This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Iterative Search for Reconfigurable Accelerator Blocks with a Compiler in the Loop

Max Willsey, Vincent T. Lee, *Student Member, IEEE,* Alvin Cheung, Rastislav Bodík, and Luis Ceze, *Senior Member, IEEE,*

***Abstract*—Domain-specific reconfigurable accelerators (DSRAs) achieve high performance and energy efficiency by using specialized processing elements (PEs) instead of general-purpose alternatives. However, the process of designing, selecting, and refining the reconfigurable PEs that compose the accelerator fabric has remained a manual and difficult task. This paper presents Reconfigurable Accelerator Design using Iterative Search for Hardware (RADISH) which is a full-stack framework for automatically identifying and generating PEs from an application corpus. RADISH uses a genetic algorithm to iteratively search for and refine the proposed PEs with a compiler-in-the-loop to guide the search. We show that RADISH-generated PEs can generalize to both larger instances of the same application as well as other previously unseen applications within the same domain. We evaluate a CGRA architecture using our RADISH-generated PEs and show it achieves a geometric mean improvement of up to 2.14× and 2.4× power and area respectively over an ALU-based CGRA designs. In terms of energy, our generated designs achieve a geometric mean improvement of 2.5× but can achieve gains up to 28.9×.**

I. INTRODUCTION

From embedded systems to data centers, the performance and energy gap has accelerated the push towards increasingly specialized accelerators [1], [2]. These systems employ many accelerators to meet the needs of increasingly compute intensive and memory demanding applications. However, the non-recurring engineering costs of ASIC fabrication and FPGA development remain stubbornly high, and overspecialization runs the risk of quickly becoming irrelevant in the face of a rapidly changing application landscape.

Domain-specific reconfigurable accelerators (DSRAs) combine the benefits of hardware specialization with the flexibility needed to implement different applications within a domain [3], [4], [5]. But despite tremendous advances in hardware implementation flows such as Vivado HLS [6], OpenCL-to-gates [7], and C-to-gates [8], designing a DSRA remains a tremendous effort, because it requires expertise from both application domain experts and hardware engineers. Critically, a domain expert must identify common functionality across applications worth accelerating. The hardware engineer must then implement processing elements (PEs) with that functionality, perhaps using the tools mentioned above, and report back to the domain expert if the functionality prevented an efficient implementation. After sufficient iterations, this yields an efficient DSRA at the cost of significant time and effort.

In this paper, we present our application-driven tool Reconfigurable Accelerator Design using Iterative Search for Hardware (RADISH) which automatically identifies and generates PEs for a DSRA. RADISH uses a genetic algorithm to generate reconfigurable PEs from a given set of domain applications. The algorithm combines parts of the application using datapath fusion and then evaluates the resulting PEs by compiling them onto the applications. By incorporating compilation during PE generation, RADISH uses metrics such as coverage, utilization, and communication reduction to iteratively improve the PEs.

We evaluate the RADISH-generated PEs for two application domains: image processing and linear algebra. We compose the PEs into a CGRA and show our design achieves a geometric average improvement of 2.14× and 2.4× in power and area respectively compared to ALU-based CGRA accelerators. We also show that the reconfigurable PEs produced by our tool are able to generalize to support larger application instances (i.e. larger input image or more iterations), and also other applications within the same domain.

Our paper makes the following contributions: (1) We define a novel reconfigurable PE generation technique using a genetic algorithm formulation for graph partitioning which iteratively identifies common functionality across applications. (2) We introduce a compiler-in-the-loop to the genetic algorithm which exposes metrics such as internal PE utilization and application coverage when evaluating the fitness of generated PEs. (3) We implement the technique in the RADISH framework, and evaluate the quality of generated PEs using real-world applications and demonstrate results that are competitive with CGRA-based designs using arithmetic logic units (ALUs).

The rest of the paper is organized as follows. The next section introduces background on genetic algorithms and defines the DSRA design problem. Section III introduces the RADISH tool flow. Section IV evaluates the search procedure and the generality of generated PEs, and Section V compares our generated accelerator against CGRA designs. Finally, we highlight related work in Section VI.

II. BACKGROUND

A. Domain-Specific Reconfigurable Accelerators

The accelerator design spectrum ranges from fully-specialized to general-purpose hardware. Fixed-function ASICs achieve maximal efficiency, while FPGAs trade some performance for flexibility provided by full reconfigurability [9]. More recently, there has been a trend towards domain-specific reconfigurable accelerators (DSRAs) such as Q100 [3], Graphicianado [4], and DianNao [5]. These solutions offer a middle ground between FPGAs and fixed-function ASICs by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

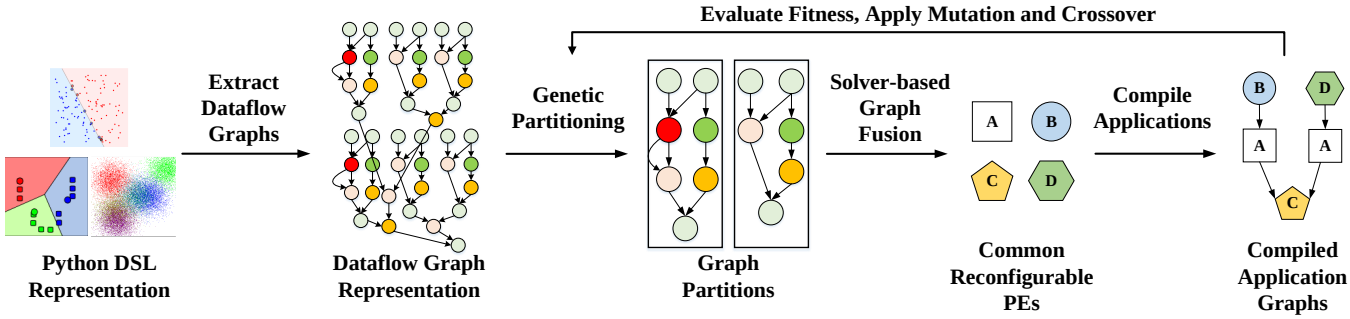


Fig. 1: RADISH flow for generating domain-specific reconfigurable accelerator PEs. Domain applications are converted into dataflow graphs, partitioned, and then combined into PEs. The PEs are then compiled back onto the applications. A genetic algorithm iteratively performs this process, refining the PEs at each step.

specializing to a domain. However, as mentioned in Section I, designing such accelerators remains largely a manual task for the experts, with few tools that support design automation.

