An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping

S. Alexander Chin
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
xan@ece.utoronto.ca

Jason H. Anderson
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
janders@ece.utoronto.ca

ABSTRACT

Coarse-grained reconfigurable architectures (CGRAs) have gained traction as a potential solution to implement accelerators for compute-intensive kernels, particularly in domains requiring hardware programmability. Architecture and CAD for CGRAs are tightly intertwined, with many prior works having combined architectures and tools. In this work, we present an architecture-agnostic integer linear programming (ILP) approach for CGRA mapping, integrated within an open-source CGRA architecture evaluation framework. The mapper accepts an application and an architecture description as input and can generate an optimal mapping, if indeed mapping is feasible. An experimental study demonstrates its effectiveness over a range of CGRA architectures.

ACM Reference Format:
S. Alexander Chin and Jason H. Anderson. 2018. An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping. In *DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3195986

1 INTRODUCTION

CGRAs contain large coarse-grained ALU-like logic blocks and employ datapath-style wide interconnect. When a CGRA's compute and interconnect capabilities align closely with application needs, higher speed and energy efficiency can be realized relative to FPGAs, as CGRAs are closer to ASICs on the scale of programmability [9]. Two commercial CGRAs have appeared in the market, the Samsung Reconfigurable Processor [8] and the STP Reconfigurable Processor from Renesas Electronics [22]. Aside from these, a considerable number of architectures and mapping techniques for CGRAs have been developed [1, 4, 7, 20, 21] in academia. In the context of CGRAs, *mapping* refers to scheduling, placing and routing an application onto a CGRA. In this paper, we consider CGRA mapping for *generic* CGRA architectures; that is, both the application, as well as the CGRA architecture model are an *input* to the mapper.

Recent CGRA mapping approaches rely on a graph representation [14] of the CGRA architecture within the mapping flows [2, 5, 13, 14, 18, 19]. The flexibility offered by this graph representation generically captures the capabilities of CGRAs with fixed-latency datapaths and multiple execution contexts (where CGRA behavior changes on a cycle-by-cycle basis). In some prior works, a simulated annealing approach is used as the core algorithm to map an application onto a CGRA [5, 14], though other heuristic techniques have also been developed [2, 13, 18, 19]. In this work, we present an Integer Linear Programming (ILP) formulation for mapping applications onto CGRAs. By using ILP for CGRA mapping, one can provably determine mapping feasibility or infeasibility, unlike heuristic methods. Decisively knowing whether a mapping solution exists or not, and whether a solution is optimal, can be of benefit to architects and CAD experts. For example, the complexity or amount of routing or storage structures can be tuned down to the limit of 'mappability' on an architecture for a very specific application domain – eliminating extra silicon area and power. Since such a formulation creates a bound on what is achievable through heuristic methods, CAD experts benefit by being able to quantify the effectiveness of their own heuristic methods and through improvements in heuristic methods, the gap to the optimum can be narrowed. Constraint-based methodologies have been successfully employed in computer-aided design tools for reconfigurable architectures [6, 15] as well as other spatial computing architectures [11, 16, 23, 24]. Though our formulation bears some similarity with prior work, we believe this to be the first that applies directly to CGRAs modelled using Modulo Routing Resource Graphs (MRRGs) [14]. The MRRG model is extremely flexible and is able to model complex routing and computation units in one or more dynamic device contexts. Further discussion on MRRGs is given in Section 3.2.

Our mapper is integrated into our open-source CGRA architecture evaluation framework, CGRA-ME [3], wherein the target architecture is not fixed or 'templated', but is described in a generic language and provided to the CGRA mapping tool, permitting mapping and evaluation of a wide range of CGRA architectures. Allowing this generic description of CGRA architectures increases the complexity of the mapping problem, as minimal assumptions can be made about the architectures themselves. This is analogous to the well-known VTR fined-grained FPGA architecture evaluation framework [12], where the underlying CAD algorithms are reactive to the user-provided architecture. We believe the ILP-based mapper within the framework will be useful to the research community for investigating a wide range of CGRA design methodologies and architectures.

2 RELATED WORK

Mei et al. first proposed the Modulo Routing Resource Graph (MRRG) model in DRESC [14]. This novel work frames the constraints of the modulo scheduling problem, operator placement, and value routing, within the graph itself and subsequent works have capitalised upon this abstraction. The core algorithm within DRESC is simulated annealing. Following DRESC, SPR [5] uses a similar simulated annealing method with some additions.

More recently, many works have examined graph based approaches to mapping. Park et al. [18] used a graph theory technique called *graph embedding* to draw the target application onto a three dimensional target space representing functional units over time.

