids backtracking between two unrelated nodes. In the optimal search process, if a node m cannot be mapped, then we backtrack to the node n which appears just before m in the DFG node ordering. However, node n may not be a predecessor or successor of node m in the DFG and hence may not be able to steer the search towards a successful mapping to m. Instead, we directly backtrack to the last predecessor or successor of node m in the ordering.

The second heuristic is motivated by the edge-centric mapping for CGRAs [35]. During graph minor testing, instead of enumerating all possible tree subgraphs for mapping node n, the procedure simply aims to find limited number of feasible subgraphs. The feasible subgraphs are chosen to be those with minimal number of nodes. After all the specified subgraphs have been explored, the node mapping fails.

The final heuristic makes it possible to escape from extensive subgraph expansions. We put a counter for each node mapping. The counter is increased every time an expansion is carried out. Once the counter reaches a pre-defined threshold value, we eliminate current mapping and backtrack to previous mappings. Our experimental evaluation reveals that this is the only heuristic that sometimes prevent us from reaching a feasible solution even if one exists.

6 Experimental Evaluation

We now proceed to evaluate the quality and the efficiency of our mapping algorithm. We initially target a mesh-like 4×4 CGRA architecture. The 4×4 array is the basic structure in many CGRA architectures and has been widely used to evaluate various mapping algorithms [34, 35, 26, 22, 25, 6]. The functional units in the array can be heterogeneous or homogeneous and each of them is connected to its immediate neighbors. We assume each functional unit is comprehensive and is capable of handling any operation. Later, we evaluate the versatility of graph minor mapping approach in supporting diverse CGRA architectures such as heterogenous functional units and various register file configurations. We also evaluate the scalability issue by mapping to 8×8, 8×16 and 16×16 CGRAs.

We select a set of loop kernels from multimedia applications, MiBench [19] and SPEC2006 [23]. The DFGs for these kernels are generated from Trimaran [8] back-end using Elcor intermediate representation [1].

Comparison with Prior Art There exist a number of approaches to CGRA mapping as presented in Section 2. We compare our graph minor approach (abbreviated as G-Minor here) with two previous techniques: simulated annealing based approaches and EPIMap [20]. Simulated annealing (SA) based approaches [32] are widely considered to provide high-quality mapping solutions with (possibly) longer compilation time. EMS, the edge-centric mapping approach [35], provides significantly reduced compilation time with some degradation in the quality of the schedule. As mentioned in Section 1, in parallel to G-Minor approach, [20] have proposed graph epimorphism based mapping approach EPIMap that produces better quality solutions than EMS with similar compilation time. We compare G-Minor with EPIMap as it represents state-of-the-art CGRA mapping approach.

For this comparison, we have re-implemented the EPIMap approach [20] and the simulated annealing (SA) algorithm [32]. All the evaluations have been carried out for 4×4 mesh-like CGRA with comprehensive functional units similar to the setup in [20]. Our implementation of all the approaches allow route sharing. The compilation time is reported on a Intel Quad-Core processor running at 2.83GHz with 3GB memory.

Figure 12 compares the scheduling quality for 18 benchmarks. The Y-axis represents the achieved II value. The first bar represents the minimal II value achievable for each kernel on the 4×4 CGRA. The remaining bars from left to right represent the II achieved for G-Minor, EPIMap, and simulated annealing (SA), respectively.

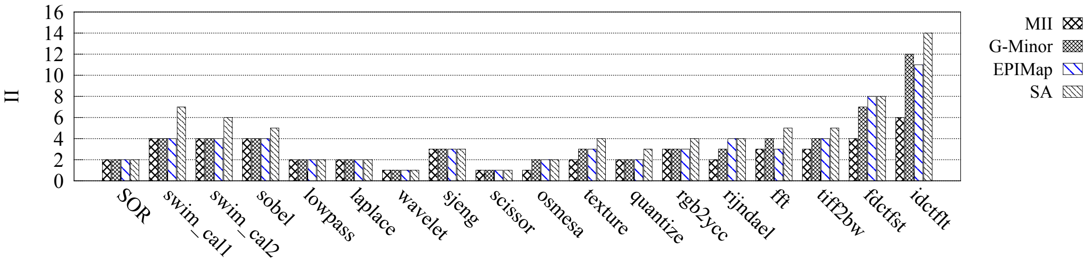


Figure 12: Scheduling quality for G-Minor, EPIMap and simulated annealing.

