Agile SMT-Based Mapping for CGRAs with Restricted Routing Networks

Caleb Donovick*, Makai Mann*, Clark Barrett*, Pat Hanrahan*
*Stanford University
{donovick, barrett, hanrahan}@cs.stanford.edu, makaim@stanford.edu

***Abstract*—Coarse-grained reconfigurable architectures (CGRAs) are becoming popular accelerators for computationally intensive tasks. CGRAs offer the reconfigurability of an FPGA, but with larger configurable blocks which provide performance closer to ASICs. CGRAs can achieve very high compute density if the routing networks are restricted; however, mapping using traditional annealing-based approaches does not perform well for such architectures. This paper uses Satisfiability Modulo Theories (SMT) solvers to rapidly map designs onto arbitrary CGRA fabrics. This approach is sound, complete, and in many cases an order of magnitude faster than state-of-the-art constraint-based mapping techniques using integer linear programming (ILP). Additionally, we propose a functional duplication strategy that decreases pressure on the routing network from high-fanout operations, leading to significant performance improvements.**

***Index Terms*—CGRA, Placement, Design Mapping, Satisfiability, SMT, Constraint Solving**

I. Introduction

For many computationally intensive tasks, such as image and signal processing applications, ASICs have often been used to meet performance and energy requirements. However, the high non-recurring engineering costs required to build ASICs have become prohibitive, especially with advanced technology nodes [1]. One solution to avoid the high cost of ASIC design is to use reconfigurable architectures such as FPGAs [1].

While FPGAs are cost effective and often more efficient than CPUs or GPUs for many tasks, there is still a significant gap between the efficiency of FPGAs and ASICs. Coarse Grained Reconfigurable Arrays (CGRAs), arrays of large compute blocks with word-level routing, have been proposed to offer the reconfigurability of FPGAs while achieving efficiency levels closer to ASICs.

One of the most glaring inefficiencies of FPGAs is their highly provisioned bit-level "island style" routing network, which can occupy as much as 80%-90% [2] of the chip. CGRAs with word-level networks and larger functional blocks have greater compute density, but 60%-70% of the area is still devoted to the routing network [3]. CGRAs with more restricted routing networks have been proposed, where interconnect accounts for less than 20% [4] of the area. However, mapping to such architectures – the process of scheduling, placing, and routing an application – is not easy, as traditional annealing-based approaches typically fail. Currently, these architectures often use custom mappers, relying on architecture-specific heuristics and patterns [5], [6], [4].

In this paper, we present a mapping technique which can be applied to arbitrary topologies, but is particularly useful for restricted routing networks where simulated annealing struggles to find a valid mapping. To illustrate, we focus on a class of routing networks that only allow routing between adjacent tiles. Further, routing through a tile prevents its functional unit from being used. In this setting, placement and routing must be considered simultaneously, as nearness cannot be used as a heuristic for routability.

We focus on the problem of quickly finding *some* valid mapping (i.e. not necessarily an optimal one). This capability is especially useful during design exploration, when a designer is trying different logical structures and mapping them to hardware for evaluation. In this scenario, keeping the iteration time low is crucial. Later, when the design is finalized, more computational resources can be allocated for optimizing the design. It is also desirable in this scenario for the designer to be notified if the design does not admit *any* valid mapping. We thus further restrict our attention to complete techniques, i.e. techniques which can always either produce a mapping or conclude that no valid mapping exists.

The leading complete approach for mapping to highly restricted routing infrastructures is Integer Linear Programming (ILP). CGRA-ME is a proposed ILP-based framework for performing mapping to restricted routing CGRAs [7]. We build on this framework by adding an alternative mapping approach[1]. We use Satisfiability Modulo Theories (SMT) encodings and solvers. Additionally, we propose a functional duplication strategy to decrease pressure on the routing network from high fanout operations. An experimental evaluation shows that for quickly finding mappings, our approach outperforms the ILP-based approach, by an order of magnitude or more in many cases.

II. Background

A. Data Flow Graphs

A data flow graph (DFG) is a directed graph, where each vertex represents an operation and each edge represents a data dependency between operations. DFGs are commonly used to model computation in compilers and mapping flows. The DFG represents all computation to be mapped to the CGRA, including computational kernels, IO, and memory. We define our DFG as the directed graph (Ops, $Vals$). Where Ops are