B. Coarse-Grain Reconfigurable Arrays

Coarse-grain reconfigurable arrays (CGRAs) [10], [11] represent another design point between application-specific accelerators and general-purpose architectures. On the specialization spectrum, CGRAs are more specialized than FPGAs since they use coarse-grained or larger compute units. At a high level, CGRAs are an array of compute units or *processing elements* (PEs) connected together by an interconnect. Processing elements can take any number of forms from fine-grained single arithmetic operators, to larger ALU units, to entire application-specific components like sorting units. Identifying the set of PEs that represent common operations within an application domain is typically done manually.

An example of such an architecture is shown in Fig. 2, where the PEs are single or compound ALUs. This particular CGRA architecture will serve as our ALU-based CGRA comparison in our later evaluations. ALUs are a natural choice of PE because they are general purpose and have sufficiently fine granularity to cover all applications. However, because of their generality, the arithmetic units within an ALU are underutilized and present opportunities for optimization. For instance, a CGRA for linear algebra applications might remove bitwise operations from the ALUs within the PEs, as such applications tend not to use them.

In this paper, we target CGRAs but our techniques should also apply to other architectures. As shown in Fig. 2, the DSRA architecture model we use to evaluate RADISH consists of a mix of compute PEs connected together with a grid routing interconnect. Our goal is to automatically identify the PEs that form the replicated units in the resulting CGRA.

C. Genetic Algorithms

Genetic algorithms [12] are commonly used for solving search or optimization problems, including graph partitioning [13]. The core components of genetic algorithms are inspired by biological evolution: mutation, crossover, and survival of the fittest.

Genetic algorithms typically start by randomly initializing sets of parameters where each parameter set (called a *genotype*) defines a candidate solution (called a *phenotype* or *individual*). On each iteration, *mutation* randomly changes some part of the genotype, and *crossover* randomly combines two genotypes into a new one. From the genotypes, the algorithm constructs the candidate solutions and evaluates them according to some fitness function. Unfit individuals are eliminated, improving the quality of the population with each iteration.

/

III. FRAMEWORK AND ENCODING FORMULATION

We now introduce our tool, RADISH, for designing DSRAs. Figure 1 shows an overview of RADISH. As inputs, RADISH takes applications written in a custom Python framework (Section III-A) that facilitates dataflow graph extraction. These dataflow graphs are then partitioned (Section III-B), and the resulting partitions are fused together (Section III-C) into reconfigurable PEs. RADISH comes with a compiler that maps the PEs back on to the dataflow graphs to evaluate the PEs' effectiveness in the domain (Section III-D). The compilation informs a genetic algorithm that re-partitions the application dataflow graphs. This cycle iteratively improves the generated PEs according to user-defined cost metrics. Finally, the RADISH returns the PEs, optionally generating synthesizable Verilog (Section III-E). The synthesized PEs can then be composed into a DSRA.

A. Application Representation

RADISH takes as input dataflow graphs that represent program executions. Nodes are the operations performed and edges represent data dependencies. To generate such inputs,

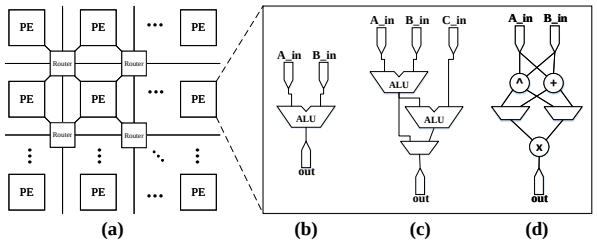
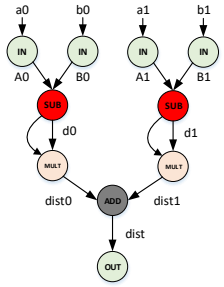


Fig. 2: (a) CGRA architectural model using (b) ALUs, (c) compound ALUs, or (d) RADISH-generated PEs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

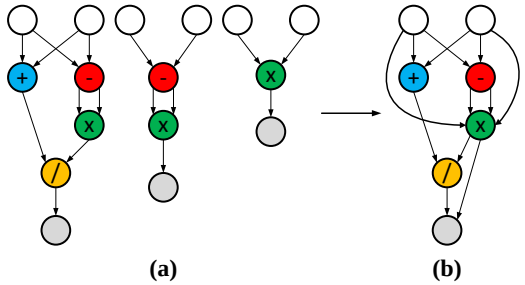
```
# a simple L2 distance calculation
def l2_distance(a0, a1, b0, b1):

    # convert values to DSL values
    A0, A1 = Node(a0, INPUT), Node(a1, INPUT)
    B0, B1 = Node(b0, INPUT), Node(b1, INPUT)

    # L2 distance
    d0, d1 = a0 - b0, a1 - b1
    dist0, dist1 = d0 * d0, d1 * d1

    # add reduction
    dist = dist0 + dist1

    # generate the dynamic dataflow graph
    dataflow_graph = dist.graph
```

(a) DSL code



(b) Dataflow graph

Fig. 3: Application DSL code and associated dataflow graph for L2 distance.



(a) (b)

Fig. 4: Fusion combines graphs to share compute elements while introducing reconfigurability. (a) Chi-squared distance, Euclidean distance, and multiplier input graphs. (b) Resulting fused graph.

we use a custom Python framework to run domain applications with specially wrapped inputs, and we dynamically construct dataflow graphs via operator overloading.