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3195986

Park et al. [19] followed on with edge-centric modulo scheduling. In this method, routing of values is prioritized, where operator placement is secondary. Chen et al. propose a *graph minor* approach to mapping [2]. Their algorithm involves node reductions on the MRRG to test if the application graph is a minor of the MRRG. Yet another graph based technique is proposed by Ma et al. [13].

Lee et al. [11] present two algorithms, not based on an MRRG approach, that are quite specific to their proposed architecture. The first is an architecture-specific ILP approach that optimizes for latency. The second approach is a list-scheduling and quantum-inspired evolutionary algorithm. Yoon et al. [23, 24] also incorporate ILP techniques into their work and Nowatzki et al. [16] present an ILP formulation for three specific template architectures. The major difference between other ILP formulations and this work is that our formulation is valid over any architecture from which an MRRG can be generated.

3 BACKGROUND

3.1 Data-Flow Graph

A data-flow graph (DFG) is a directed graph $G(V, E)$, where vertices represent operations and edges are data dependencies between operations. In many CGRA toolflows, a DFG is used to represent the kernel computations. The DFG accounts for all computation operations, as well as memory accesses and I/O. This graph representation is also able to capture loop-carried dependencies through back-edges within the DFG structure. Two example DFGs are shown in Fig. 5, and will be elaborated upon below.

3.2 Modulo Routing Resource Graph

The Modulo Routing Resource Graph (MRRG) [14] is an abstract representation of the physical architecture of a CGRA and in this work we use an MRRG to model the CGRA during mapping. The graph contains, as vertices, all of the resources of the CGRA: the routes within the physical architecture, the storage elements, and the functional units that execute operations, including I/O and memory access.

An MRRG is capable of modelling *multiple contexts*, which is a feature of many CGRAs. In these architectures, routes and/or functional units can be shared to perform different routes and/or operations on *subsequent* cycles in a repeating pattern. The MRRG is a 'modulo' graph that represents the resources available during multiple recurring cycles of operation of the architecture [14]. For example, with two contexts, the CGRA toggles between two configurations; with three contexts, the CGRA cycles through three configurations, and so on.

An MRRG is a directed graph $G(V, E)$, where each vertex, $v \in V$, represents a CGRA resource. There are two types of vertices in the graph, one set for the routing resources, *RouteRes*, and one for functional units, *FuncUnits*. Each edge, $e \in E$, represents connectivity between an element of *FuncUnits* and an element of *RouteRes*, or between two elements of *RouteRes*. The MRRG contains a replica of the device model graph for each context. Edges between nodes in different replicas represent the capability to pass data from one context to another. For the case of N contexts, there would be N replicas, indexed from 0 to $N - 1$. Edges are present between some nodes in replica i and replica $i + 1 \ mod \ N$, modelling the ability to produce data in a context, and consume the data in the subsequent context. For example, in the two context case, a node representing a register in context 0 may have an edge to a node representing a computational unit in context 1, representing the ability to compute a value in context 0 and then consume it in the next context. We illustrate properties of the MRRG in the following examples.

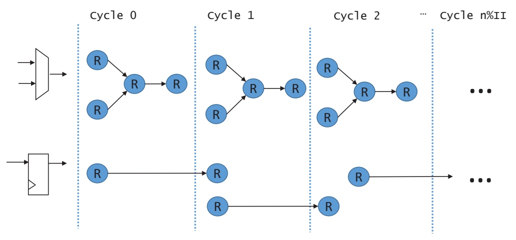

Figure 1: MRRG fragment for a multiplexer and register.

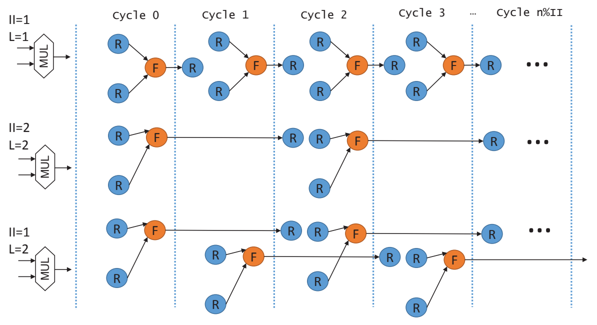

Figure 2: MRRG fragments for different latency and initiation interval functional units.

