Graph Minor Approach for Application Mapping on CGRAs

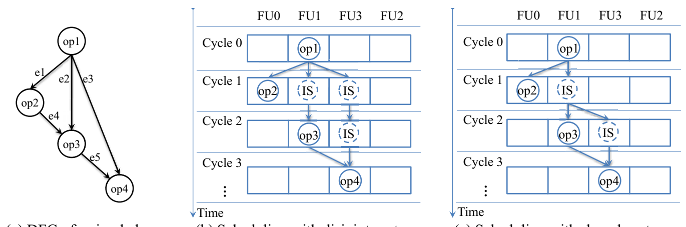
Liang Chen, Tulika Mitra
School of Computing
National University of Singapore
{chenliang,tulika}@comp.nus.edu.sg

***Abstract*—Coarse-grained reconfigurable arrays (CGRA) exhibit high performance, improved flexibility, low cost, and power efficiency for various application domains. Compute-intensive loop kernels are mapped to CGRA through modified modulo scheduling algorithms that integrate placement and routing. Most existing approaches are heavily influenced by VLIW compilation and FPGA synthesis techniques. A salient feature of these approaches is that data routing from a single source node to multiple destination nodes follow independent paths leading to resource wastage and hence inefficient schedule. We transform the CGRA mapping problem with route sharing into a graph minor problem. Our graph minor formalization provides a solid foundation for application mapping on CGRA. We provide an efficient framework based on graph mapping to solve this problem. Experimental validation shows that our approach leads to higher performance compared to state-of-the-art solutions with better resource utilization and minimal compilation time.**

I. Introduction

Coarse-Grained Reconfigurable Arrays (CGRA) are a promising alternatives between ASIC and FPGA. 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 (e.g., smartphone) demand flexibility to support a diverse range of applications. FPGAs provide high flexibility, but suffer from low efficiency and high power consumption. To bridge this gap, CGRA architectures have been proposed such as ADRES [11], MorphoSys [18], and others. Most of these architectures arrange the function units (FUs) in a mesh-like structure with different register file (RF) configurations (Fig. 8).

The compute-intensive loop kernels are perfect candidates to be mapped to CGRA containing multiple functional units and dedicated or shared register files targeting high instruction-level parallelism. Software pipelining techniques, e.g., modulo scheduling, are thus introduced to map applications onto CGRA. Most of the CGRA mapping algorithms are inspired by mapping approaches for VLIW architectures and FPGA synthesis as CGRA share some similarities with both these architectures. For example, CGRA mapping algorithms adopt placement and routing techniques from FPGA synthesis process and inherit register allocation as a post-processing phase from VLIW compilation process. While these mechanisms have dominated the research in application mapping for CGRA, it is important to note that the inherent structure of the CGRA is very different from both FPGA and VLIW architecture. More concretely, the connectivity among the functional units in CGRA is usually fixed unlike FPGA where the interconnection can be reconfigured. Thus the mapping algorithms based on FPGA place and route techniques find it challenging to identify feasible routing paths in fixed interconnect structure of CGRA. This is one of the principle reasons why popular mapping algorithms, such as DRESC [12] with its simulated annealing based routing, take long time to converge to a solution. Similarly, unlike VLIW architecture, the functional units in most CGRAs have limited and explicit connection to the register files. Thus it is not safe to assume sufficient register resources during the



(a) DFG of a simple loop (b) Scheduling with disjoint routes (c) Scheduling with shared routes

Fig. 1. Disjoint routes and shared routes.

scheduling phase and apply register allocation as a post-processing step. Instead, register allocation should be integrated in the early stage with scheduling (place and route) to achieve the optimal mapping.

Another drawback of most existing approaches is that they do not consider route sharing among different edges of the data flow graph (DFG). For example consider the DFG corresponding to a simple loop in Fig. 1 being mapped onto a CGRA with four nodes. The edges $e2$ and $e3$ have the same source node but different destination nodes, i.e., they are propagating the same data. But existing approaches route $e2$ and $e3$ independently as shown in Fig. 1(b). In this case, we require three intermediate storage (IS) nodes for routing. If we can exploit node sharing among the routes, the number of intermediate storage required for routing can be reduced to two as shown in Fig. 1(c). This reduced resource usage often lead to more efficient schedules. In fact, application mapping with route sharing across edges essentially defines a restricted version of the graph minor problem. This graph minor based formalization provides a solid foundation to efficiently solve the application mapping problem for CGRA.

In this paper, we provide a comprehensive solution to the application mapping problem for CGRA. We first model the CGRA architecture as an enhanced modulo routing resource graph (MRRG) with wrap-around edges and register file. We formalize the modulo scheduling problem with node sharing across routes as a restricted version of the graph minor problem between the DFG and the MRRG. We propose an efficient solution for the graph minor problem that fully exploits the structure of the DFG and the CGRA interconnects to effectively navigate and prune the mapping alternatives. Moreover, register allocation is integrated with scheduling and routing rather than being performed in isolation as a post-processing step. Our approach leads to high quality mapping both in terms of performance and resource usage with minimal compilation time.

II. Related work

Mapping a loop kernel to CGRA using modulo scheduling was first discussed in [12]. While this simulated annealing based approach provides high quality schedules, it has long runtime especially for large graphs. Most of the following work use node-centric techniques. In node-centric approaches [12], [14], [7], [5], [6], the operations (nodes) in the DFG are placed one at a time followed by routing. A later work, [15] observes that node-centric modulo scheduling is a poor match for CGRA. Their edge-centric approach shows