Fig. 3b shows an example of our Python tool to extract dataflow graphs. Porting existing Python programs to this framework requires minimal changes. The user can also introduce custom operations to the dataflow graph. Python loops, conditionals, and functions are all supported, as the `Node` object simply wraps a value to keep track of the dataflow graph. Unlike a compiler static analysis that works on a control flow graph of basic blocks, this approach exposes data dependencies across basic block and memory access boundaries. RADISH can therefore generate reconfigurable PEs that capture more behavior.

However, this dynamic approach introduces redundancy: parts of the program executed many times are represented many times in the dataflow graph. The PE fusion stage (to be discussed in Section III-C) has to cope with this duplication by finding similar or identical parts of the graphs and recombining them. Using the dataflow from a particular run of a program also raises the question of generality; our evaluation in Section IV-C addresses this.

B. Genetic Algorithm and Partitioning

Our main contribution is a formulation of the DSRA design problem that is amenable to automated search. Genetic algorithms are frequently used to search large solution spaces, and we adapted these techniques for use in RADISH.

Inputs to the tool are represented as dynamic dataflow graphs. We partition each graph into smaller program subgraphs. These partitions are then deterministically fused to form reconfigurable PEs (the phenotype). The behavior of the phenotype (the evaluated reconfigurable PEs) is entirely determined by the genotype (the partitioning).

For both the initial partitioning and subsequent partitions performed during mutation, we use the Kernighan-Lin partitioning algorithm [14] to achieve low communication cost between partitions. The algorithm is seeded by random assignment of nodes to partitions; it is therefore non-deterministic and suitable for random initialization and mutation.

1) Initial Partitioning: : Given the application dataflow graphs, RADISH iteratively partitions each application until all partitions are smaller than a user specified size. The initial partition size only determines the maximum size of the *initial* partitioning of each application. During the course of the genetic search, mutations will combine and split these partitions arbitrarily to optimize the given cost function.

As mentioned above, dynamic dataflow graphs likely contain duplication. Ideally, the application graphs would be partitioned in a way that isolates duplicate subgraphs, so the fusion can easily identify their similar structure. However, the random initial partitioning is very unlikely to partition the input dataflow graph in such a manner. One approach is to rely on mutations made during genetic search to eventually converge to the right partitions, but this can be extremely time consuming.

Instead, RADISH seeds the initial partitioning by pre-cutting edges outside of frequently used subgraphs. We use apriori graph mining [15] to find frequently occurring subgraphs in the application dataflow graph. We pre-cut the edges going in to or out of these frequent subgraphs, encouraging the partitioner to isolate them and not break them apart. This technique ensures that the initial partitioning highlights the common structure in the application, making it easy for the fuser to combine partitions into high-utilization reconfigurable PEs.

2) Mutation and Crossover: : Mutation and crossover allow genetic algorithms to explore different parts of the search space. For mutation, we simply change the partitioning (the genotype) of the dataflow graph by randomly combining or breaking apart partitions. We implement crossover between two search instances by choosing some application and swapping the way the different instances partition that application.

C. Reconfigurable PE Fusion

Once the applications are partitioned, fusion combines similar dataflow subgraphs into *processing elements* that can be reconfigured to behave as any of their component subgraphs. Fig. 4 shows an example of fusing three small subgraphs into a reconfigurable PE. Fusion inserts multiplexors when necessary to allow the resulting PE to implement the functionality of the input subgraphs; the multiplier in Fig. 4b requires multiplexors to choose 2 inputs from the possible 4. These PEs are guaranteed by construction to implement at least the functionality of their component parts of the input programs, but they often support more. Section IV-C demonstrates that fusion produces reconfigurable PEs that support previously unseen functionality.

While other user-defined cost metrics guide the genetic partitioning, fusion aims to maximize resource sharing. Con-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

ceptually, our algorithm (shown in Algorithm 1) iteratively combines pairs of PEs that have the highest *compatibility*, a notion that we will define later. Only fusing the most compatible PEs ensures that the resulting reconfigurable PEs have high internal utilization: most configurations will use most of the PE's functionality. The iterative, n-way fusion algorithm in Algorithm 1 is based on a simpler algorithm responsible for fusing two PEs at a time, inspired by previous work [16], [17].

Both the binary and iterative, n-way fusion algorithms are deterministic. Thus, the output reconfigurable PEs are completely determined by the input partitioned applications. From the perspective of the genetic algorithm, PE fusion takes the genotype (partitions) and generates the phenotype (reconfigurable PEs).

1) Binary Fusion: : Prior work [16], [17] developed techniques to fuse two dataflow graphs and maximize resource sharing via reduction to integer linear programming (ILP). We use similar techniques to combine two reconfigurable PEs (also via ILP) in a way that maximizes sharing and preserves all the possible configurations of the input PEs.

The encoding creates a variable v_{v_1,v_2} for every node v_1 in the first input graph and v_2 in the second input graph, and variables e_{e_1,e_2} for edges e_1, e_2 from the first and second input. These variables indicate whether vertex v_1 will be combined with v_2; likewise for the edges. We constrain the edges to only combine if their endpoints combine. We further constrain the v variables to only combine alike operations (ex. adders should only combine with adders). Finally, we instruct the ILP solver to maximize the number of combined edges and vertices. We read out the variable assignments to construct a reconfigurable PE with maximal resource sharing.

At its core, binary PE fusion attempts to solve the minimum common supergraph problem which is NP-hard. While this means that fusing entire application dataflow graphs would take a significant amount of time, fusing pre-partitioned subgraphs is fast in practice, motivating our partitioning step (Section III-B).

After fusion, PEs may have more input edges than the operator arity. For instance, the two-input multiplier in Fig. 4b has four different possible inputs. To handle this, we introduce configuration signals and multiplexors which allows us to select between inputs, guaranteeing that the fused result can implement at least the functionality of the input PEs. This introduces the minimal amount of reconfigurability necessary to support the two input graphs.