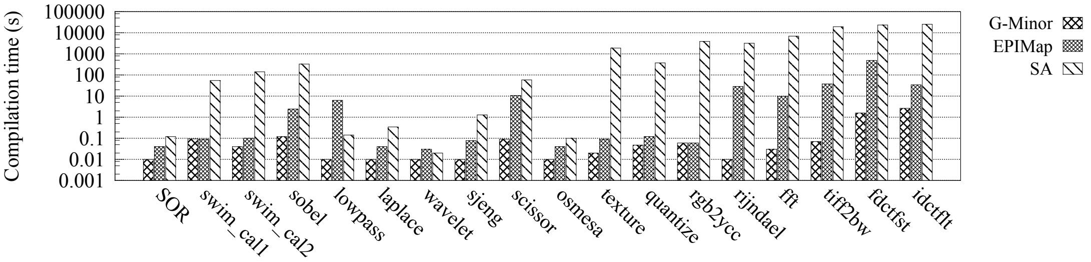


Figure 13: Compilation time for G-Minor, EPIMap and simulated annealing

We first observe that the scheduling quality generated by EPIMap and G-Minor are quite similar. The achieved II value is different between the two for only 4 out of 18 benchmarks. For example, G-Minor produces better scheduling results for *rijndael* and *fdctfst*, while EPIMap performs better for *fft* and *idctflt*. Even for these benchmarks, the difference is only one cycle. The two reasons for the competitive results between G-Minor and EPIMap are the following. G-Minor exhaustively searches for minor with all routing possibilities, while EPIMap restricts the number of routing nodes. On the other hand, EPIMap provides extra choices for mapping the DFGs such as replication (or re-computation) for high fan-out nodes. An interesting possible future research direction would be to combine the relative strengths of G-Minor and EPIMap.

We also observe that for a large subset of benchmarks (11 out of 18), both G-Minor and

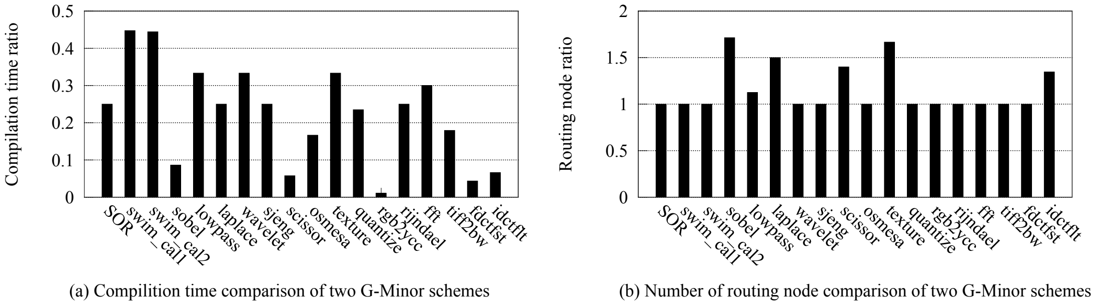
EPIMap achieve Minimal II (MII). This is consistent with the results reported in [20]. SA, on the other hand, achieves minimal II value for 6 benchmarks. In general, G-Minor and EPIMap provide better quality schedule compared to simulated annealing approach.

The runtime of the three approaches for all the benchmarks are shown in Figure13. It is well known that SA approaches require longer compilation time [35] specially for large kernels. Similar compilation time has been reported in [21].