Fig. 1 shows the translation of a multiplexer and register primitive to their respective MRRGs. The 2-to-1 multiplexer shown can provide routing of values between functional units. The 2-to-1 multiplexer in this example is dynamically reconfigurable - it is able to route from different inputs in different cycles or *contexts*. The MRRG shown contains four nodes per cycle, two input and one output *RouteRes* and an internal multiplexing *RouteRes* that guarantees exclusivity to a single input. That is, on any cycle only one input can be routed to the output. Since it modelled as dynamically reconfigurable, this multiplexer may be reused on subsequent cycles for different routes, and multiple copies of this four node structure are present for each cycle (each context). In the case of the register in Fig. 1, it is modelled in the MRRG as a special wire as it moves a value from one cycle to the next. So its input node starts in cycle i and its output node ends in cycle $i + 1$. Each cycle this register can be reused, so this pattern is repeated for each cycle.

Fig. 2 shows the translation of three functional units, that perform a multiplication operation, with different latencies (L) and initiation intervals (II) and their respective MRRGs. The first example shows a functional unit that performs a 1-cycle multiply (latency of 1 cycle and initiation interval of 1 cycle). The MRRG for this unit consists of two input *RouteRes* vertices that are the operands of the multiply, a functional unit resource node that represents usage of the physical function at the associated timeslot, and an output *RouteRes* vertex. Since operation has a latency of 1-cycle, the output vertex is in the subsequent cycle. Since the initiation interval is 1-cycle, this functional unit can take inputs every cycle, so the MRRG is replicated every cycle. The second example in Fig. 2 shows a functional unit that performs a multiply that takes 2-cycles (latency of 2 cycles) with no pipelining (initiation interval of 2 cycles). Here, we have a similar structure but the output node is delayed by a second cycle and instead of repeating this structure every cycle, we repeat every 2 cycles since this resource is only available every two cycles (initiation interval of 2 cycles). In the last example in Fig. 2, we show a functional unit performing multiply

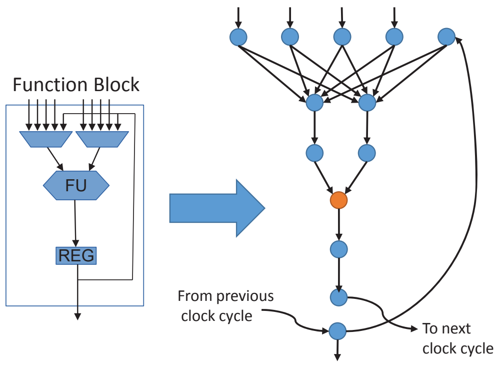

Figure 3: An example functional block that contains a functional unit (with a latency of 0 cycles), register and input multiplexers, and its corresponding MRRG structure for 1 cycle / 1 context.

with 2-cycle latency but is fully pipelined (initiation interval of 1 cycle).

Fig. 3 shows an example functional block datapath and its corresponding MRRG, illustrating how a larger MRRG is generated from the primitive components.

3.3 Mapping

CGRA mapping associates the operations and their connectivity within the DFG to the MRRG. The overall CGRA mapping problem is simplified through the use of the MRRG, as the MRRG frames the constraints of the modulo scheduling problem, operator placement, and data routing within the graph itself. Associating the DFG to the MRRG involves placing each operation in the DFG on a valid MRRG *FuncUnit* node, as well as finding a valid data routing through the *RouteRes* nodes to respective operations placed on other *FuncUnit* nodes. These *FuncUnit* nodes within the MRRG represent an execution time-slot of a physical functional unit - there could be multiple nodes that correspond to a single functional unit, but each node would represent execution on different device contexts. As long as all operations can be mapped to *FuncUnit* nodes, while having valid routes between the operations, a feasible mapping exists that respects data-dependence, signifying that the architecture is able to perform the necessary computations of the benchmark application.

4 ILP FORMULATION

4.1 Definitions

We first define four sets of items; the first two relate to the CGRA device model, the second two relate to the application being mapped into the CGRA:

- *FuncUnits*: contains all execution slots of every functional unit within the architecture – each corresponds to one functional unit node within the MRRG.
- *RouteRes*: contains all routing resources (wire, bus, multiplexer, or register) within the architecture: each corresponds to one routing resource node within the MRRG.
- *Ops*: all operations in the DFG to be mapped.
- *Vals*: all values produced by operations in the DFG to be mapped. Each *Vals* is split into *SubVals*. A sub-value represents a source to sink connection in a multi-fanout value. For example, there are two sub-values for the two fanout net in DFG B in Fig. 5.

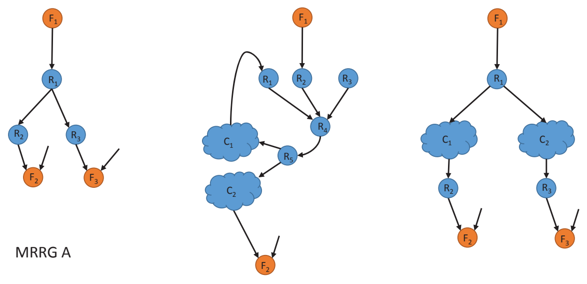

Figure 4: Three illustrative MRRGs.