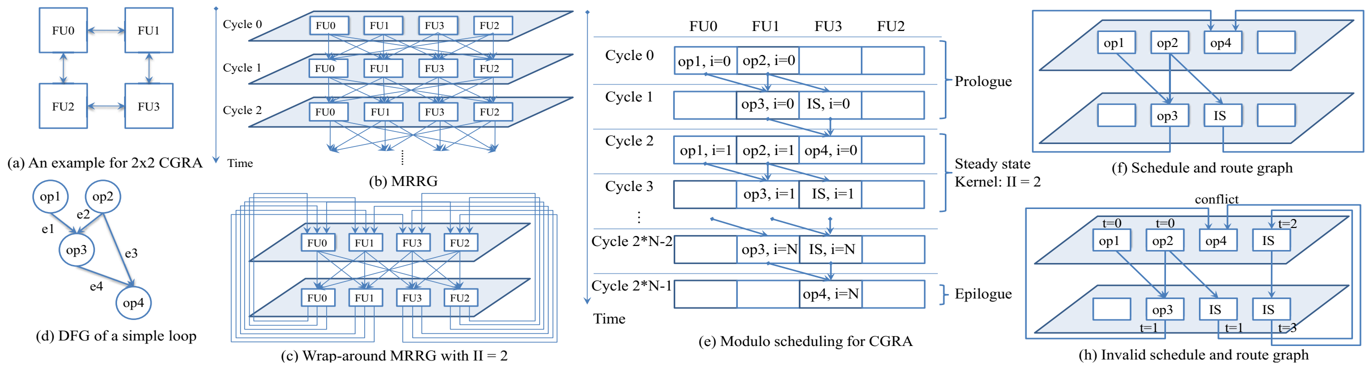


Fig. 2. Modeling of loop kernel mapping on CGRAs: An illustrative example.

significant benefit obtained from explicit routing of the edges in the DFG. However, none of these approaches attempt to share routes corresponding to different edges with the same source.

Most techniques also do not model explicit routing through the register files. While the register files consume significant amount of area and substantially impact the performance of CGRA [10], most current approaches implicitly assume the availability of sufficient number of registers and connectivity between the functional units and the registers. Thus the generated schedule may use a large number of registers with register allocation performed as a post-processing step. [19] performs explicit routing through register files. Their register rotation approach still uses a post-pass register allocator and assumes enough capacity in the register file during scheduling. In contrast, our approach integrates register allocation with scheduling through explicit modeling of the register files and their connectivity with the functional units. Our approach with route sharing in both registers and routing functional units often leads to better schedule and potentially reduces the register pressure compared to adapting VLIW compilation technologies such as register spilling [4] and register rotation [19].

III. Modulo Scheduling for CGRA

Given an input loop kernel from an application and a CGRA architecture, the goal of application mapping is to generate a schedule such that the application throughput is maximized. The loop kernel is represented as a data flow graph (DFG) where the nodes represent the operations and the edges represent the dependency among the operations. Fig. 2(d) shows the DFG corresponding to a simple loop. Fig. 2(a) 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. Modulo Scheduling

Modulo scheduling is a software pipelining technique used to exploit instruction-level-parallelism in the loops by overlapping consecutive iterations [16]. The schedule produced includes three phases: the prologue, the kernel and the epilogue. The length of the kernel, which is also the interval between successive iterations, is called the initiation interval (II). The kernel corresponds to the steady state execution of the loop and comprises of operations from consecutive iterations. 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 (resMII) and recurrence-minimal II (recMII), and attempts to find a feasible schedule with that II value. If the scheduling fails, then the scheduling process is repeated with an increased II. Fig. 2(e) shows the modulo-scheduled version of the loop in Fig. 2(d) to the CGRA architecture in Fig. 2(a) with prologue, kernel, and epilogue where II=2. Notice that *op4* from the i^{th} iteration is executing in the same cycle with *op1* and *op2* from the $(i{+}1)^{th}$ iteration in the steady state. Also, we need to hold the output of *op2* in an intermediate storage (IS) till it gets consumed by *op4*. 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 CGRA, the modulo scheduler has to be aware of the details of the underlying architecture, such as the interconnections among the FUs and the RFs, to route the data.

B. Modulo Routing Resource Graph (MRRG)

Mei et al. [12] defined a resource management graph for CGRA mapping, called Modulo Routing resource graph (MRRG), which has been used extensively in subsequent research [7], [14], [15], [5]. 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. In this paper, we adopt a simplified form of MRRG proposed in [14] where a node corresponds to a FU or a RF rather than the ports. Thus MRRG is a directed graph $G = (V, E, II)$ where II corresponds to the initiation interval. Each node $v \in V$ is a tuple (n, t), where n refers to the resource (FU or RF) and t is the cycle. Let $\{e = (u, v)\} \in E$ be an edge in the MRRG 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 in the MRRG, node $u = (m, t)$ is connected to node $v = (n, t{+}1)$ for $t \geq 0$.

For example, Fig. 2(b) shows the MRRG corresponding to the CGRA shown in Fig. 2(a). 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 $v'{=}(n, t{+}k{\times}II)$ (where $k > 0$) are also marked occupied. For example, in the modulo schedule with II=2 shown in Fig. 2(e), as FU1 is occupied by *op1* in cycle 0, it is also occupied by *op1* 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 [12].

The goal of CGRA modulo scheduler is to generate II different configurations for the CGRA where each configuration corresponds to a particular cycle. These configurations are stored in a configuration RAM and provide configuration context to the CGRA every cycle. The configuration context specify the functionalities of the FUs, connectivity among the FUs and the registers, etc. Usually, this is done by configuring the context registers every cycle. If the scheduling specifies routing explicitly, then a configuration also includes directives for each FU about where to get its input from the previous cycle and where to write its output for the next cycle. As these configurations are reloaded every II cycles, the output from the configuration in the last cycle are consumed by the configuration in the first cycle. 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 Fig. 2(c) (similar graph is also used in [5]). Now we are only interested in the configuration in the steady state, that is, the kernel. For example, the modulo scheduling kernel shown in Fig. 2(e) could now be simplified to the graph in Fig. 2(f). This simplified graph will be referred to *schedule and route graph* which captures all the necessary information for scheduling and routing 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 a clear view during the mapping process. *In the following, the term MRRG will be used to refer to MRRG with wrap around edges.*

C. Register File Modeling

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* [19]. However, different from [19], 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, which in essence helps to accelerate the mapping algorithm.