Clearly, both G-Minor and EPIMap reduce compilation time significantly using more guided approach to mapping. Across all the benchmarks, the average compilation time for EPIMap is 34.26 sec, which is consistent with the timing reported in [20]. G-Minor, on the other hand, provides extremely fast compilation speed. On an average, it only requires 0.27 sec. There are two reasons for the fast compilation time of G-Minor compared to EPIMap. First, the graph minor test algorithm in G-Minor has been highly optimized using various pruning constraints and different acceleration strategies. EPIMap transforms the DFG and uses it as an input to the off-the-shelf maximal common subgraph (MCS) kernel [29]. Thus the compilation time of EPIMap is dependent on the MCS kernel implementation; a faster and more optimized MCS kernel would lead to better compilation time. Also, once the mapping fails in the MCS kernel, the DFG is re-transformed according to the specified heuristics and input into the MCS kernel again. Thus, EPIMap might need to transform the DFG and repeat the MCS kernel computation multiple times potentially leading to longer compilation time.

Impact of acceleration strategies We evaluate reduction in compilation time brought about using the acceleration strategies presented in Section 5.5. We compare compilation time for two different versions of G-Minor: the slow mode and the fast mode in Figure 14. The fast mode uses the acceleration strategies we illustrated in Section 5.5. Both modes achieve identical II for all the benchmarks in our experimental evaluation. This is because the acceleration strategies are designed such that they do not impact the quality of the solutions, but provides better guidance for the search process.

In Figure 14(a), the compilation time of the fast mode is normalized w.r.t. the slow mode. The fast mode can effectively reduce the compilation time by more than 50% for all the benchmarks. The penalty for the fast mode is in the form of using more routing nodes. Figure 14(b) compares the number of routing nodes for the two schemes. The average ratio is around 1.15, which means there are 15% extra routing nodes used in fast mode. This is because the fast implementations for pruning constraints using static shortest path connectivity information can lead to more node expansions .

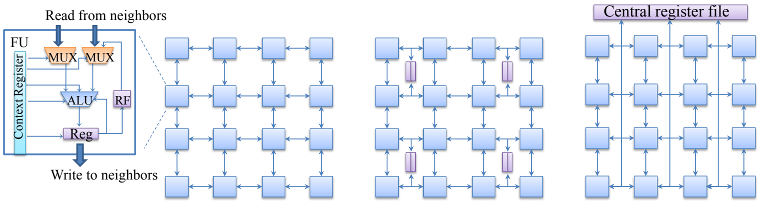


(a) Compilition time comparison of two G-Minor schemes

(b) Number of routing node comparison of two G-Minor schemes

Figure 14: Experimental results for fast G-Minor scheme (with acceleration strategies) comparing to slow G-Minor scheme.

Different CGRA configurations Our mapping approach can support diverse CGRA architectures through parameterization. We can fully exploit the structural information of the CGRAs for explicit data routing. Our register file modeling approach can also support many different register file configurations. We evaluate three different register file configurations denoted as NORF (architecture with no RF shown in Figure 15(a)), LRF (architecture with local shared RF shown in Figure 15(b)) and CRF (the architecture with central shared RF shown in Figure 15(c)). Heterogeneities for functional units are also supported in our framework. We use *Homo* to denote homogeneous functional units and *MxC* denoting the availability of x columns of memory units in the array. So an architectural configuration MxC-LRF-yR corresponds to an array with x columns of memory units and a locally shared register file with y registers. Each register file is associated with two read ports and one write port. The experiment results for 4×4 CGRAs with different number of memory units and different register file configurations are shown in Figure 16.