The results of the PE fusions are used to measure how "compatible" the two input PEs are. We first define the *cost* of a PE, $|p|$. This is user-defined and should capture the desired cost metric which could include metrics such as estimated area of the PE or its critical path. For simplicity, we define cost as a linear combination of the number of operations and the number of internal edges.

We measure the compatibility of PEs p_1 and p_2 based on the best and worst case of fusion. In the best case, one PE is a subgraph of another, $|p_1 \cdot p_2| = \max(|p_1|, |p_2|)$ where "$\cdot$" denotes PE fusion. In the worst case, fusion finds no resource sharing at all, so $|p_1 \cdot p_2| = |p_1| + |p_2|$. So we define compatibility as an interpolation between these two cases, 1 being maximally compatible and 0 being not compatible at all:

Algorithm 1 Iterative Fusion

```
1:  procedure ITERATIVEFUSION(P_init)
2:      P_new ← P_init
3:      P_final ← ∅
4:      initialize empty G_compat = (V, E)
5:      loop
6:          for p1, p2 ∈ (P_new × P_new) ∪ (V × P_new) do
7:              V ← V ∪ {p1, p2}
8:              E ← E ∪ {(p1, p2, ESTCOMPAT(p1, p2))}
9:          end for
10:         m ← MAXWEIGHTMATCHING(G_compat)
11:         if m is empty then
12:             return P_final ∪ V
13:         end if
14:         P_new, P_old ← ∅, ∅
15:         for p1, p2 ∈ m do
16:             c, p_fused ← FUSE(p1, p2)
17:             if c < c_thres then
18:                 E ← E \ (p1, p2)
19:             else
20:                 P_new ← P_new ∪ p_fused
21:                 P_old ← P_old ∪ {p1, p2}
22:             end if
23:         end for
24:         P_isolate ← ISOLATES(G_compat)
25:         P_final ← P_final ∪ P_isolate
26:         V ← V \ (P_isolate ∪ P_old)
27:     end loop
28: end procedure
```

$$c(p_1, p_2) = 1 - \frac{|p_1 \cdot p_2| - \max(|p_1|, |p_2|)}{\min(|p_1|, |p_2|)}$$

2) Iterative Fusion: : We now present an algorithm that solves the n-way fusion problem: given a number of PEs, perform pairwise fusions such that overall resource sharing is maximized while maintaining high utilization. Binary fusion effectively combines two PEs in a way that maximizes resource sharing, and compatibility provides a way to measure that sharing. Naively fusing n PEs using this method will result in a single huge PE with very low utilization and resource sharing, simply because some parts of the input application are not similar. An n-way fusion algorithm needs to crucially decide what to fuse and when to stop in order to avoid poor results. We accomplish this by building a *compatibility graph* (defined below) that dictates what to fuse.

Our algorithm relies on a few subroutines. MAXWEIGHTMATCHING and ISOLATES are standard graph algorithms from the NetworkX library [18]. MAXWEIGHTMATCHING finds a set of edges with maximum weight that share no vertices. ISOLATES finds vertices without any edges. The FUSE subroutine is the binary fusion algorithm described above. ESTCOMPAT (estimate compatibility) provides conservative upper bound on the compatibility of two PEs without calling the relatively expensive ILP fusion algorithm.

The iterative algorithm centers around the *compatibility graph*, defined as $G_{\text{compat}} = (V, E)$ in the listing. The vertices V in this graph are PEs, and the edges E are weighted with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

the compatibility of their endpoints. The algorithm begins by taking the given PEs (P_{init}) and marking them as unprocessed by placing them in P_{new}. At the start of each iteration, we update the compatibility graph with unprocessed PEs in P_{new} by adding them as vertices. We add edges between the new PEs and themselves as well as between them and existing PEs. New edges are weighted with the estimated compatibility to reduce the number of calls to the ILP solver.

Once the compatibility graph is updated, we generate a maximum weight matching of non-overlapping pairs of PEs that should fuse well. The algorithm then performs these fusions; if the compatibility is greater than the threshold, we keep it, otherwise we delete that edge. For those that are kept, the component PEs are removed from the compatibility graph (via P_{old}), and the new PE is added in next iteration.

After each iteration, we remove any isolated nodes from the graph. These isolates have been determined to incompatible with anything else, so we move them into the set of PEs to be returned (P_{final}). The algorithm terminates when the matching is empty, indicating that there are no useful fusions to be done. We return the set of PEs removed as well as those remaining in the compatibility graph.

D. PE Compilation and Evaluation

Once the reconfigurable PEs, or the phenotype, have been generated, we must evaluate them. Our phenotype evaluation is an approximation of compiling programs onto a DSRA, representing how the PEs would perform when running applications from the given domain. Evaluation first compiles the given applications onto the generated PEs in a simplified architecture. From this compilation information, we gather metrics used to score the phenotype including coverage, utilization, and communication reduction.

One of the key contributions of RADISH is that we learn which PEs are effective based on a realistic model of compilation. Ideally, the compilation would reflect both the performance of the PEs in isolation and how their composition affects performance. However, a full compilation requires placing and routing the PEs both internally and externally, which would be prohibitively expensive for our genetic search. Instead, we make some simplifying assumptions about the system architecture to make compilation easier and faster.

Specifically, we assume that PEs are placed optimally within the fabric for a grid interconnect between PEs. We also assume that the resulting architecture has an "unlimited" number of each PE. In reality, if PEs are used more times in the program than they are present in architecture, we would break down the program and compile it in stages. This assumption allows our compiler to work with programs in their entirety, without worrying about whether or not it can fit onto the resulting architecture.

Finally, we assume that a dynamic trace is representative of program behavior. This is the same assumption used early in the RADISH pipeline to learn the PEs. We validate this assumption in Section IV-C by demonstrating that the PEs learned from one set of program traces can be used to compile different program traces.