[1] available at github.com/cdonovick/pycgrame

978-1-7281-1957-1/19/$31.00

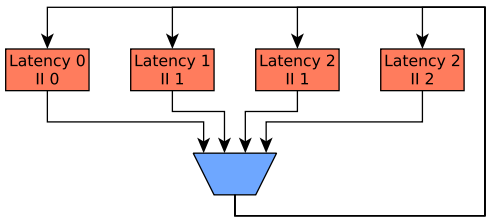


Fig. 1: Architecture Graph

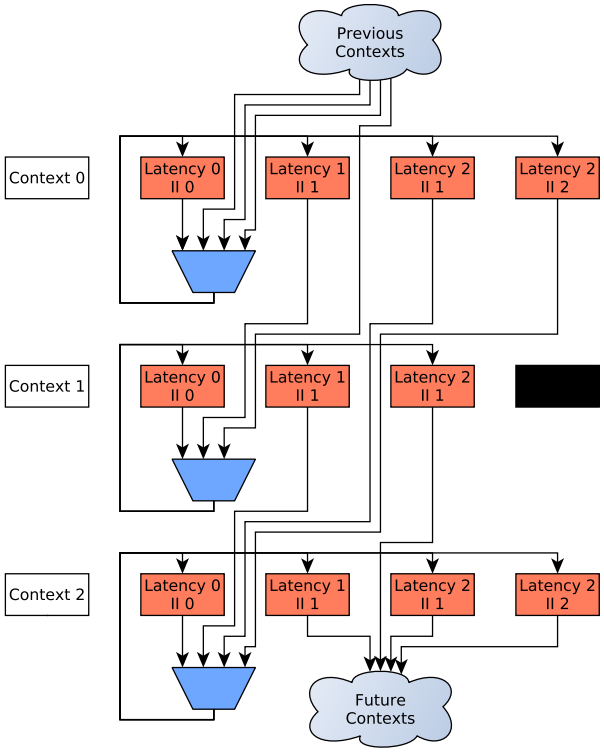


Fig. 2: Expanded MRRG
Note: all contexts are mod N.

the operations and *Vals* are the values produced by the *Ops*. Note that the members of *Vals* are hyper-edges connecting a single source operation to one or more destination operations.

B. Modulo Routing Resource Graph

A Modulo Routing Resource Graph (MRRG) [8] is an abstract representation of a CGRA architecture, which captures the network topology, the latency and initiation interval of operations, and its capacity to support multiple configuration contexts. A node in the MRRG represents a CGRA resource in both time and space. The graph contains nodes for each functional unit (e.g. arithmetic units, memory, IO, etc..) as well as for routing resources (e.g. muxes and registers).

Some proposed CGRA designs support multiple configuration contexts. In such architectures, configurations are periodically rotated. This allows larger computation graphs to be mapped onto smaller arrays. Data can be shared between contexts via registers.

A MRRG is a directed graph ($Nodes$, $Nets$) where each $n \in Nodes$ represents a CGRA resource in a specific context. Each $e \in Nets$ represents the connectivity between resources. The graph is "unrolled" in time with the netlist replicated for each context. A register node in context j outputs to a node in context $(j{+}1) \bmod N$ where N is the number of configuration contexts. In general, a node with initiation interval i and latency l exists only in contexts such that $j \bmod i = 0$ and produces outputs in context $(j + l) \bmod N$. We show an architecture with varying latencies and initiation intervals (II) in fig. 1 and its unrolled MRRG in fig. 2. Note how the node with II 2 is utilized in context 1 by the operation started in context 0, and so is absent from the graph at that point.

Arbitrary initiation patterns are also possible, e.g. consume for two cycles, produce for two cycles. If either $N \bmod i \neq 0$ or $N \bmod l \neq 0$ for any node in the graph, the MRRG must be further unrolled so that the pattern can be fully captured (here nodes representing the same resource in contexts j and k will share a configuration if $j \bmod N = k \bmod N$). For example, consider an architecture with 2 contexts and a node with latency 3. Such a node in context 0 must output in context 3 but no context 3 exists. To accommodate such a node the graph must be further unrolled to 6 virtual contexts.

C. Integer Linear Programming

The state-of-the-art *sound* and *complete* approach for mapping a DFG onto a MRRG is ILP. A linear programming problem involves linear constraints over real-valued variables. Integer linear programming further constrains the values to be integers and is known to be NP-hard [9]. An integer linear program in canonical form is written as follows:

$$\begin{aligned} \textit{maximize} \quad & c^T x \\ s.t. \quad & Ax \leq b \\ & x \geq \mathbf{0} \\ & x \in \mathbb{Z}, \end{aligned}$$

where A is a matrix and x, c, b, and $\mathbf{0}$ are vectors. $c^T x$ is referred to as the *objective* function and if replaced by a constant, this reduces to a *constraint* satisfaction problem over the integers.

One of the main algorithms for solving ILP problems is *branch and bound*, which enumerates integer solutions via a tree state-space search, while maintaining an estimated upper and lower bound on the optimal objective. The estimates are typically found by solving linear programs obtained by relaxing the integer requirements on the unassigned variables (a real relaxation). One benefit of this approach is that it can return a feasible solution as soon as one is found, even without proving optimality. We use this feature to find a *valid* but not necessarily optimal mapping.

D. Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is an extension of the Boolean satisfiability problem (SAT) to arbitrary first-order theories. It provides a richer interface for encoding problems and allows reasoning at a higher level of abstraction

when compared to SAT solvers. Decision problems in these theories are typically NP-complete (or worse), but modern SMT solvers are incredibly efficient and often avoid this worst-case behavior in practice.

Some examples of common background theories in SMT solvers are: fixed-width bitvectors, integers, uninterpreted functions, and reals. Each theory includes relevant sorts (i.e. types) and operations. In this paper, we exclusively use the theory of bitvectors, which supports both arithmetic and bit-wise constraints over vectors of bits.

Many SMT solvers also support *incremental* solving. This allows the user to repeatedly check a set of constraints, then add or delete constraints, then check again. Incremental solving can significantly improve performance on sequences of related queries. A more thorough introduction to SMT can be found in [10].

III. Encoding

We now describe the encoding of the CGRA mapping problem using SMT. We define our constraints assuming that no unrolling of the MRRG is needed beyond the number of contexts. However, our constraints can be generalized by asserting equality between the variables representing nodes that must share a configuration.

A. Definitions

We define our encoding over the DFG of the application to be mapped and the MRRG of the target architecture. We modify the base MRRG ($Nodes$, $Nets$) by inserting "port" nodes before every functional unit for each of its input ports. We also insert wire nodes between every mux to mux connection. The necessity of the extra nodes is explained in III-B8. We partition $Nodes$ into two sets: $FNodes$, which contains nodes representing function units; and $RNodes$ which includes all nodes used for routing (muxes, wire nodes, registers and ports).

We define the following functions on the graphs:

- $Support(n, d)$ returns true if a functional node n can support the operation d
- $Sinks(v) \subset Ops$ returns the set of sinks of a value
- $Src(v) \in Ops$ returns the source of a value
- $Operand(v, d)$ returns $k \in \mathbb{N}$, such that value v is the k^{th} operand to operation d
- $Ports(n, k) \in RNodes$ returns the k^{th} port node of functional node n
- $Inputs(n) \subset Nodes$ is the set of nodes which are inputs to node n
- $Outputs(n) \subset Nodes$ is the set of nodes which are driven by node n

Many of the constraints are asserted over the $Popcount$ of some set of variables V (the number of variables set to true, also called hamming weight). We encode $Popcount_{x \in V}(x) \leq 1$ as $b \;\&\; (b-1) = 0$ where b is the bitvector consisting of the concatenation of all variables $x \in V$. $Popcount_{x \in V}(x) = 1$ is encoded as $b \;\&\; (b-1) = 0 \wedge b \neq 0$. This is a common bithack for checking if a bitvector is a power of 2. For the general case, we use a divide and conquer encoding as described in [11].

B. SMT Constraints

We define 3 sets of variables, F, R, and S. F defines the mapping of operations to functional nodes. R and S define the mapping of values to routing resources.

- $F_{n,d}$ $\quad n \in FNodes,\ d \in Ops$
  n is used for performing d.
- $R_{n,v}$ $\quad n \in Nodes,\ v \in Vals$
  n is used for routing v.
- $S_{n,v,d}$ $\quad n \in Nodes,\ v \in Vals,\ d \in Sinks(v)$
  n is used for routing v to its sink d. The necessity of S is explained in III-C3

First, we constrain the mapping operations to functional nodes by asserting:

1) Operation Placement

All operations are performed exactly once:

$$\forall d \in Ops : \underset{n \in FNodes}{Popcount}(F_{n,d}) = 1$$

2) Functional Unit Exclusivity

All functional nodes are used no more than once:

$$\forall n \in FNodes : \underset{d \in Ops}{Popcount}(F_{n,d}) \leq 1$$

3) Functional Unit Legality

Operations are placed in functional units which support them. For example, IOs are mapped to IO units, adds are mapped to PEs, etc.:

$$\forall n \in FNodes,\ d \in Ops : F_{n,d} \implies Support(n, d)$$

Next, we constrain mapping of values to nodes by asserting:

4) Route Exclusivity

All nodes are used for routing no more than one value:

$$\forall n \in Nodes : \underset{v \in Vals}{Popcount}(R_{n,v}) \leq 1$$

5) Routing Resource Usage

When a node is used to route a value to a downstream operation, it must be used for routing a value and vice versa. In other words, $R_{n,v}$ and $S_{n,v,d}$ are consistent:

$$\forall n \in Nodes,\ v \in Vals : \left(\bigvee_{d \in Sinks(v)} S_{n,v,d} \right) \iff R_{n,v}$$

We now ensure that values originate from the correct functional node and terminate at the correct port.