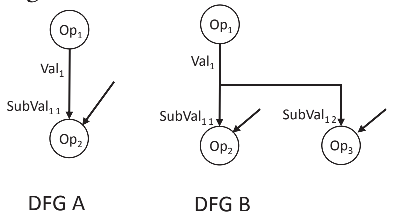

Figure 5: Two DFG fragments to illustrate value routing and multi-fanout.

We define three sets of binary variables. The first set of variables define the placement of operations onto functional units (a mapping from *Ops* to *FuncUnits*). The second set of variables define the use of routing resources by values (a mapping from *Vals* to *RouteRes*). The third set of variables are abstract, and define the routing path of values between functional units for each sink of a multi-fanout value. For each value, we create a new binary variable that associates routing resources, the value, and the termination point (sink). These variables are necessary for the formulation, as will be elaborated upon below. The three sets of binary variables are:

- $F_{p,q}$: functional-unit node p in the MRRG is used for supporting operation q in the DFG.
- $R_{i,j}$: routing node i is used for routing value j.
- $R_{i,j,k}$: routing node i is used for routing value j to value j's sink k.

As will be apparent in the formulation below, $R_{i,j}$ will be constrained to 1 whenever $R_{i,j,k}$ is 1 (i.e. for any sink k). The sink-specific variables, $R_{i,j,k}$, are required to achieve routing connectivity to each sink; the sink-agnostic variables, $R_{i,j}$, are used to ensure no routing-resource overuse and in the ILP objective function to minimize overall resource usage.

4.2 ILP Constraints and Objective Function

Operation Placement: Every operation in the DFG is placed on exactly one functional unit.

$$\sum_{p \in FuncUnits} F_{p,q} = 1, \forall q \in Ops \tag{1}$$

Functional Unit Exclusivity: Each functional unit slot (represented by *FuncUnits*) is occupied by at most one DFG Operation (i.e. there exist no overlaps among operations on functional units).

$$\sum_{q \in Ops} F_{p,q} \leq 1, \forall p \in FuncUnits \tag{2}$$

Functional Unit Legality: Ensure that operations are only placed on functional units that can implement the operation (applies to heterogeneous architectures). Here, $SupportedOps(p)$ is the set of operations that are able to be executed by functional unit p.

$$F_{p,q} = 0 \quad \forall p \in FuncUnits, q \in Ops \quad \text{where: } q \notin SupportedOps(p) \tag{3}$$

Route Exclusivity: Ensure that each routing resource is occupied by *at most* one value (i.e. multiple DFG Values cannot be routed on a single routing resource).

$$\sum_{j \in Vals} R_{i,j} \leq 1, \forall i \in RouteRes \tag{4}$$

Fanout Routing: Guarantee continuity of values between adjacent routing resources. We ensure for each fanout of a routing resource used by a value, there is at least one downstream routing resource that is used by the *same* value, whether this be another routing resource or the sink of the fanout (which is itself a routing resource). So, at least one output of a routing resource node must be driven with the same value, or it must terminate at the input of the downstream functional unit node.

$$R_{i,j,k} \leq \sum_{m \in fanouts(i)} R_{m,j,k} \quad \forall i \in RouteRes, \forall j \in Vals, \forall k \in sinks(j) \tag{5}$$

Implied Placement: Ensure that Fanout Routing terminates at the input of a functional unit, thereby implying a mapping of the downstream operation to the functional unit. Here, we use the $\rightsquigarrow$ symbol to denote the existence of an edge between two nodes in a directed graph. This constraint means that if the route for sink k of value j terminates at functional unit p, then operation q *must* necessarily be mapped onto functional unit p. This constraint also accounts for operand correctness in the case of non-commutative operations.

$$F_{p,q} \geq R_{i,j,k} \quad \forall p \in FuncUnits, \forall q \in Ops, \forall i \in RouteRes, \forall j \in Vals \quad \text{where: } \exists (j \rightsquigarrow q) \land (i \rightsquigarrow p) \quad \forall k \in sinks(j) \tag{6}$$

Initial Fanout: Ensure that the routing resources at the output of a functional unit are set to the output value of the mapped operation. The binary variable $R_{i,j,k}$ is set for each sink, k, of the output value. This constraint is only applied when there is an edge from q to j in the DFG and an edge from p to i in the MRRG. Again, we use the $\rightsquigarrow$ symbol to denote the existence of an edge between two nodes in a directed graph.