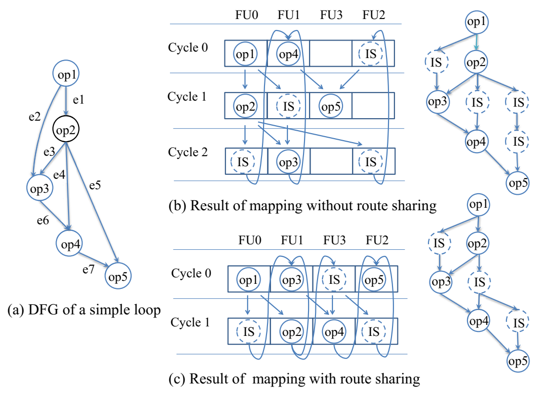
IV. Graph Minor Mapping

As mentioned in Section I, node sharing among the data routes can bring in substantial savings in resource usage. Consider an operation *v* in the DFG with multiple direct successors. In existing modulo scheduling techniques, the data from node *v* will be routed to the direct successors along different routes. However, this is wasteful as all the direct successors need the same data and hence can easily share the routes. We call the sharing of nodes among routes starting from the same source node, *route sharing*. We model the application mapping on CGRA with routes sharing as a **graph minor** problem [17] between the DFG and the MRRG. Finally, an efficient algorithm is provided to compute the optimal or near-optimal application mapping.

A. Advantages of Route Sharing

Route sharing can lead to lower initiation interval (II) while minimizing the resource usage. Route sharing reduces resource usage when data is routed through functional units and reduces register pressure when data needs to be stored in the register file temporarily. When resource budget and connectivity are limited, route sharing leads to lower II.

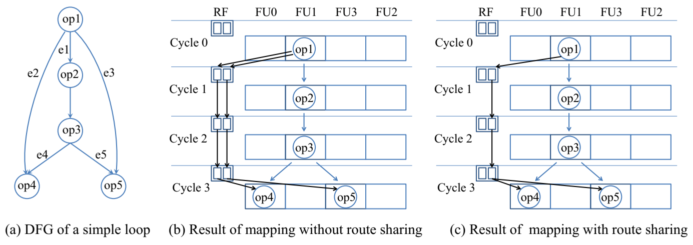
Fig. 3 presents an illustrative example. The DFG is being mapped on the 2x2 CGRA shown Fig. 2(a). The node *op2* in the DFG has three direct successor nodes: *op3*, *op4* and *op5*. Existing approaches route the data corresponding to each edge independently and required three intermediate storage (IS). Moreover, the route between *op1* and *op3* needs an additional IS, bringing the total number of IS required to four. As we have five nodes in the DFG, we now require $5+4=9$ FU slots, while the CGRA contains only four FU slots. Thus we need



(a) DFG of a simple loop
(b) Result of mapping without route sharing
(c) Result of mapping with route sharing

Fig. 3. Illustration of smaller II achieved by route sharing

II=3 because $\lceil 9/4 \rceil = 3$ and the modulo schedule is shown in Fig. 3(b). Once we enable route sharing, the edges *op2* $\rightarrow$ *op4* and *op2* $\rightarrow$ *op5* can share IS to achieve II=2 as shown in Fig. 3(c).



(a) DFG of a simple loop
(b) Result of mapping without route sharing
(c) Result of mapping with route sharing

Fig. 4. Illustration of reduced register pressure achieved by route sharing.

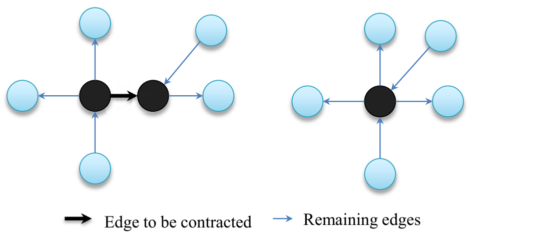
Fig. 4 presents an illustrative example of reduced register pressure due to route sharing. Let us assume that the 2x2 CGRA in Fig. 2(a) has a central register file with two registers and each FU is directly connected to the register file. We would like to map the DFG in Fig. 4(a). We model the RF as a single node with capacity constraint as discussed before. We further assume that the RF has higher priority to route data compare to FUs. In this DFG, *op1* has three direct successor nodes. In mapping without route sharing, edges *e2* and *e3* require different registers as shown in Fig. 4(b). However, with route sharing, we can use only one register that is shared by both edge *e2* and *e3* leading to much reduced register pressure.

B. Problem Formulation

We now present graph minor [17] based formulation of the application mapping problem on CGRA with route sharing. 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. [1] The definition of graph minor is restricted to undirected graphs. However, in our problem we would like to perform edge contractions on a subgraph of the MRRG to obtain the DFG. Thus we need to develop the definition of a **restricted graph minor for directed graphs**. Similar to the undirected case, we need to define edge contraction for directed graphs.

Edge contraction: An edge contraction operation in a directed graph removes an edge by merging the start node and end node of the

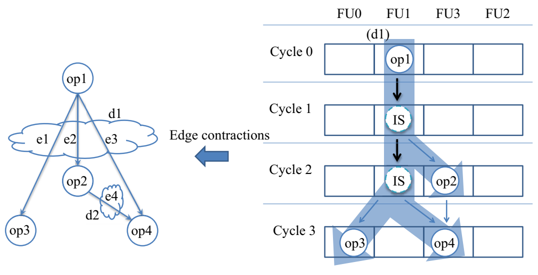
[1]More formally, an undirected graph *H* is a minor of another undirected graph *G* if a graph isomorphic to *H* can be obtained from *G* by contracting some edges, deleting some edges, and deleting some isolated vertices. The order in which a sequence of such contractions and deletions is performed on *G* does not affect the resulting graph *H*.