6) Value Initialization

Values are routed from the functional node where their source is mapped:

$$\forall n \in FNodes,\ v \in Vals:$$
$$F_{n,Src(v)} \iff R_{n,v}$$

7) Value Termination

Values are routed to the correct input port of the functional node where their destination is mapped:

$$\forall n \in FNodes,\ v \in Vals,\ d \in Sinks(v):$$
$$\text{Let } k = Operand(v,d)$$
$$\text{Let } p = Ports(n,k)$$
$$F_{n,d} \iff S_{p,v,d}$$

Finally, we assert that the values are connected.

8) Input connectivity

If a node is used to route a value to a sink, exactly one of the inputs must also be used to route a value to a sink. We exclude functional nodes from this constraint as they generate values.

$$\forall n \in RNodes,\ v \in Vals,\ d \in Sinks(v):$$
$$S_{n,v,d} \implies \underset{n' \in Inputs(n)}{Popcount}\ (S_{n',v,d}) = 1$$

Here, we see the necessity of the wire nodes, as it would be impossible to route through a set of muxes that form a cycle. For example, in fig. 4b, routing from **IO A** to **M0** to **M3** would violate this constraint without the wire nodes (green circles), as **M0** would have two of its inputs routing the value. Intuitively, it may seem that this constraint is stronger than necessary and that asserting that *at least* one instead of *exactly* one of the inputs also route the value would be sufficient and remove the need for wire nodes. However, this would allow for self-reinforcing loops (see III-C2).

9) Output connectivity

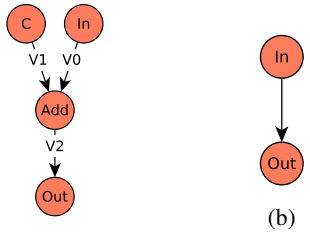
Similarly, if a node is used to route a value to a sink, exactly one of its outputs is used for that value. We exclude ports from this constraint as values terminate at ports.

$$\forall n \in Nodes - Ports,\ v \in Vals,\ d \in Sink(v):$$
$$S_{n,v,d} \implies \underset{n' \in Outputs(n)}{Popcount}\ (S_{n',v,d}) = 1$$

Again, this constraint may seem stronger than necessary but is explained in III-C3.

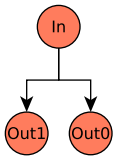
C. Examples

While the placement constraints are fairly intuitive, the routing constraints are less so. Here, we provide examples of how the constraints lead to a correct placement and routing. In the first example, we demonstrate the constraints in the general case. In the second, we justify III-B8 (*Input connectivity*). Finally, in the third example we justify III-B9 (*Output connectivity*) and show why breaking $R_{n,v}$ into $S_{n,v,d}$ is necessary.



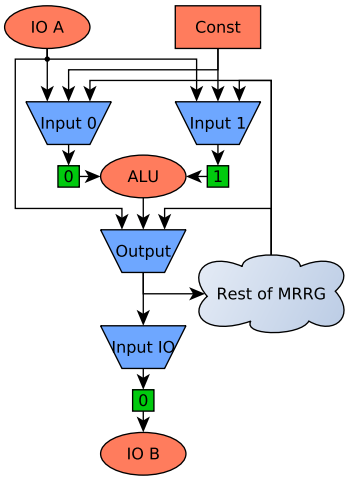
(a)

(b)



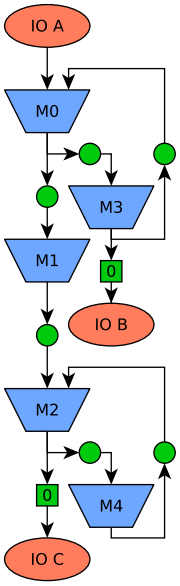
(c)

Fig. 3: DFGs



(a)

Note: wire nodes not shown



(b)

Fig. 4: MRRGs

1) Example 1

Let us consider the application **Out = In + Constant C** (fig. 3a) being mapped to the MRRG in fig. 4a. By III-B3 (*Functional Unit Legality*), **add** must be mapped to **ALU** and **C** to **Const**. Now, by III-B7 (*Value Termination*), we know that **V0** must be routed by **port 0** of the **ALU**, and similarly **V1** must be on **port 1** (the green squares). Hence, by III-B8 (*Input connectivity*), the muxes **input 0** and **input 1** must also route **V0** and **V1**, respectively. Now, as **Const** routes **V1** by III-B6 (*Value Initialization*), routing **V1** on **input 1** is consistent with III-B9 (*Output connectivity*). Similarly, **V0** may be routed from **IO A** (or some place not shown), and hence it is consistent for **In** to be mapped to **IO A**. Finally, we see that **V2** must be routed from **ALU** and therefore is routed by **Output** and then **port 0** of **IO B**, which forces **Out** to be mapped to **IO B**.