With these assumptions, compilation becomes a graph matching problem: given a program and a set of reconfigurable PEs, find an assignment of program parts to PEs. This allows us to use a simple, efficient algorithm for compilation. Given an application dataflow graph and a PE, we try to map the outputs of a PE onto all nodes of the program. From each initial mapping, we use depth first search to match the PE from the outputs, and try to map each node of the PE onto a node of the program until we reach the inputs. We rank the compilations based on how many program nodes were compiled and how much of the PE was utilized. We greedily accept the best compilation, contract the compiled program nodes into a graph of PE nodes, and then repeat until the program graph consists of only PE nodes (i.e., the entire program is compiled).

1) Quality Metrics: The result of compilation is a set of program dataflow graphs where operations have been contracted into PE nodes. PE nodes store the operations in the program that were compiled and the PE that was used. Edges between PE nodes represent inter-PE communication that would have to take place over the router or network on the resulting hardware. Because of the flexibility of the genetic algorithm, users can define any metrics to evaluate the compiled graph.

Ideally, the search for reconfigurable PEs would be guided by a fitness function that measures the quality of a PE after logic synthesis, such as power consumption and silicon area. Unfortunately, logic synthesis is prohibitively slow to be invoked on each candidate PE since compiling our PE to application dataflow graphs is orders of magnitude faster than hardware logic synthesis. We therefore rely on proxy metrics that can be obtained with our compiler.

We choose three metrics: coverage, utilization, and communication reduction. Higher is better for all three metrics. The genetic search simultaneously optimizes all of the metrics for each application, producing a Pareto frontier of solutions (sets of PEs) that perform well. Section IV evaluates whether the search can effectively optimize these metrics, and Section V evaluates how they correlate with hardware metrics like power and area.

2) Coverage: Coverage measures how much of an application graph was successfully compiled, or *covered*, by the given set of PEs. We define coverage as the fraction of operations in an application graph that were covered. Coverage is a metric of functional generality; an ideal set of PEs will have a coverage of 1 across all applications. If an application has coverage below 1, the PEs are not general enough to implement the desired functionality, so some general purpose computation such as arithmetic logic units (ALUs) or lookup tables (LUTs) are required.

Note that our algorithm will almost always produce PEs with perfect coverage on the training applications, because the partitions of those application seeded the PE fusion. Perfect coverage is not guaranteed, however; the greedy compilation could possibly make a suboptimal choice that prevents perfect coverage.

3) Utilization: PE *utilization* measures how much of the reconfigurable PEs are active, as they may have more compute nodes than can be activated in any given configuration. We define a compiled PE's utilization as the fraction of used

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

operations in that compiled instance. The utilization of an application graph is then defined as the average utilization of the PE compilations instances.

Notice that while a more general ALU is general enough to cover the graph, the generality comes at the cost of very low utilization. Low utilization manifests as increased area and static power needed to maintain the unused functional units. Ideally, generated PEs will have 100% utilization whenever they are used; in practice this is impossible for reconfigurable PEs, so instead we attempt to maximize utilization by encoding it within the genetic search cost function.

4) Communication Reduction: Finally, communication reduction is defined as the number of inter-PE wires in the original application graph versus in the compiled graph. These graph edges correspond to external wires that must be supported by the CGRA routing fabric. These external routing wires are significantly more expensive than internal wires or *intra-PE* wires that occur between operators within a PE We define communication reduction as the fraction of edges in the compiled graph versus the original application graph.

E. Hardware Implementation

Once the reconfigurable PEs are identified, our framework generates ASIC synthesizable Verilog for each of the PEs. Prior to generating Verilog, our tool must first resolve arity issues for each operator by introducing reconfigurability. Recall that fusion may produce fused dataflow graphs where there are more input operands to an operator than the operation can take. For an operator node with $N > M$ inputs where M is the arity of a node, we introduce N-wide multiplexors and introduce configuration wires connect to select ports. These configuration wires introduce the minimal amount of reconfigurability necessary to support the provided application graphs and computation subgraphs.

IV. Genetic Algorithm Evaluation

This section evaluates our PE generation algorithm, including the quality of generated PEs. We explore whether the generated PEs generalize to larger instances of the same application as well as to new applications. We also compare generated PEs against CGRA using ALUs as their PEs shown in Fig. 2. In this section, all evaluations use metrics such as utilization and communication reduction.

Our evaluation draws sample applications from two application domains: image processing and linear algebra. Table I details the sample applications along with the input sizes used in PE generation. We intentionally keep input sizes small to produce small data flow graphs, allowing the genetic search to perform more iterations with larger populations. Using smaller application instances is sufficient as long as their computation structure is representative of larger workload instance. In Section IV-C, we show that the PEs generated by small instances do in fact generalize to larger application instances.

Application	Input size	# nodes
Geometric Interpolation	5×5	265
Convolution	4×4, 3×3	257
Anisotropic Diffusion	3×4	262
Fast Fourier Transform	8	216
Harris Corner Detection	3×3	224
Laplacian	3×3	212
Median Filter	3×3, 3×3	295
Lucas-Kanade Optical Flow	3×3	287
Sobel Operator	5×5	385

(a) Image processing domain workloads

Application	Input size	# nodes
Fast Fourier Transform	8	216
Linear Least Squares Solver	16×2	155
Gaussian Elimination	5×5	185
Matrix Inversion	4×4	228
Matrix Multiply	4×4, 4×4	192
Covariance Matrix	4×4	364

(b) Linear algebra domain workloads

TABLE I: Image processing and linear algebra domain workloads. Small application instances allow RADISH to rapidly search the solution space. All application inputs are matrices, and the second column lists their sizes.

A. Search Progress

We now evaluate whether RADISH's genetic search algorithm effectively finds high-quality solutions according to the given metrics. We perform the searches on an Intel Core i7-8700K CPU with 6 cores (12 threads) at 3.70 GHz and 16 GB of memory. We configured the search to iterate through 100 generations, each with the population size of 50 individuals. RADISH took 9 and 16 hours to generate PEs for the linear algebra and image processing domains, respectively.