$$R_{i,j,k} = F_{p,q} \quad \forall i \in RouteRes, \forall j \in Vals, \forall p \in FuncUnits, \forall q \in Ops \quad \text{where: } \exists (q \rightsquigarrow j) \land (p \rightsquigarrow i) \quad \forall k \in sinks(j) \tag{7}$$

Example 1: Consider mapping DFG A in Fig. 5 to MRRG A in Fig. 4. If Op_1 is placed on $FuncUnit_1$, $F_{1,1} = 1$. Since the Value in DFG A has only one fanout, the Initial Fanout constraint ensures that $SubValue_1$ is associated with $RoutingRes_1$ with $F_{1,1} = 1 = R_{1,1,1}$. The Fanout Routing constraint then ensures that at least one of $R_{2,1,1}$ or $R_{3,1,1}$ is equal to 1. Application of the Implied Placement constraint on $RoutingRes_2$ and $RoutingRes_3$ allows the routing to terminate at $FuncUnit_2$ or $FuncUnit_3$ setting $F_{2,2} = 1$ or $F_{3,2} = 1$, placing Op_2.

Routing Resource Usage: Since routing is formulated on a sink-by-sink basis using sub-values, the following constraint ensures that routing resources are associated with the corresponding values.

$$R_{i,j} \geq R_{i,j,k} \quad \forall i \in RouteRes, \forall j \in Vals, \forall k \in sinks(j) \tag{8}$$

Multiplexer Input Exclusivity: Prevents self reinforcing routing loops that would terminate fanout routing within the loop instead of the required sink of the route. By disallowing multiplexer inputs from having the same value, loops are prevented.

$$R_{i,j} = \sum_{m \in fanins(i)} R_{m,j} \quad \forall i \in \{RouteRes \mid |fanins(i)| > 1\}, \forall j \in Vals \tag{9}$$

Example 2: Consider mapping DFG A in Fig. 5 to MRRG B in Fig. 4. If Op_1 is placed on $FuncUnit_1$ ($F_{1,1} = 1$), Initial Fanout and Fanout Routing would set $R_{2,1,1}$, $R_{4,1,1}$ and $R_{5,1,1}$ to 1. Routing can now continue through more routing resources in a cloud of connected routing resources in C_1 or C_2. *Without* the Multiplexer Input Exclusivity constraint, routing through C_1 and setting $R_{1,1,1} = 1$ is feasible. Now, the Fanout Routing constraint for $RouteRes_1$ is already satisfied since $R_{4,1,1} = 1$ and $SubValue_1$ has not been routed to any *FuncUnit*. By applying the Routing Resource Usage constraint and Multiplexer Input Exclusivity constraint, $R_{1,1,1} = 1$ is infeasible because $R_{2,1,1} = 1$. Fanout Routing at $RouteRes_5$ can now only be satisfied by routing through C_2, ultimately to the sink at $FuncUnit_2$.

Example 3: Consider mapping DFG B in Fig. 5 to MRRG C in Fig. 4. Val_1 has two fanouts – one to Op_2 and one to Op_3. Consider applying our routing constraints to *Values*, instead of *SubValues*. Op_1 is placed on $FuncUnit_1$ ($F_{1,1} = 1$), the Initial Fanout constraint would set $R_{1,1}$. Going through C_1, $R_{2,1}$ could be 1 fulfilling the Fanout Routing constraint and with the Implied Placement constraint, mapping either Op_2 or Op_3 to $FuncUnit_2$. If Op_2 is mapped to $FuncUnit_2$ ($F_{2,2} = 1$), $F_{3,3} = 1$ due to the Operation Placement constraint. Now, there is no other constraint that guarantees a connection of the value through C_2 and $RouteRes_3$ to $FuncUnit_3$. Hence, each sink is assigned a distinct *SubValue* for routing.

The objective function we use is minimizing the number of routing resources used by the mapping.

$$Minimize \sum_{\forall i \in RouteRes, \forall j \in Vals} R_{i,j} \tag{10}$$

However, it is straightforward to apply alternative objective functions, where, for example, specific types of resources have unique costs. For example, one can imagine that resources such as long wires, registers, register files or other data value routing structures contribute significantly to power consumption and these nodes could be weighted to optimize for power.