2) Example 2

Here, we demonstrate why it is necessary to assert *exactly* one instead of *at least* one of the inputs also routes the value in III-B8 (*Input connectivity*). Consider the DFG pictured in fig. 3b being mapped onto fig. 4b. Suppose we have the following scenario: **In** mapped to **IO A** and **Out** to **IO C**. It is clear that **M0**, **M1**, and **M2**, should be used to route the value between **IO A** and **port 0** of **IO C**. However, if we relax III-B8, there is a routing which is consistent with the relaxed constraint but does not use **M1** at all and thus results in an invalid mapping. The pair of loops, **M0 / M3** and **M2 / M4** would satisfy the connectivity constraints, despite the value never actually reaching its destination. By asserting that *exactly* one input also routes the value, we prevent such loops as **IO A** or **M3** may drive **M0** but not both.

3) Example 3

Consider the DFG in fig. 3c being mapped to fig. 4b. As in the previous example, the correct mapping should be intuitively obvious. **In** should be mapped to **IO A**, **Out0** to **IO B**, and **Out1** to **IO C**, with the value being routed using **M0 / M3** to **IO B** and **M0 / M1 / M2** routing to **IO C**. Now, suppose that instead of enforcing III-B9 (*Output connectivity*) on $S_{n,v,d}$, we enforced it on $R_{n,v}$. Then, the constraints would be unsatisfiable, as there is no way to fanout and have exactly one of the outputs route the signal. It again might seem reasonable to weaken this constraint and require *at least* one output route the value. While this would allow for fanout, it would once again lead to self-reinforcing loops as in III-C2. Consider a mapping where **M0 / M3** are used to route the value from **IO A** to **IO B**, and the loop **M2 / M4** to route the value to **IO C**. Here again, **M2** would reinforce itself. The use of $S_{n,v,d}$ prevents such loops because the signal originates at **IO A** and by III-B9 must be connected all the way to its destination **IO C**.

D. Functional Duplication

For some operations, it may be less expensive to duplicate the operations than to route the fanout (e.g. constants). In fact, depending on the routing topology, it may not even be possible to route high-fanout nodes [6]. This could be addressed with a prepossessing step on the DFG. While it may be possible to arbitrarily decide which nodes should be duplicated, in general such decisions can affect whether or not a design is mappable and the quality of results. Therefore, we allow nodes to be marked as candidates for duplication, and let the solver determine which to duplicate. To allow duplication we modify III-B1 (*Operation Placement*).

$$\forall d \in Ops: \quad \begin{cases} \bigvee_{n \in FNodes} F_{n,d} & \text{if } duplicate(d) \\ Popcount_{n \in FNodes}(F_{n,d}) = 1 & \text{otherwise} \end{cases}$$

E. Optimization

So far, we have focused on finding a valid but not necessarily *optimal* mapping. However, both ILP and SMT can also be used to find an optimal mapping. Our objective is to minimize routing resource usage. ILP solvers have built-in support for this and CGRA-ME's ILP encoding can minimize the sum of the routing resource indicator variables. Note, the canonical ILP form uses maximization, but we can simply negate the sum.

To optimize using our SMT-based technique, we use the SMT solver in incremental mode and perform a binary search over possible optimization bounds. Intuitively, we first find a valid mapping and count the number of routing resources used. This gives us an upper bound on the optimal value. Next we calculate a lower bound as a placement in which each operation is placed in the nearest possible functional unit to its inputs and outputs (this is generally not feasible). We then add constraints restricting the number of routing resources used to be less than half-way between the bounds and solve again. An unsat results gives a new lower bound, while a sat results provides new upper bound. We continue this process by iteratively refining the bounds using binary search until we find the point at which the problem becomes unsatisfiable for any bound below our current best bound.

Aditionally, instead of stopping only for a provably optimal result, we can provide a $cutoff$ which terminates the loop when our current bound is within some percentage tolerance of the lower bound. Algorithm. 1 gives this procedure as pseudocode. For the remainder of this paper, we exclusively use $cutoff = 0$ when optimizing, i.e. we look only for a completely optimal result.

```
solver.assert(base_constraints)
if solver.check_sat():
  best_model = solver.get_model()
  upper = Popcount(R_{n,v})
  lower = calculate_lower_bound()
  while (upper - lower) / upper > cutoff:
    pivot = (lower + upper) / 2
    solver.assert(lower ≤ Popcount(R_{n,v}) ≤ pivot)
    if solver.check_sat():
      best_model = solver.get_model()
      upper = Popcount(R_{n,v})
    else:
      solver.pop_assertion()
      lower = pivot + 1
  return best_model
else:
  return INFEASIBLE
```

Algorithm 1: Optimization using incremental solving

IV. Related Work

There have been many approaches to CGRA mapping. The traditional techniques are node-centric, where nodes are placed first, then routed. One of the most popular node-centric techniques for both FPGAs and CGRAs is simulated annealing [12], [13]. Other approaches include particle

swarm optimization [14] and force-directed placement [15]. In [5], the authors observed that node-centric techniques are not amenable to CGRA architectures and instead developed an edge-centric approach, which considers routing from the beginning and obtains a placement as a by-product. Routing-focused techniques are particularly important for restricted routing architectures. Some recent edge-centric approaches are a graph minor approach [16] and an Integer Linear Programming (ILP) encoding [7] focused on architectures with severely restricted routing resources. [6] mitigates resource limitation issues by allowing re-computations. We employ a similar functional duplication strategy in our encoding for high-fanout, low-cost operations.