(a) Original graph (b) Graph generated after edge contraction

Fig. 5. An example of edge contraction in a directed graph.

directed edge into one. Fig. 5 shows an example of edge contraction in a directed graph. After the contraction of the bold edge, the start node and the end node of the edge now are merged into a new node. All the edges originally connected to/from the two nodes are now connected to/from the new node.



(a) DFG of a simple loop (b) Schedule and route graph

Fig. 6. Graph minor relationship between DFG and MRRG

Restricted edge contraction: We need to impose some restrictions on the edges that can be contracted. A directed edge $e = (u \to v)$ in the schedule and route graph (a subgraph of the MRRG) can be contracted if and only if the end node v of the edge e corresponds to either an intermediate storage or a register file. In other words, a directed edge cannot be contracted if its end node represents an operation from the DFG. Clearly, edge contractions in shared data routes can generate the corresponding fragment of the DFG. As an example consider the DFG in Fig. 6(a). The corresponding schedule and route graph appears in Fig. 6(b). The highlighted portion represents data routing from the source node *op1* to all its destination nodes. Route sharing is present through the two intermediate storage (IS) nodes. As we can see, contracting the two bold edges in the schedule and route graph can recreate the original DFG.

Invalid Mapping: Once we have identified a subgraph of the MRRG from which the DFG could be obtained through restricted edge contraction, we have to reconstruct the cycle-by-cycle modulo schedule to ensure the correctness for the identified subgraph. It may not be feasible to reconstruct this modulo schedule for some identified subgraphs due to the wrap-around nature of the MRRG. For example, consider the identified subgraph of the MRRG shown in Figure 2(h) for the DFG shown in Figure 2(d). When we attempt to reconstruct the module schedule, we realize that the edge e_3=*(op2,op4)* is routed through three IS nodes. On the other hand, the edge e_4=*(op3, op4)* is routed directly. Thus the input to *op4* along the edge e_3 is reached in cycle 4, while the input along the edge e_4 is reached in cycle 2. Thus the mapping is invalid.

The validity check could be performed during the mapping process by explicitly assigning time stamps (cycle values). For example, when the first node of the DFG is mapped to a MRRG node, we could assign it a time stamp of 0. This time stamp is propagated along the edges and incremented (or decremented as detailed in Section IV-C2) by one every time it passes through a node in a different tier (cycle) in the MRRG (see Fig. 2(h)). One can proceed with the mapping of an internal DFG node to a MRRG node only if the time stamps passed along all the connected edges are identical.

Restricted Graph Minor: We can now define application mapping on the CGRA as finding a valid subgraph *G'* of the MRRG such that the DFG can be obtained through repeated restricted edge contractions of *G'*. We call the DFG a restricted minor of the MRRG.

C. Mapping algorithm

Graph minor is an important concept in graph theory. Unfortunately, there does not exist an appropriate algorithm to test if graph *H* is a minor of graph *G*. Fortunately for us, we are considering a somewhat constrained version of the problem, where we would like to check if the DFG is a restricted minor of the MRRG. We solve this graph minor testing problem through a systematic and intelligent search of the mapping alternatives with effective structural and pruning constraints (similar to state space search widely used in solving graph matching problems [13])

The basic idea of our algorithm is as follows. 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 corresponding to the CGRA architecture. We now attempt to find a subgraph *G'* in the MRRG such that the DFG is a restricted minor of *G'*. If we can find such a subgraph *G'*, then the DFG can be mapped to the CGRA with *II* initiation interval. Otherwise, we increment II by one, create the MRRG corresponding to this new II value, and again try to find a subgraph *G'* in the new MRRG such that the DFG can be a minor. This process is repeated till we have generated a MRRG with sufficiently large value of II so that the DFG can satisfy the graph minor test. Algorithm 1 shows the pseudo code for our iterative modulo scheduling framework.

Algorithm 1: Mapping Algorithm

```
begin
    order_list = DFG_node_ordering();
    II = min(resMII, recMII);
    while do
        /*Create MRRG with II*/;
        Initialization();
        if Minor(DFG, MRRG,II,PMap) == 1 then
            break;
        II++;

Func Minor(DFG,MRRG,II,PMap)

begin
    get next unmapped dfg_node;
    /*Node mapping*/
    for all possible MRRG → node do
        Check_node_mapping_constraits(W_MRRG→node);
        save(PMap);
        PMap = map(dfg_node, MRRG → node);
        successful = Minor(DFG, MRRG, II, PMap);
        if successful == 1 OR all dfg nodes have been mapped then
            return 1;
        restore(PMap);
    /*Data routing through edge mapping*/
    for all possible MRRG → node do
        for every dfg_edge between PMap → dfg_node_set and dfg_node do
            Check_edge_mapping_constraits(MRRG→node);
            backup(PMap);
            PMap = map(data(dfg_edge), MRRG→node);
            successful = Minor(DFG, MRRG, II, PMap);
            if successful == 1 then
                return 1;
            restore(PMap);
```

Recall that given the DFG and the MRRG, we have to decide whether the DFG is a restricted minor of the MRRG by identifying a subgraph of the MRRG such that the DFG can be obtained through a series of restricted edge contractions. The subgraph of the MRRG then represents the place and route of the DFG on the MRRG. Instead of finding a subgraph *G'* of the MRRG and then performing a sequence of restricted edge contractions to reach the DFG, we take the dual approach. Starting with the DFG, we perform a series of node and edge mappings to create the subgraph *G'* of the MRRG. The mapping process *Minor()* in Algorithm 1 starts with an empty set. At each step, we add either a node or an edge mapping to the current partial mapping. If it is not feasible to add any node or edge mapping to a partial mapping without violating restricted graph minor constraints, then current partial mapping is eliminated and the search process backtracks to the previous valid partial mapping to attempt a new node or edge mapping. Exploitation of route sharing is embedded in the constraints while looking for the candidate MRRG nodes for either node mapping or edge mapping. The details will be explained later in this section. The mapping process continues till we have either found a complete mapping (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.

As we perform an exhaustive exploration, that is, we consider all possible mappings between the DFG and the MRRG, 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 extremely 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. 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 either quickly read a valid mapping or achieve substantial pruning that help establish the absence of any valid mapping.