(a) No shared RF (b) Local RF (c) Central RF

Figure 15: 4×4 CGRAs with different register file configurations

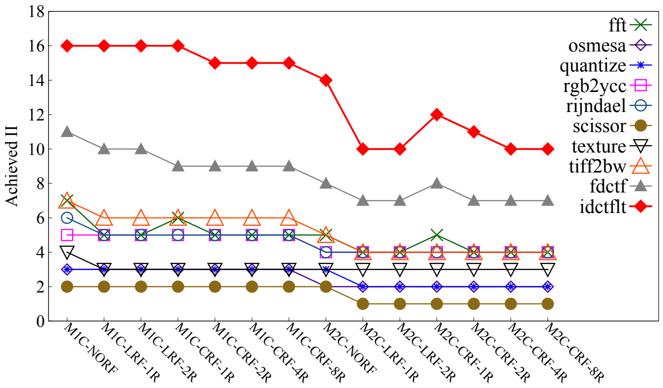


Figure 16: Achieved II for different CGRA configurations.

The experiment results indicate that memory units is the most critical resource. Adding more memory units brings substantial benefit by reducing the achieved II. The most interesting point to note is that adding more registers may not necessarily improve II. This is because the intelligent exploration of the search space can find mappings within limited routing resources. Adding more routing resources such as increasing the size of local/global register files can reduce the mapping efforts but could also end up with resource wastage. We notice that starting from M2C-LRF-1R configuration, increasing the number of registers and providing more connectivity through registers for routing do not manage to achieve reduced II.

Scalability Our G-Minor fast mode can dramatically accelerate the compilation time. This highlights the potential for G-Minor approach to target CGRAs with large sizes. We test the scalability by configuring the size of NORF CGRA to 8×8, 8×16 and 16×16 2D-mesh. The average compilation time for G-Minor with different CGRA sizes is shown in Table 1. The results confirm that G-Minor approach can easily scale to map kernels on large CGRAs.

Table 1: Compilation time for CGRAs with different sizes

	4×4 CGRA	8×8 CGRA	8×16 CGRA	16×16 CGRA
Avg. compilation time (s)	0.27	1.91	4.33	8.40

7 Conclusions

We present a comprehensive technique for application mapping on CGRAs. We formalize the CGRA mapping problem as restricted graph minor containment of the data flow graph representing the computation kernel in the modulo routing resource graph representing the CGRA architecture. We design a customized and efficient graph minor search algorithm for our problem that employs aggressive pruning and acceleration strategies. We conduct extensive experimental evaluation of our approach and show that it achieves quality schedule with minimal compilation time.

References

[1] Shail Aditya, Vinod Kathail, and B Ramakrishna Rau. *Elcor's machine description system: Version 3.0.* Hewlett Packard Laboratories, 1998.

[2] Isolde Adler, Frederic Dorn, Fedor V Fomin, Ignasi Sau, and Dimitrios M Thilikos. Fast minor testing in planar graphs. *Algorithmica*, 64(1):69–84, 2012.

[3] Mythri Alle, Keshavan Varadarajan, Reddy C Ramesh, Joseph Nimmy, Alexander Fell, Adarsha Rao, SK Nandy, and Ranjani Narayan. Synthesis of application accelerators on runtime reconfigurable hardware. In *Proceedings of the 2008 international conference on Application-Specific Systems, Architectures and Processors*, ASAP'08, pages 13–18. IEEE, 2008.

[4] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. Interconnect-aware mapping of applications to coarse-grain reconfigurable architectures. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application*, pages 891–899. Springer, 2004.

[5] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *Proceedings of the 2004 Conference on Design, Automation and Test in Europe*, DATE'04, pages 474–479. IEEE, 2004.

[6] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations. In *the 2003 Workshop on Application Specific Processors, held in conjunction with the International Symposium on Microarchitecture (MICRO)*, 2003.

[7] Janina A Brenner, Sándor P Fekete, and Jan C van der Veen. A minimization version of a directed subgraph homeomorphism problem. *Mathematical Methods of Operations Research*, 69(2):281–296, 2009.