SAT solvers have been successfully applied for routing [17] and even (edge-centric) simulataneous place-and-route [18] in ASICs. [19] proposes an SMT-based constraint solving technique for mapping modulo scheduling in CGRAs. While our approach is also SMT-based, we use an edge-centric bitvector encoding for restricted routing architectures, while [19] uses a node-centric integer encoding targeting more traditional routing infrastructures.

V. Experimental Results

A. Methodology

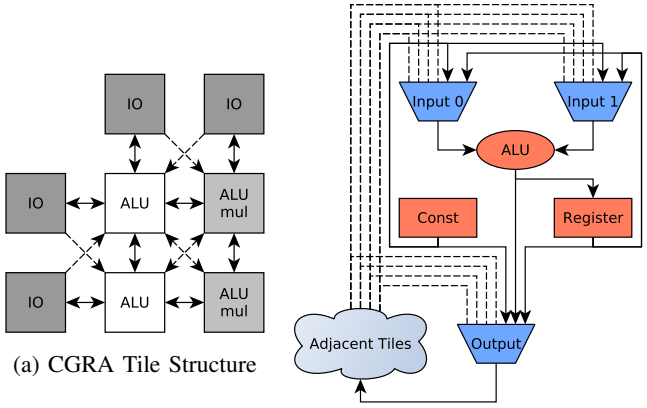
We compare the performance of the SMT encoding, with various duplication strategies, to CGRA-ME's ILP encoding on 25 designs. We included 15 of the 17 available benchmarks from CGRA-ME, excluding the two smallest because they are trivial: (*2x2-f* and *2x2-p*).[2] We also added 10 handcrafted benchmarks performing various linear algebra operations. The number of nodes in these benchmarks ranges from 12 to 35, and the maximum fanout ranges from 1 (no fanout) to 15.

Our tests attempt to map each design to one of 16 fabrics obtained by generating all combinations of these fabric parameters:

1) *size*: 4x4 and 8x8 grids of function units plus IOs
2) *context*: 1 or 2 contexts
3) *interconnect*: orthogonal only or orthogonal and diagonal wiring
4) *resource*: homogeneous resources or heterogeneous resources where only 50% of the units support multiplication.

Fig. 5 depicts the architectures. This is a superset of the settings explored by CGRA-ME which explored all combinations of *interconnect*, *context* and *resource*. Additionally, to match CGRA-ME we assume all functional units have a latency and initiation interval of 0.

CGRA-ME supports two ILP solvers, a commercial solver, Gurobi [20], and a non-commercial solver, SCIP [21]. We compare to both solvers and note that Gurobi is expected to be about an order of magnitude faster. For our SMT-based approach, we use the open-source SMT solver Boolector [22].[3] Boolector can be used with different back-end SAT solvers. For non-optimizing runs, we use the CaDiCaL SAT solver, and for optimizing runs, we use Lingeling [24]. This is because CaDiCaL is faster overall, but it does not support incremental solving, which is useful for optimizing.



(a) CGRA Tile Structure

(b) ALU Tile

Fig. 5: Architecture

Within this section, we use the following names to denote the solving approach:

1) *SMT*: SMT solving with no functional duplication
2) *SMT+DC*: SMT solving where constants can be duplicated
3) *SMT+DR*: SMT solving where everything but multipliers and IO are allowed to be duplicated
4) *SMT+DA*: SMT solving where any node can be duplicated
5) *ILP-GRB*: ILP with Gurobi
6) *ILP-SCIP* ILP with SCIP

All experiments were run on a 3.5GHz Intel Xeon E5-2637 v4 with 16GB of memory and a two hour timeout per run.

B. Valid Mapping Results

We first focus on non-optimizing runs where we are only looking for a *valid* mapping. We found that while ILP solves some instances very quickly, SMT scales better over time and continues to solve new instances while ILP starts to plateau. Fig. 6 shows the cumulative solving time against the number of benchmarks solved. For each technique, benchmarks are sorted in increasing order by the time to solve. It is clear that the SMT-based approaches hit the exponential knee much later than the ILP encoding. It is also interesting to note that the more functional duplication is included, the better SMT scales. Furthermore, notice that timeouts count as two hours in the cumulative time, but in reality the time to solve could be much greater so this plot actually underapproximates the improvement over ILP encodings.

[2]The CGRA-ME authors provided access to all of their original benchmarks except *accum* and *mac*.

[3]We tried other SMT solvers but found that Boolector was consistently the most performant for this application. This is not surprising as Boolector is a consistent winner of the bitvector division of the yearly SMT competition [23].