1) DFG node ordering: An appropriate ordering of the DFG nodes during mapping is crucial to quickly find a feasible solution. To achieve this goal, we employ an ordering that helps us validate and meet the time stamp constraints as discussed in Section IV-B. The idea of the ordering is simple. A node *v* can be mapped only when at least one of its direct predecessor or direct successor nodes has been mapped. That is *v* should appear in the ordering after at least one of its direct predecessor or direct successor nodes. The only exception is the first node in the ordering. The advantage of this ordering is that the time stamps are generated appropriately for the nodes so that time stamp conflict at edges can be avoided easily. Moreover, we can filter out the invalid mappings of *v* due to mismatched time stamps at the edges of *v*.

We also impose an additional constraint that the nodes along the critical path has higher priority , i.e., they appear earlier. The higher priority of the critical path nodes is motivated by the fact that 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.

Fig. 7(b) shows a simple DFG and Fig. 7(c) shows the ordering of the nodes in the DFG (arrow signs). We start with the input node *op1* on the critical path. We proceed along the critical path ordering nodes *op3*, *op4*, *op5*. We could not order *op2* because none of its direct predecessors or direct successors appeared in the ordering. After *op5*, we are free to include *op2* in the ordering.

When the DFG contains disjoint parts, a new time stamp is generated and propagated for every disjoint component during the mapping process. The relative ordering of the components is not important for our problem.

2) Mapping Example: It is best to illustrate the mapping process with an example. We have a simple DFG as shown in Fig. 7(b) being mapped to a 2x2 CGRA array. FU2 in the CGRA is a memory node. Any operation from the DFG with the exception of memory operation can map to any functional unit in the CGRA. The memory operation *op4* can only map to a memory unit (FU2). Let us assume that we are currently considering II=2. For simplicity of exposition, we only draw the utilized edges in the MRRG. The entire mapping process is illustrated in Fig. 7(d-m).

The process starts with mapping *op1* and *op3* to FU1 in different cycles. We also generate the time stamps. Now we would like to map *op4*, which can only be mapped to FU2. However, we cannot map *op4* to FU2 in either cycle 0 or cycle 1 because of missing direct edges from *op1* in cycle 0 and *op3* in cycle 1. As node mapping fails, we have to perform mapping of the edges *e1* and *e2* first and have to resort to the use of intermediate storage (IS) for edge mapping as shown in partial mapping $PMap_3$. After a series of mapping for the edges, *op4* can be mapped in cycle 0 and the time stamp constraint is satisfied. Unfortunately, we have used up all the resources with IS and the remaining operations cannot be mapped. So we are forced to backtrack. After backtracking, we attempt an alternative route for edge *e2* through FU2 in $PMap'_3$. Finally *op4* is successfully mapped in $PMap_4$ with enough resources left.

The next partial mapping illustrates data route sharing. We are now attempting to map *op5*, two of whose predecessors *op1* and *op4* have been mapped previously. Again we cannot find a valid node mapping with direct connection from the predecessors. So we try to map the edge *e3*. At this point, we explore an interesting option of routing *e3* by sharing a node from the route of *e2*. This is shown in $PMap_5$ where the IS in cycle 1 is shared between the two routes. Then *op5* could be mapped and assigned a time stamp of 3. Finally, the last node *op2* is mapped with time stamp of 2, which is consistent with the time stamp of the other incoming edges of node *op5*. Notice that the time stamp can be propagated backwards from a child node to its parent node in DFG. For example, when we map *op2*, the time stamp of the child node *op5* is propagated to the parent node *op2*. In this case, the time stamp is decreased rather than increased.

3) Constraints: The constraints play a crucial role in pruning the search space effectively. The constraints can be divided into two categories: structural and pruning constraints. Structural constraints are imposed by inherent properties and structure of the DFG and the MRRG as well as time stamp matching requirement. We use the following structural constraints to ensure validity of the mapping.

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 the attribute of a MRRG node may signify that it supports memory operations. Attribute constraints ensure that a DFG node or edge is mapped to a MRRG structure with matching attribute. For example, a memory operation in the DFG can only be mapped to a functional unit supporting memory access. The register files in the MRRG can only be used for routing data during edge mapping. However, any functional unit or register can be used to route data if necessary.

Data routing constraint for edge mapping: The data produced by a source node may need to be routed to all its direct successors through multiple edge mapping steps. To enable sharing among these routes, we remember the intermediate nodes used to route an edge *e*. While mapping another edge *e'* with the same source node as *e*, the routing for *e'* continues from one of the intermediate nodes of *e*