Fig. 5 demonstrates that our genetic algorithm improves the set of solutions identified over time. Because the search optimizes many variables at once (Section III-D1), we are interested in solutions on a Pareto-optimal surface. The surface is high-dimensional and not conducive to visualization, so we instead plot the cumulative number of Pareto-optimal solutions over time. Fig. 5 shows that the algorithm does not stagnate. Instead, the line continues to improve which indicates that RADISH continues to find Pareto-optimal solutions throughout the entire search.

B. Genetic Search Results

To evaluate the quality of the results, we plot the communication reduction and utilization of all Pareto optimal solutions in

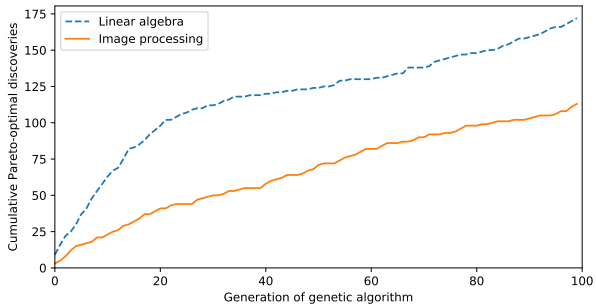
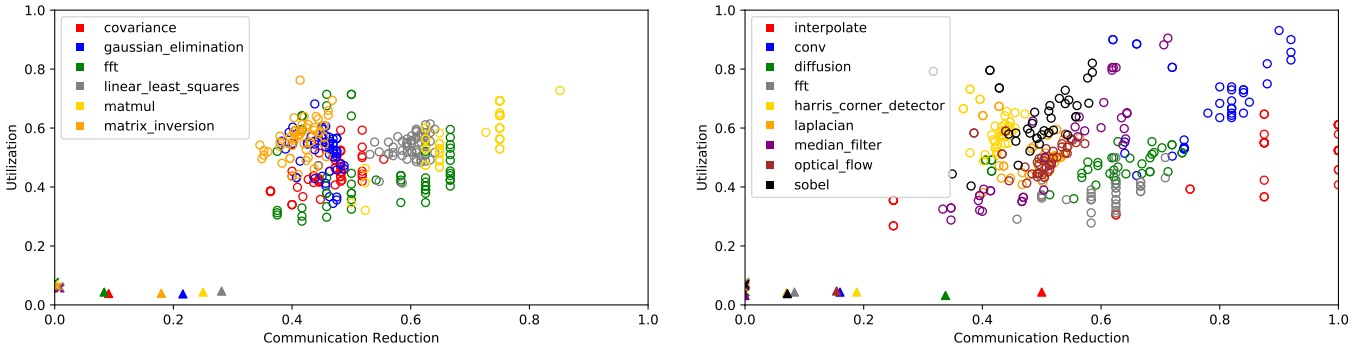


Fig. 5: Our genetic algorithm continues to find Pareto-optimal solutions even later in the search.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Linear Algebra Metrics

(b) Image Processing Metrics

Fig. 6: Utilization versus communication reduction results (up and to the right is better). The Pareto optimal results (circles) generated by our algorithm have higher communication reduction and utilization than ALUs (×s) and double ALUs (triangles). All results have full application coverage.

Fig. 6. All solutions had nearly perfect coverage, so the results are not shown. Each solution (set of PEs) has a marker for each application. The different colors represent the different benchmarks in the domain: for example, a green circle in Fig. 6a represents a Pareto optimal set of PEs being compiled on the FFT application. Green circles lower and to the left are Pareto optimal in some way, just not on the FFT benchmark.

We compare the RADISH-generated PEs to ALUs (×s in the plot) and fused ALUs (triangles) that are traditionally used in CGRA design (see Fig. 2). In our framework, an ALU is represented as a family of operators all connected to the same two inputs and one output. It is reconfigurable only by selecting the single operation to be performed. A fused ALU is an ALU connected to both an output and another ALU. The second ALU must take only one input; the other comes from the first ALU. We represent these ALUs as graphs in our framework and run them through the same compilation and evaluation flow as the generated PEs.

The results in Fig. 6 show that RADISH-generated PEs have significantly higher communication reduction and utilization than ALUs. Unlike ALUs, PEs can activate multiple operations at a time (higher utilization) and replace inter-PE edges with intra-PE edges (communication reduction). The results also highlight the fact that solutions may perform better on some applications than others. By exploring a large swath of the design space at once, RADISH allows the designer to choose the solution that optimizes performance on the most important applications.

The geometric interpolation benchmark is particularly noteworthy because some solutions (red markers in Fig. 6b) achieve 100% communication reduction. The dataflow graph for this workload is "flat" rather than "deep", performing many small operations in parallel. We do not count edges from application inputs or to application outputs in communication reduction (as they are not part of a PE), and the generated PEs capture all of the remaining structure.

C. Generated Processing Element Generality

One of the main benefits of a DSRA is flexibility within the domain. Unlike ASICs, DSRAs promise to accelerate new applications that were not considered at design time, as long as those applications fall within the same domain. In Section III-A we detail how and why we use dynamic dataflow traces as the input to our tool. The approach offers many simplifications, but raises the issue of generality. Programs may exhibit different characteristics with different data or dataset sizes. This section evaluates the generality of the set of PEs generated by our tool.

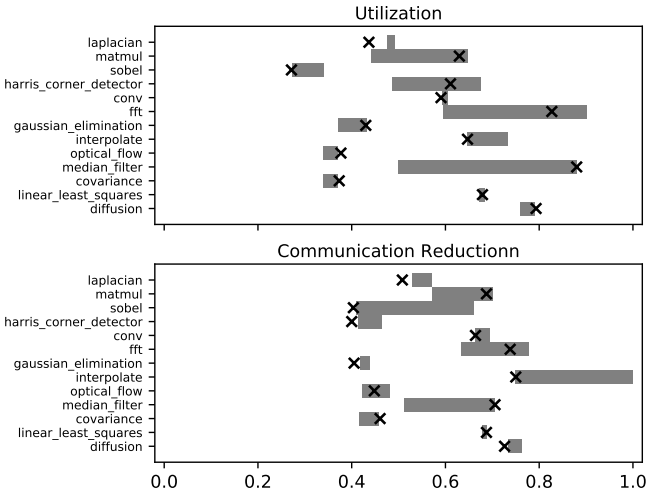
Coverage is the metric of concern when discussing generality. If a set of PEs has lower than 100% coverage for an application, the compiler could not map all functionality onto the PEs. To run these applications, the architecture would need more general-purpose hardware like ALUs or LUTs as a fallback. Using the general purpose fallback will reduce performance compared to the specialized PEs, so higher coverage is always better.