5 EXPERIMENTAL SETUP & RESULTS

To test our formulation, a number of benchmarks and architectures were considered. The architectures were chosen to be representative of CGRAs proposed in the literature with varying flexibility. Since higher degrees of flexibility generally increases hardware costs, it is interesting to see how much flexibility is enough to map a set of benchmarks. Each architecture comprises a 4×4 2D-array of functional blocks with 32-bit-wide bus-based interconnect. Each block

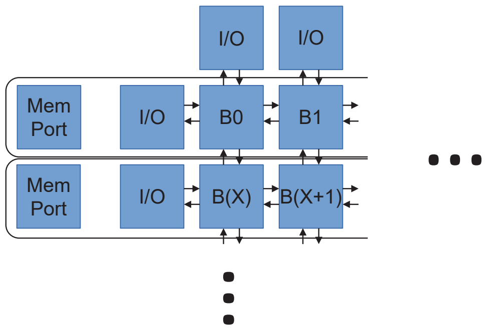

Figure 6: The arrangement of functional blocks in our test architectures for *Orthogonal* connectivity. The functional blocks are orthogonally connected to their nearest neighbours while the periphery contains I/Os. Each row of Functional Blocks share connectivity to one memory access port.

within our test architecture (shown in Fig. 3), contains a single functional unit ALU, an output register, and block I/O. The functional unit within the block can perform RISC-like operations such as `add`, `mul`, `shl`, etc. Each row within the array shares a memory port as shown in Fig. 6. This port is modelled as a special functional unit that can only perform `load` and `store` operations. We consider two functional block architectures and two interconnect architectures with multiple contexts. One set of functional block architectures, *Homogeneous*, has full fledged ALUs including a multiplier, whereas in the *Heterogeneous* architectures, only half of the ALUs in the architecture contain a multiplier. One style of interconnect architecture is *Orthogonal*, with each block having connectivity with the four ordinal directions. This connectivity is also depicted in Fig. 6. The other style of interconnect architecture is *Diagonal*, with each block having additional connectivity to diagonal blocks. For Diagonal interconnect, the size of each functional block's input multiplexer was increased to accommodate the additional inputs. Additionally, we model these architectures with 1 and 2 execution contexts.

The benchmark data-flow graphs were chosen to have varying number of operations, number of multiply operations and routing requirements. They reflect different mapping difficulties for the target architectures. The benchmarks are comprised of LLVM [10] compiled DFGs, leveraging CGRA-ME [3], as well as hand-crafted DFGs. Table 1 shows detailed characteristics of each benchmark.

CGRA-ME [3] was also leveraged for modelling the architectures and the existing mapping infrastructure. The framework allows for the generic specification of architectures in a high-level XML-based language. Detailed functional blocks and routing structures can be constructed directly within this language, and also the higher level connectivity such as how top-level blocks are integrated together. From this architecture model, the framework also has the capability of automatically generating a corresponding MRRG model for the mapper.

The ILP formulation was implemented as new mapper within our existing framework. To solve the ILP formulation, the Gurobi [17] solver was integrated into the framework. This sophisticated solver is free-to-use for academic use. The overall flow of mapping is shown and described in Fig. 7.

More than 80% of the runs completed within one hour and the ILP solver was able to determine feasibility/infeasibility for all formulations of benchmark/architectures except 2 that timed out after 24 hours (Table 2).

Benchmark	I/Os	Operations	# Multiplies
accum	10	8	4
mac	1	9	3
add_10	10	10	0
add_14	14	14	0
add_16	16	16	0
mult_10	10	9	9
mult_14	14	13	13
mult_16	16	15	15
2x2-f	5	5	1
2x2-p	6	6	1
cos_4	5	14	12
cosh_4	5	14	12
exp_4	4	9	5
exp_5	5	12	9
exp_6	6	15	14
sinh_4	5	13	9
tay_4	5	10	6
extreme	16	19	4
weighted_sum	16	16	8

Table 1: Benchmarks: showing the number of I/Os, internal operations and Multiplies. Load/Stores are considered to be internal operations.

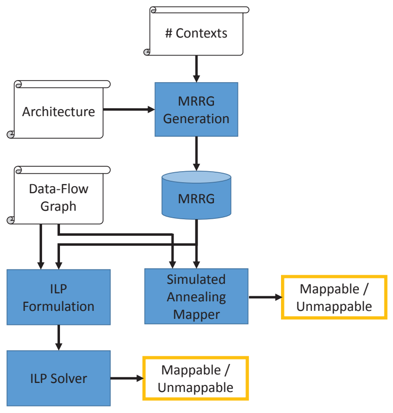

Figure 7: The mapping flow within our framework for ILP-based mapping and simulated annealing-based mapping. The architecture description and the number of contexts is input into the framework, where it generates an MRRG. For ILP-based mapping, the ILP formulation is created from the input DFG benchmark and the MRRG. The formulation is then solved by the ILP Solver, in our case Gurobi. The simulated annealing-based mapper takes the DFG and MRRG directly as inputs to map the benchmark to the architecture.