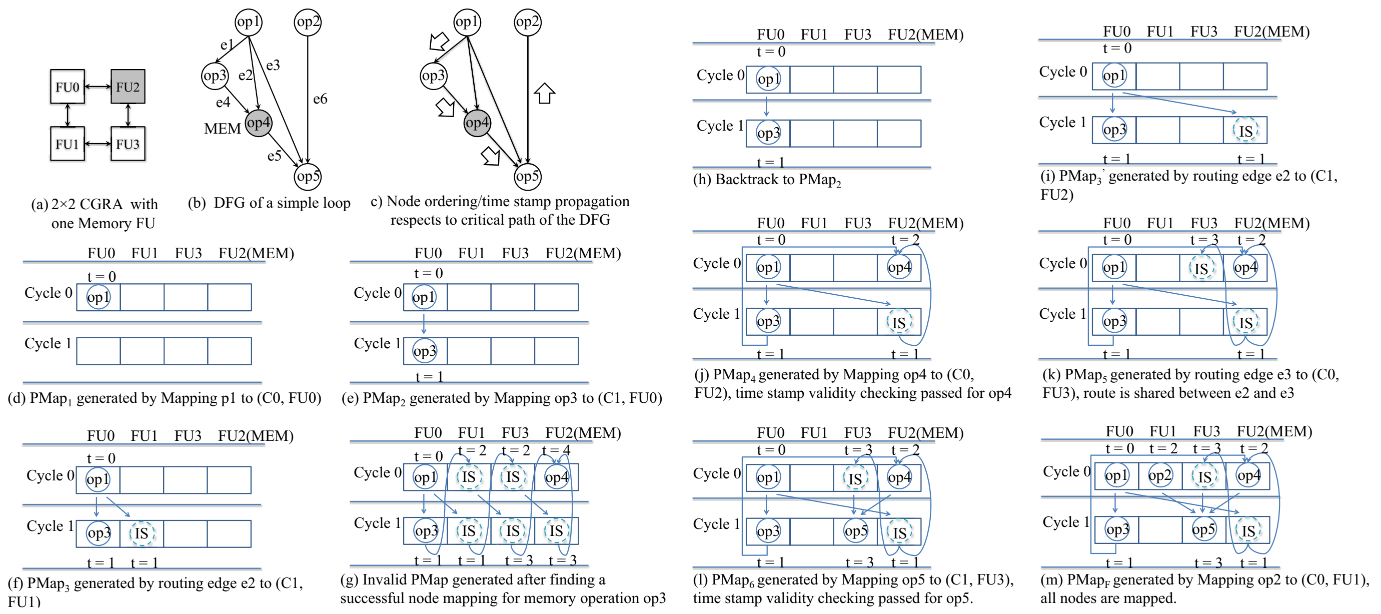

(a) 2×2 CGRA with one Memory FU
(b) DFG of a simple loop
c) Node ordering/time stamp propagation respects to critical path of the DFG
(h) Backtrack to PMap₂
(i) PMap₃' generated by routing edge e2 to (C1, FU2)
(d) PMap₁ generated by Mapping p1 to (C0, FU0)
(e) PMap₂ generated by Mapping op3 to (C1, FU0)
(j) PMap₄ generated by Mapping op4 to (C0, FU2), time stamp validity checking passed for op4
(k) PMap₅ generated by routing edge e3 to (C0, FU3), route is shared between e2 and e3
(f) PMap₃ generated by routing edge e2 to (C1, FU1)
(g) Invalid PMap generated after finding a successful node mapping for memory operation op3
(l) PMap₆ generated by Mapping op5 to (C1, FU3), time stamp validity checking passed for op5.
(m) PMap_F generated by Mapping op2 to (C0, FU1), all nodes are mapped.

Fig. 7. Illustrative example of partial mapping and backtracking during mapping space search for restricted minor test

as shown in *PMap*$_5$ in Fig. 7.

Data routing constraint for node mapping: A node can be mapped only after data from/to all direct predecessor/successor nodes in the partial mapping can be routed to/from this node successfully. This requires direct connection from/to direct predecessor/successor nodes and no conflict in time stamp values. Otherwise, the edges from/to direct predecessor/successor nodes have to be mapped first through intermediate storage.

Recurrence edge constraint: The data routing for a recurrence edge requires a different time stamp check that respects its recurrence distance. Let *tstamp* be a function that returns the time stamp value assigned to a MRRG node during the search process and *f* be a function that returns the MRRG mapped corresponding to a DFG node. If an edge $e = (u, v)$ is a recurrence edge with recurrence distance *d* in the DFG, then the data routing path identified by this recurrence edge from source MRRG node *f(u)* to the destination MRRG node *f(v)* should have a length equal to $II \times d - tstamp(f(v)) + tstamp(f(u))$.

Register file constraint: Both node mapping and edge mapping should ensure availability of register file read/write ports and capacity in the corresponding cycle if the data is routed from/to the register file. The routing through the same register in consecutive cycles does not require any port.

Pruning constraints: Pruning constraints help us look ahead in the future and quickly identify if the current valid partial mapping can be extended to a successful final mapping. If not, then we do not need to explore this partial mapping further. This looking ahead helps us eliminate a partial mapping *PMap* that will fail in the future, thereby pruning substantial portion of the design space consisting of all the children of *PMap*, that is, all the partial and complete mappings that can be obtained by extending *PMap*.

Unlike structural constraints, most of which have *O(1)* complexity, the pruning constraints are usually more complex and have higher time and space complexity. The pruning constraints we integrate in our search process are listed in Table I. A node mapping pair is denoted as *(m, f(m))*, where *m* is a DFG node and *f(m)* is the corresponding MRRG node. A predecessor/direct predecessor of a node *n* in a graph (DFG or MRRG) is denoted as *pred(n)/direct_pred(n)*, and similarly a successor/direct successor of a node *n* could be denoted as *succ(n)/direct_succ(n)*. Normally, a constraint with stronger pruning ability would require more running time and space. For example, constraint *C_pred_succ* and *C_feasible* have strong pruning ability but both require the reachability information for nodes in the partial mapping to/from free nodes in the MRRG. We build the reachability matrix using an efficient algorithm by Italiano et al. [8] with *O(N)* time complexity but $O(N^2)$ space overhead, where N is the number of nodes in MRRG. We build two matrices both with M_0 rows for M_0 DFG nodes within the partial mapping bringing the time complexity to $O(M_0N)$.

D. 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 (i.e., whether a valid mapping exists from the DFG to the MRRG) for each value of II, starting with the minimum possible value. However, even with the pruning strategies, the runtime of the optimal algorithm can be prohibitive when both the number of DFG nodes and CGRA functional units are quite large. Therefore, we integrate some heuristics to speed up the search process. This may introduce sub-optimality, i.e., the search process may miss a valid mapping at lower II value even though it exists. But the search process becomes extremely fast.

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. For example, consider the DFG in Fig. 4. Let us suppose the node ordering is *op1, op2, op3, op4, op5*. If node mapping fails for *op5*, normal search process would backtrack to the previous mapping, i.e., node *op4*. However, *op4* is unrelated to *op5*. So instead we backtrack to the node *op3*.