We use cross validation to measure two kinds of generality: *application generality* and *domain generality*. Results are shown in Fig. 7 and discussed later. We performed the genetic searches described in this section with population size 10 for 10 generations.

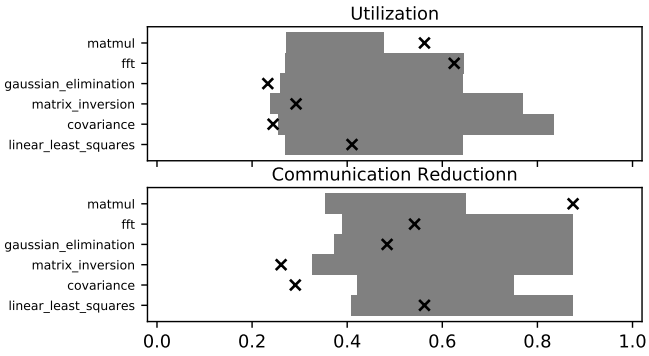
1) Application Generality: Application generality is defined as how well PEs support the same application with different parameters or dataset sizes. Since inputs to our tool are dataflow graphs generated from a specific application instance, it is not obvious whether PEs generated from one instance of the application are sufficient to implement others. To measure application generality, we first generate PEs from instances of an application that differ in parameter or input dataset sizes. We then introduce a new, larger instance of the application and attempt to compile it using the generated PEs. Ideally, if the generated PEs generalize to other application instances, they would achieve 100% compilation coverage on the new application instance, and the other metrics would be comparable to those from the training set.

Fig. 7a shows the results when we use four application instances (training set) to generate PEs and compile them for a fifth application instance. The black × shows the metrics for the holdout instance, and the gray bar shows the range of the metric for the instances from the training set. When the × lies

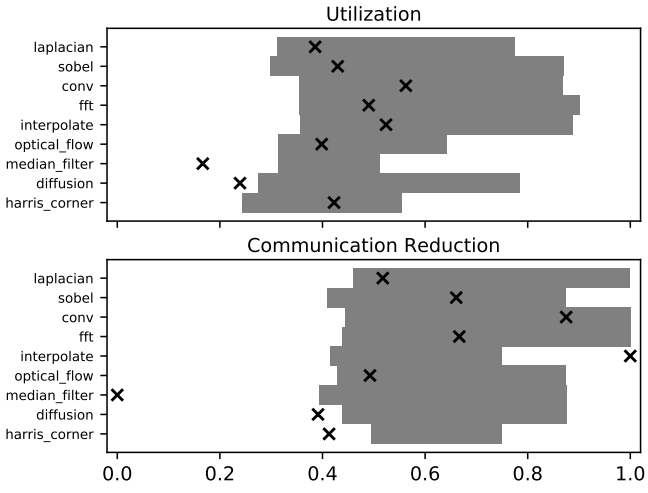
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Application Generality



(b) Linear Algebra Domain Generality



(c) Image Processing Domain Generality

Fig. 7: Generality tests via cross validation. The holdout benchmark is on the y-axis. The gray bar shows the range of a metric for the training set itself, while the metric for the holdout is the black $\times$.

in the gray bar, the tool performed roughly as well on the new instance as it did on the ones it trained on.

The results in Fig. 7a show that our tool successfully generalizes to unseen instances of seen applications. Notably, RADISH achieves 100% coverage on all benchmarks (not shown), meaning that it successfully compiled all unseen instances. For other metrics - utilization and communication reduction - we find that all applications are comparable to those from the training set.

2) Domain Generality: Domain generality measures how well PEs support new, unseen applications from the same domain. We use an experiment similar to the previous one to demonstrate that our tool does not suffer from over-specialization to a specific application. For each application a in a domain, we generate PEs based on the entire domain except a. We then compile application a with those PEs, and compare the resulting metrics with those from compiling the PEs on the other applications.

Fig. 7b and Fig. 7c show the cross validation results. The results are mostly positive, with most $\times$s laying inside the gray bars to show that the test performance was within that of the training set. Like application generality, we also find that most generated PEs achieve nearly 100% coverage against the holdout benchmarks. We observe a few anomalies which do not score well against all three metrics; this is due to that fact that some applications do not compile well against PEs learned from the rest of the domain. In particular, the median filter does not score well in coverage, communication reduction, and utilization. This indicates that the application has a different structure of operator use, and is (in some sense) not representative of the domain.