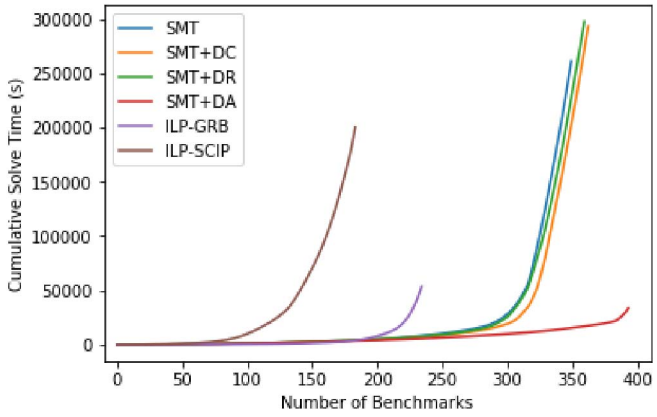


Fig. 6: Number of benchmarks vs. cumulative solve time for each approach (non-optimizing).

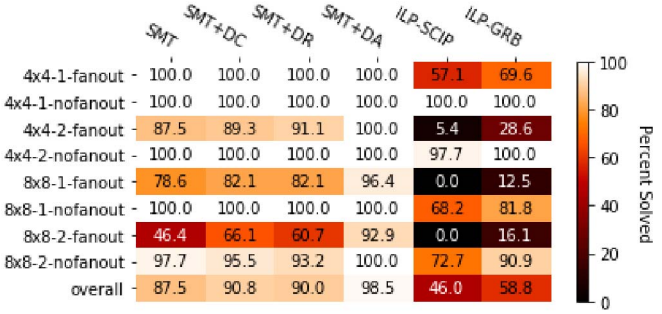


Fig. 7: Percentage of benchmarks that were solved within the timeout (non-optimizing). There are 44 benchmarks without fanout and 56 with fanout.

The SMT encoding solves a strict superset of the instances that the ILP encoding solves, and our results are consistent with CGRA-ME on all commonly solved instances. However, there are some benchmarks which were infeasible, but become feasible when the solver is allowed to duplicate units.

Fig 7 shows the percentage of instances that were solved for each technique on benchmarks which have been divided according to fabric and design parameters, and Fig 8 shows the median runtimes. We observed that fabric *resource* had little to no effect on solving time. The fabric *interconnect* did have a slight impact on performance: fabrics supporting diagonal connections were easier to map to than those with only orthogonal connections, but the number of solved instances increased by about 10% regardless of method. Hence, we do not divide our results along those two parameters. In addition to splitting along fabric parameters *size* and *context*, we also distinguish between benchmarks where the design to be mapped has *any* fanout and those that do not. We use the following naming scheme for benchmark groups: `<size>-<contexts>-<fanout>`.

While all techniques slow as the size of the architecture increases, the SMT encodings scale significantly better than ILP. In particular, *SMT+DA* is able to solve almost all the instances even in the hardest category. Additionally, notice that

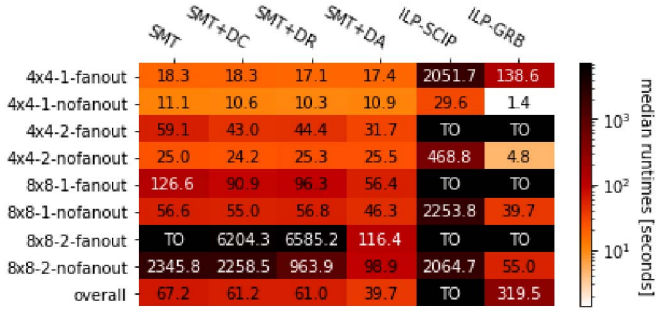


Fig. 8: Median runtime (non-optimizing).

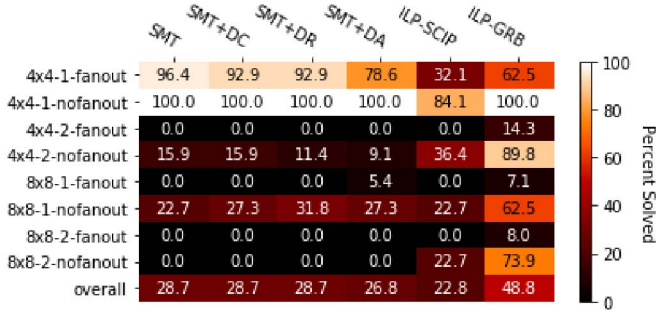


Fig. 9: Percentage of benchmarks for which an optimal solution is found within the timeout.

ILP encodings perform quite well on designs without fanout, but *any* fanout in the design significantly hurts performance. SMT on the other hand continues to perform well even on designs with fanout.