[8] Lakshmi N Chakrapani, John Gyllenhaal, W Hwu Wen-mei, Scott A Mahlke, Krishna V Palem, and Rodric M Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In *Languages and Compilers for High Performance Computing*, pages 32–41. Springer, 2005.

[9] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. In *Proceedings of the 2012 International Conference on Field-Programmable Technology*, ICFPT'12, pages 285–292. IEEE, 2012.

[10] Nathan Clark, Amir Hormati, Scott Mahlke, and Sami Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES'06, pages 147–157. ACM, 2006.

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[12] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded System*, LCTES'08, pages 151–160. ACM, 2008.

[13] Giuseppe Di Battista, Maurizio Patrignani, and Francesco Vargiu. A split&push approach to 3d orthogonal drawing. In *Graph Drawing*, pages 87–101. Springer, 1998.

[14] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceedings of the 1995 International Federation for Information Processing Working Group*, pages 264–267, 1995.

[15] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[16] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA'09, pages 191–200. ACM, 2009.

[17] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *Proceedings of the 2010 International Conference on Soft Computing and Pattern Recognition*, SoCPaR'10, pages 145–151. IEEE, 2010.

[18] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan. Analysis of inner-loop mapping onto coarse-grained reconfigurable architectures using hybrid particle swarm optimization. *International Journal of Organizational and Collective Intelligence*, 2(2):17–35, 2011.

[19] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *the 2001 IEEE International Workshop on Workload Characterization*, pages 3–14. IEEE, 2001.

[20] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12, pages 1284–1291. ACM, 2012.

[21] 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*, DAC'13, pages 18:1–18:10. ACM, 2013.

[22] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium*, IPDPS'07, pages 1–8. IEEE, 2007.

[23] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[24] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(2-3):273–281, 1986.

[25] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W Yoon, Doosan Cho, and Yunheung Paek. High throughput data mapping for coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1599–1609, 2011.

[26] Zion Kwok and Steven JE Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *Proceedings of the 13th International Symposium on Field-Programmable Custom Computing Machines*, FCCM'05, pages 35–44. IEEE, 2005.

[27] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Proceedings of the 2009 International Conference on Field-Programmable Technology*, ICFPT'09, pages 376–379. IEEE, 2009.

[28] Jong-eun Lee, Kiyoung Choi, and Nikil D Dutt. Compilation approach for coarse-grained reconfigurable architectures. *Design & Test of Computers, IEEE*, 20(1):26–33, 2003.

[29] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 1973.

[30] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 7th ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA'99, pages 135–143. ACM, 1999.

[31] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the 13th International Conference on Field Programmable Logic and Application*, FPL'03, pages 61–70. Springer, 2003.

[32] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the 2003 Conference on Design, Automation and Test in Europe*, DATE'03, pages 296–301. IEEE, 2003.

[33] Nils J Nilsson. *Principles of artificial intelligence*. Springer, 1982.

[34] 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*, CASES'06, pages 136–146. ACM, 2006.

[35] 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*, PACT'08, pages 166–176. ACM, 2008.

[36] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, MICRO'94, pages 63–74. ACM, 1994.

[37] Neil Robertson and Paul D Seymour. Graph minors. *Journal of Combinatorial Theory Series B*, 77(1):162–210, 1999.

[38] Neil Robertson and Paul D Seymour. Graph minors. XX. Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[39] 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.

[40] M Tuhin and Theodore S Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *Proceedings of the 2008 Canadian Conference on Electrical and Computer Engineering*, CCECE'08, pages 1723–1728. IEEE, 2008.

[41] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[42] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, and Yunheung Paek. SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASPDAC'08, pages 776–782. IEEE, 2008.

[43] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, and Yunheung Paek. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration Systems*, 17(11):1565–1578, 2009.

[44] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. MIRS: Modulo scheduling with integrated register spilling. In *the 2003 Workshop on Languages and Compilers for Parallel Computing*, LCPC'03, pages 239–253. Springer, 2003.