Benchmark	Single Context (II=1)				Dual Context (II=2)			
	Hetero. Orth.	Hetero. Diag.	Homo. Orth.	Homo. Diag.	Hetero. Orth.	Hetero. Diag.	Homo. Orth.	Homo. Diag.
accum	1	1	1	1	1	1	1	1
mac	1	1	1	1	1	1	1	1
add_10	1	1	1	1	1	1	1	1
add_14	0	1	0	1	1	1	1	1
add_16	0	1	0	1	1	1	1	1
mult_10	0	0	1	1	1	1	1	1
mult_14	0	0	0	1	1	1	1	1
mult_16	0	0	0	1	1	1	1	1
2x2-f	1	1	1	1	1	1	1	1
2x2-p	1	1	1	1	1	1	1	1
cos_4	0	0	0	0	1	1	1	1
cosh_4	0	0	0	0	1	1	1	1
exp_4	0	1	0	1	1	1	1	1
exp_5	0	0	0	1	1	1	1	1
exp_6	0	0	0	0	T	1	T	1
sinh_4	0	0	0	1	1	1	1	1
tay_4	0	1	0	1	1	1	1	1
extreme	0	0	0	0	1	1	1	1
weighted_sum	0	0	0	1	1	1	1	1
Total Feasible	5	9	6	15	18	19	18	19

Table 2: Mapping Results. 1 signifies a feasible mapping, 0 signifies an infeasible mapping. T, signifies a solver timeout where the solver was unable to find a feasible solution or prove infeasibility.

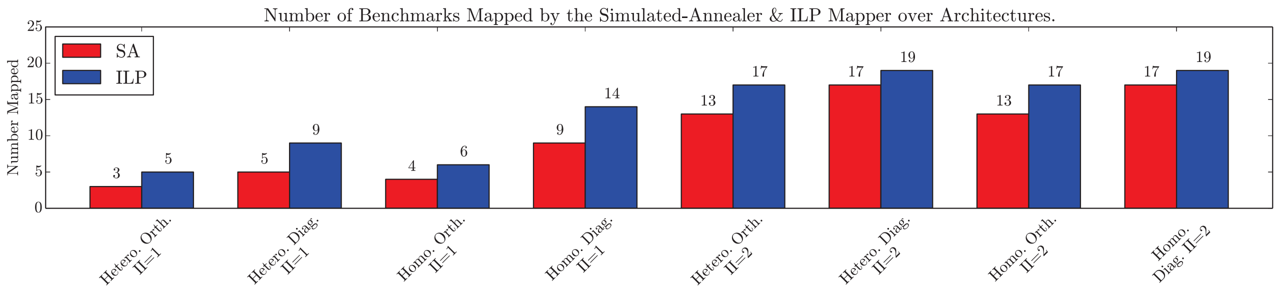

Figure 8: Comparison between the Simulated Annealing Mapper and the ILP Mapper.

Table 2 shows the full results of the ILP mapper over the eight CGRA architectures and 19 benchmarks. The single context architectures on the left of the table exhibit varying degrees of success to map each benchmark. The Heterogeneous Orthogonal architecture, the most constrained architecture, was only able to accommodate 5 benchmarks, whereas the most flexible, Homogeneous Diagonal, was able to map 15. It is also interesting to note that quite a few benchmarks were not able to be mapped in a single context (cos_4, cosh_4, and extreme). The Heterogeneous Diagonal and Homogeneous Orthogonal architectures show interesting trade-offs between reduced multipliers with extra routing and many multipliers with constrained routing. The mult_14 and mult_16 benchmarks both require many multipliers and routing, and are both unmappable. However the smaller mult_10 benchmark is able to map to the Homogeneous Orthogonal architecture.

All benchmarks were able to be mapped in dual context architectures aside from the 2 solver timeouts. This shows the flexibility that a second context gives at the price of halved throughput (since II=2) and extra hardware (and power) to support the second configuration context. If strict mappability for the given benchmark set is the architect's concern, a Heterogeneous Diagonal architecture (that contains half the multipliers of a Homogeneous architecture) with support for two contexts may be sufficient, since all benchmarks can be mapped to this architecture. Additionally, 9 of the benchmarks could still be mapped with higher throughput ($II = 1$) while the other 10 would need to be mapped with two configuration contexts ($II = 2$). But if performance is a concern, a Homogeneous architecture with Diagonal routing will be the best option since feasible mappings exist for the most benchmarks with $II = 1$.

For contrast, the annealing-mapper that is built into the existing framework was also run with moderate parameters (number of inner-loop iterations, penalty factors, temperature schedule, etc.) on the same set of benchmarks and architectures. A comparison of the results of both mappers is shown in the bar graph in Fig. 8. Overall, the ILP mapper is able to find more mapping solutions for all eight architectures.