The second heuristic is motivated by the edge-centric mapping for CGRA [15]. During graph minor testing, instead of enumerating all possible edge mappings to route data to node n, the procedure

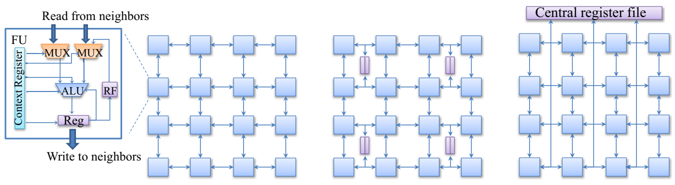
Constraint	*Description*	*Complexity*	*Pruning Ability*
C_degree	*For DFG node m in the partial mapping, check the number of available direct_pred(f(m)) in MRGG. The number should be larger than the number of unmapped direct_pred(m). On the other hand, if there is any unmapped direct_succ(m), then we should have at least one available direct_succ(f(m)) or one direct_succ(f(m)) used to route the data provided by f(m).*	*Time: O(N)*	*Medium*
C_units	*Check the number of available FUs in MRRG. This number should be larger than the number of unmapped DFG nodes.*	*Time: O(1)*	*Medium*
C_pred_succ	*For every node mapping pair: (m, f(m)), check the number of all the free FUs that could be reached by f(m) or any MRRG nodes used for routing the data provided by f(m) through a path only containing free nodes. This number should be larger than the number of unmapped succ(m). Similarly, check the number of all the free FUs that could reach the node f(m) by a path containing free nodes. This number should be larger than the number of unmapped pred(m).*	*Time:* $O(M_0N)$ *Space:* $O(N^2)$	*Strong*
C_feasible	*For every unmapped DFG node m, there is at least one free MRRG node n so that n could reach every f(direct_succ(m)), if direct_succ(m) is in the partial mapping, through a path containing only free MRRG nodes. Meanwhile, for a direct_pred(m) in the partial mapping, this free MRRG node n should be reachable from f(direct_pred(m)) or it could be reached by a MRRG node used for routing the data provided by f(direct_pred(m)).*	*Time:* $O(M_0N)$ *Space:* $O(N^2)$	*Strong*

TABLE I
PRUNING CONSTRAINTS.

simply aims to find one feasible route. Once a feasible route has been found and the search progresses, even if we backtrack to node n, the procedure does not attempt to find alternative routes.

The final heuristic helps us to escape from extensive mappings to route one edge. We put a counter for each of the partial mapping generated by node mapping. The counter is increased every time we backtrack to this partial mapping. Once the counter reaches a pre-defined threshold value, we eliminate the partial mapping and backtrack to its predecessors. Our experimental evaluation reveals that this is the only heuristic that sometimes prevent us from reaching a feasible solution even if one exists.

V. EXPERIMENTAL EVALUATION



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

Fig. 8. 4×4 CGRA with different register file configurations

We now proceed to evaluate the quality and the efficiency of our mapping algorithm. Our target CGRA architecture is a mesh-like 4×4 array with different register file configurations (Figure 8). The 4×4 array is the fundamental unit in many CGRA architectures including ADRES and MorphoSys. The 4×4 array has also been widely used to evaluate various mapping algorithms [14], [15], [10], [7], [9], [3]. The functional units in the array can be heterogeneous or homogeneous and each of them is connected to its immediate neighbors. We evaluate three different register file configurations denoted as NORF (architecture with no RF shown in Figure 8(a)), LRF (architecture with local shared RF shown in Figure 8(b)) and CRF (the architecture with central shared RF shown in Figure 8(c)). We also use *Homo* to denote homogeneous functional units and *MxC* to denote the availability of x columns of memory units. 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.

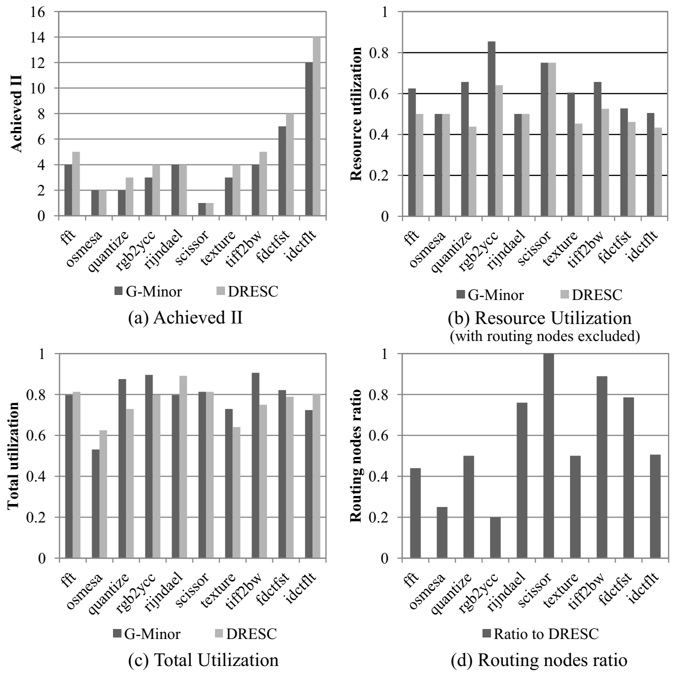
We select a set of loop kernels from MediaBench and MiBench benchmark suite shown in Table II. The DFGs for these kernels are generated from Trimaran [1] back-end using Elcor intermediate representation [2]. The column MII shows the minimal initiation interval imposed by resource and recurrence constraints. The MIIs are computed for three different configurations.

Comparison with DRESC: The most well-known algorithm and arguably the one that can achieve the best performance (smallest

Kernel	*#ops*	*#MEM ops*	*#edges*	*MII(Res, Rec) (Homo/M1C/M2C)*
fft	40	20	42	3/5/3
osmesa	16	9	17	1/3/2
quantize	21	8	24	2/2/2
rgb2ycc	41	15	44	3/4/3
rijndael	32	13	35	2/4/2
scissor	12	4	13	1/1/1
texture	29	7	31	2/2/2
tiff2bw	42	20	50	3/5/3
fdctfst	59	16	80	4/4/4
idctflt	87	25	114	6/7/6

TABLE II
BENCHMARK CHARACTERISTICS.