C. Optimizing Results

As mentioned above, our technique targets a design exploration development flow, for which optimal solutions are overkill. However, for completeness of the evaluation, we also evaluated the techniques for optimizing runs (using the optimizing algorithm described in Section III-E). Fig. 9 shows that, as expected, the ILP techniques perform much better when optimizing. However, among solutions based on open-source (non-commercial) solvers, the SMT techniques are comparable, though clearly they struggle when there is more than one context.

D. Discussion and Future Work

For non-optimizing runs, *SMT+DA* is by far the best and most scalable solution. Thus, this technique fills an important niche for workflows that prioritize design iteration time. The SMT-based techniques are especially good for designs without fanout. One reason for this is that without fanout, $S_{n,v,d}$ is the same as $R_{n,v}$, which drastically simplifies the encoding. Better encodings for designs with fanout is an interesting avenue for future work.

We observe that SMT finds mappings quickly, but struggles to optimize them and conversely, ILP struggles to find mappings but can optimize the ones it does find. Another potential avenue for future work would be to use SMT to find an initial

mapping and then use ILP to optimize it.

VI. Conclusion

We have presented an SMT-based approach for mapping to arbitrary CGRAs, which rapidly determines the feasibility of mapping applications. This approach is at least an order of magnitude faster than state of the art approaches using ILP. Further, we introduced functional duplication, which increased the number of solvable instances. This tool will allow architects to rapidly explore novel architectures and routing topologies without needing to develop custom algorithms to perform mapping.

Acknowledgments

This work was partially supported by a National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by a gift from Intel Corporation to the Stanford Agile Hardware Center and by the DARPA DSSoC program, grant #FA8650-18-2-7861.

References

[1] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov 2010.

[2] U. Farooq, Z. Marrakchi, and H. Mehrez, *FPGA Architectures: An Overview*. New York, NY: Springer New York, 2012, pp. 7–48. [Online]. Available: https://doi.org/10.1007/978-1-4614-3594-5_2

[3] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, "Evaluating programmable architectures for imaging and vision applications," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[4] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," ser. ISCA 2017. IEEE, 2017, pp. 389–402.

[5] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," ser. PACT 2008, Oct 2008, pp. 166–176.

[6] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1284–1291. [Online]. Available: http://doi.acm.org.stanford.idm.oclc.org/10.1145/2228360.2228600

[7] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," ser. DAC '18. New York, NY, USA: ACM, 2018, pp. 128:1–128:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3195986

[8] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 166–173.

[9] C. H. Papadimitriou, "On the complexity of integer programming," *J. ACM*, vol. 28, no. 4, pp. 765–768, Oct. 1981. [Online]. Available: http://doi.acm.org.stanford.idm.oclc.org/10.1145/322276.322287

[10] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, E. Clarke, T. Henzinger, and H. Veith, Eds., 2017, in preparation. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BT14.pdf

[11] E. M. Reingold, J. Nievergelt, and N. Deo, "Combinatorial algorithms: Theory and practice," 1977.

[12] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 5, pp. 255–, Sept 2003.

[13] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "Spr: An architecture-adaptive cgra mapping tool," ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 191–200. [Online]. Available: http://doi.acm.org.stanford.idm.oclc.org/10.1145/1508128.1508158

[14] R. Gnanaolivu, T. S. Norvell, and R. Venkatesan, "Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization," in *2010 International Conference of Soft Computing and Pattern Recognition*, Dec 2010, pp. 145–151.

[15] A. Fell, Z. E. Rkossy, and A. Chattopadhyay, "Force-directed scheduling for data flow graph mapping on coarse-grained reconfigurable architectures," ser. ReConFig '14, Dec 2014, pp. 1–8.

[16] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 285–292.

[17] N. Ryzhenko and S. Burns, "Standard cell routing via boolean satisfiability," 06 2012.

[18] A. Nadel, "A correct-by-decision solution for simultaneous place and route," ser. CAV 2017, 2017, pp. 436–452. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_23

[19] K. Fan, H. h. Park, M. Kudlur, and S. o. Mahlke, "Modulo scheduling for highly customized datapaths to increase hardware reusability," ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 124–133. [Online]. Available: http://doi.acm.org.stanford.idm.oclc.org/10.1145/1356058.1356075

[20] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2018. [Online]. Available: http://www.gurobi.com

[21] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 6.0," Optimization Online, Technical Report, July 2018. [Online]. Available: http://www.optimization-online.org/DB_HTML/2018/07/6692.html

[22] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0." in *CAV (1)*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595. [Online]. Available: http://dblp.uni-trier.de/db/conf/cav/cav2018-1.html#NiemetzPWB18

[23] 13th International Satisfiability Modulo Theories Competition, "Smtcomp 2018 qf_bv results," www.smtcomp.org, 2018, accessed: 2019-03-28.

[24] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2017-1. University of Helsinki, 2017, pp. 14–15.