6 CONCLUSION

The presented ILP formulation allows one to provably determine mapping feasibility or infeasibility and produce an optimal mapping. Architects are able to evaluate the 'mappability' of the architectures for sets of domain-specific benchmarks, allowing them to tune the flexibility of the architecture while staying isolated from CAD tool effects. CAD experts are more easily able to evaluate and improve upon the effectiveness of heuristic methods since this ILP formulation forms a bound on what is achievable. To these ends, this work enables future architecture exploration and CAD tool improvements.

The implementation of the formulation in this work is already integrated into our open-source CGRA framework, CGRA-ME [3], furthering the tools available to the CGRA research community.

REFERENCES

[1] Hideharu Amano. 2006. A Survey on Dynamically Reconfigurable Processors. *IEICE Transactions* 89-B, 12 (2006), 3179–3187.
[2] Liang Chen and Tulika Mitra. 2014. Graph Minor Approach for Application Mapping on CGRAs. *ACM TRETS* 7, 3, Article 21 (Sept. 2014), 25 pages.
[3] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A Unified Framework for CGRA Modelling and Exploration. In *IEEE ASAP 2017*. 184–189.
[4] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. 2013. *Coarse-Grained Reconfigurable Array Architectures*. Springer New York, 553–592.
[5] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: An Architecture-adaptive CGRA Mapping Tool. In *ACM FPGA*. 191–200.
[6] Marcel Gort and Jason H. Anderson. 2013. Combined Architecture/Algorithm Approach to Fast FPGA Routing. *IEEE Trans. on VLSI* 21, 6 (June 2013), 1067–1079.
[7] Reiner Hartenstein. 2001. Coarse Grain Reconfigurable Architectures. In *IEEE/ACM ASP-DAC*. 564–569.
[8] Changmoo Kim, Moo-Kyoung Chung, Yeon-Gon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2014. ULP-SRP: Ultra Low-Power Samsung Reconfigurable Processor for Biomedical Applications. *ACM TRETS* 7, 3 (2014), 22:1–22:15.
[9] Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Trans. On CAD* 26, 2 (Feb. 2007), 203–215.
[10] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE / ACM Intl. Symp. on Code Gen. and Opt.* 75–88.
[11] Ganghee Lee, Kiyoung Choi, and Nikil D. Dutt. 2011. Mapping Multi-Domain Applications Onto Coarse-Grained Reconfigurable Architectures. *IEEE Trans. on CAD* 30, 5 (2011), 637–650.
[12] Jason Luu et al. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM TRETS* 7, 2, Article 6 (July 2014), 6:1–6:30 pages.
[13] Lu Ma, , Wei Ge, and Zhi Qi. 2012. A Graph-Based Spatial Mapping Algorithm for a Coarse Grained Reconfigurable Architecture Template. *Inf. in Ctrl., Auto. and Robo.*, 669–678.
[14] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: A Retargetable Compiler for Coarse-grained Reconfigurable Architectures. In *IEEE FPL*. 166–173.
[15] Gi-Joon Nam et al. 2002. A New FPGA Detailed Routing Approach Via Search-Based Boolean Satisfiability. *IEEE Trans. on CAD* 21, 6 (2002), 674–684.
[16] Tony Nowatzki et al. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. *SIGPLAN Not.* 48, 6 (June 2013), 495–506.
[17] Gurobi Optimization. 2017. Gurobi Optimizer. (2017). http://www.gurobi.com/
[18] Hyunchul Park et al. 2006. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. In *IEEE/ACM CASES*. 136–146.
[19] Hyunchul Park et al. 2008. Edge-Centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In *IEEE/ACM PACT*. 166–176.
[20] Vaishali Tehre and Ravindra Kshirsagar. 2012. Survey on Coarse Grained Reconfigurable Architectures. *Intl. Jrnl. of Comp. Appl.* 48, 16 (2012), 1–7.
[21] Russell Tessier, Kenneth Pocek, and AndrÃĺ DeHon. 2015. Reconfigurable Computing Architectures. *Proc. of the IEEE* 103, 3 (2015), 332–354.
[22] Takao Toi and et al. 2013. Optimizing Time and Space Multiplexed Computation in a Dynamically Reconfigurable Processor. In *IEEE FPT*. 106–111.
[23] Jonghee W. Yoon et al. 2008. SPKM : A Novel Graph Drawing based Algorithm for Application Mapping onto Coarse-Grained Reconfigurable Architectures. In *ASP-DAC 2008*. 776–782.
[24] Jonghee W. Yoon et al. 2009. A Graph Drawing Based Spatial Mapping Algorithm for Coarse-Grained Reconfigurable Architectures. *IEEE Trans. on VLSI* 17, 11 (Nov 2009), 1565–1578.