II) for CGRAs is DRESC [12], a simulated annealing based modulo scheduling approach. Although edge-centric modulo scheduler (EMS) [15] is widely considered to be the most efficient, EMS reports 10–13% degradation in schedule quality compared to DRESC, while compilation time is reduced by 27–46%. We show that our graph minor approach, referred to as G-Minor, can generate better schedules compared to DRESC, while compilation time is reduced by orders of magnitude. As DRESC does not explicitly handle routing through register files and heterogeneous FUs, we restrict our target architecture for this comparison to a homogeneous 4×4 CGRA with no register file *Homo-NORF*. All the functional units in this CGRA are comprehensive function units capable of handling any operation.



(a) Achieved II (b) Resource Utilization (with routing nodes excluded)
(c) Total Utilization (d) Routing nodes ratio

Fig. 9. Schedule quality of G-Minor and DRESC

Figure 9 shows the schedule qualities for G-Minor and DRESC. The achieved II in Figure 9(a) shows that G-Minor obtains equal or smaller II compared to DRESC for all benchmarks. Notice that the reduction of II by one improves the throughput by $100/II$ percentage. This also leads to better resource utilization (with routing nodes excluded) for G-Minor as shown in Figure 9(b) where G-Minor achieves 62% resource utilization on an average compared to 54% for DRESC. Both schedulers achieve high total resource utilization when routing resources are included (average 79% shown in Figure 9(c)). G-Minor is more efficient in utilizing routings resource even in the case when both schedulers achieve the same II, as shown in Figure 9(d).

	Average compilation time (seconds)	*Memory usage (Megabytes)*
G-Minor 4×4	3.4	< 3
DRESC 4×4	8045.2	< 2
G-Minor 8×8	15.6	< 8

TABLE III
COMPILATION TIME AND MEMORY USAGES.

Scalability: The average compilation time across all the benchmarks is presented in Table III. G-Minor achieves substantial reduction in compilation time compared to DRESC. To test the scalability of G-Minor approach, we also report the average compilation time to map the 10 benchmarks to 8 × 8 CGRA. The average compilation time is still minimal. However, for DRESC, we fail to obtain a solution for most benchmarks on 8 × 8 CGRA even after running for a day largely owing to expensive alternative routing computations for avoiding congestions [12]. This demonstrates the scalability of G-Minor approach. The memory usage reported in Table III reveals that both the schedulers have similar memory footprint.

Different CGRA configurations: Our mapping approach can support diverse CGRA architectures through parameterization. This is because we can fully exploit the structural information of the CGRA for explicit data routing. Our register file modeling approach is highly flexible in that it can support many different register file configurations. The experiment results for different CGRA configurations with different number of memory units and different register file configurations are shown in Figure 10.

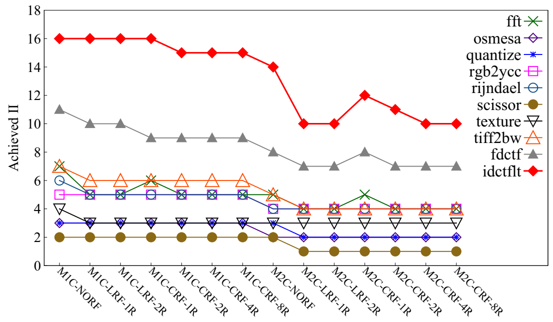


Fig. 10. Achieved II for different CGRA configurations.

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 result contradicts previous work on register file evaluation for CGRAs [10] that recommends a global register file with relatively large number of registers. This is because a mapping algorithm that performs register allocation as a post-processing step may end up with a schedule using a large number of registers. But increasing the number of read/write ports for a register file, specially when the register file is global and is connected to all the functional units, is an expensive proposition from both area and power perspective. Instead, our scheduler aims to achieve smaller II with limited number of registers. Moreover, as we share nodes across routes, register pressure is further alleviated. Thus we notice that starting from M2C-LRF-1R configuration, increasing the number of registers and providing more connectivity through registers for routing does not reduce the achieved II any further.

VI. Conclusions

In this paper, we provide a comprehensive approach for application mapping on CGRA. We argue and experimentally validate that node sharing across routes can substantially improve the quality of the schedule. We formalize this new mapping problem with node sharing across routes as a restricted version of the graph minor problem. We propose an efficient algorithm to solve this problem that effectively navigates through the mapping alternatives. Experimental evaluation confirms that our approach can generate better quality schedules with minimal compilation time.

VII. Acknowledgments

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033.

References

[1] The trimaran compiler infrastructure. http://www.trimaran.org.
[2] S. Aditya et al. Elcors machine description system. Technical report, HPL-98-128, 1998.
[3] N. Bansal et al. Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations. In *Micro*, 2003.
[4] G. Dimitroulakos et al. Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures. In *IPDPS*, 2006.
[5] S. Friedman et al. SPR: an architecture-adaptive CGRA mapping tool. In *FPGA*, 2009.
[6] R. Gnanaolivu et al. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *SoCpaR*, 2010.
[7] A. Hatanaka and N. Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *IPDSP*, 2007.
[8] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(2-3), 1986.
[9] Y. Kim et al. High throughput data mapping for coarse-grained reconfigurable architectures. *TCAD*, 30(11), 2011.
[10] Z. Kwok and S.J.E. Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *FCCM*, 2005.
[11] B. Mei et al. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *FPL*, 2003.
[12] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE*, 2003.
[13] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982.
[14] H. Park et al. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *CASES*, 2006.
[15] H. Park et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT*, 2008.
[16] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Micro*, 1994.
[17] N. Robertson and P. D. Seymour. Graph minors. *J. Comb. Theory Ser. B*, 77(1), 1999.
[18] H. Singh et al. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5), 2000.
[19] B.D. Sutter et al. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *LCTES*, 2008.