V. HARDWARE EVALUATION

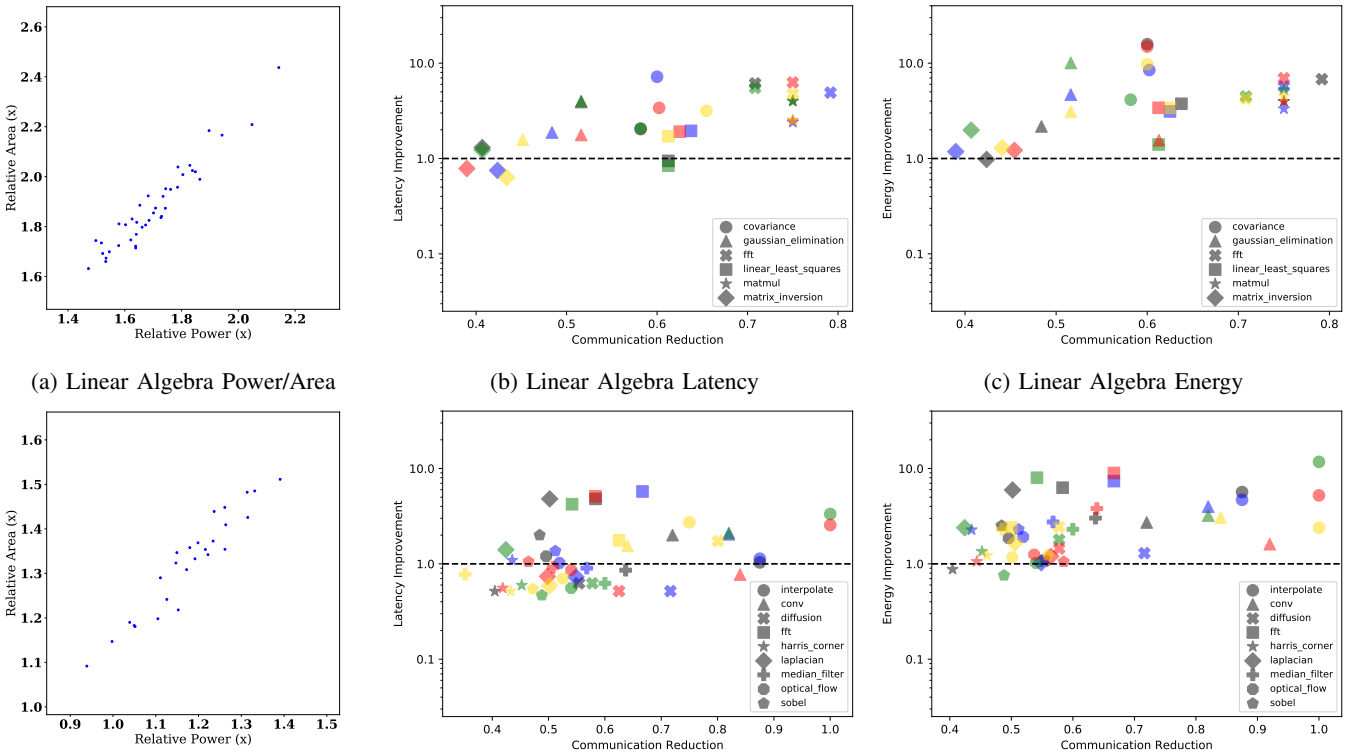
A. Methodology

To evaluate RADISH, we model a simple CGRA-style accelerator with PEs arranged in a grid array and memory on the periphery (see Fig. 2). For each domain, we use RADISH to generate 40 different sets of Pareto-optimal PEs. We compose each of those into a CGRA fabric to form a DSRA for that domain, and we compare against a CGRA with the traditional ALU and fused-ALU PEs.

We compare the power, area, latency, and energy usage of the two designs. We estimate the power, area, and latency per PE using Synopsys Design Compiler using a 32 nm generic library. We then use compilation results to determine the number of occurrences of each PE. Application compute power and area is estimated by multiplying power and area per PE by the number of PE occurrences. Application latency is estimated by extracting the estimated critical path of each PE from post-synthesis results. We assume an adjustable clock generator which clocks the CGRA at the maximum possible frequency. The clock period is set by taking the compiled application and identifying the block with the longest critical path. The latency of each application is then computed as the longest path of PEs in the compiled graph.

We also compare against ALU-based coarse-grained reconfigurable array designs to estimate the potential power

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Linear Algebra Power/Area (b) Linear Algebra Latency (c) Linear Algebra Energy

(d) Image Processing Power/Area (e) Image Processing Latency (f) Image Processing Energy

Fig. 8: Power, area, latency, and energy improvement of CGRA with RADISH-generated PEs over an ALUs. Power and area reported as geometric mean over applications for each Pareto optimal design. RADISH designs have similar latency but lower power yielding better energy efficiency. Latency and energy improvements are correlated with communication reduction. The latency and energy plots show the top 5 solutions (across the geometric mean of the applications). Colors correspond to different solutions while shapes denote different applications. For example, in (c), all red markers are part of the same design, and the red "X" is the FFT application in that design.

and area savings of our DSRA designs. In particular, we compare against CGRA designs which use ALUs or compound ALUs PEs shown in Fig. 2. Each ALU element contains four arithmetic operations (add, subtract, multiply, popcount), 12 bitwise operations (binary and unary OR, XOR, NOT, and AND, and bit shifts), and six comparison operations. Division is implemented as a separate unit for ALU-based designs.

For each set of PEs in the CGRA and DSRA designs, we assume a square grid interconnect between PEs in the array. We estimate the size of the router based on the maximum number of inputs that are used for any application and add it to the number of ports needed to communicate with neighboring routers. To do this, we first compile the application with the generated PEs then find the maximum number of input ports p of any compiled PE in the application. This provides an estimate of how many wires must feed into the router connected to the PE with the most inputs. We then assume that each router connects to its neighboring routers also with p ports and that routers use an all-to-all crossbar topology. We then estimate the power and area required to implement each router using Synopsys Design Compiler. Total power and area per application is estimated by determining individual PE power and area, and adding it to the estimated power and area to support interconnect routers.

B. Results

The relative power, area, latency, and energy results for each application within the linear algebra and image processing domains is shown in Fig. 8. Results show the improvements for all of the Pareto-optimal solutions (sets of PEs) against the single ALU-based CGRA design.

We find that in most cases, RADISH is usually able to generate solutions which are better than CGRAs with ALU PEs. For linear algebra, we observe that the geometric mean power and area improvements of RADISH-generated designs over ALU-based CGRAs is 1.4× and 1.5× respectively. However, for certain Pareto optimal designs, we observe power and area improvements of up to 3.18× and 3.24× respectively. For image processing, our results yield a geometric mean power and area improvement of up to 2.14× and 2.4× respectively. In the best cases, we observe up to 3.7× and 4.5× power and area improvement respectively.

Unlike ALU PEs, our generated solutions do not contain all possible operators which improves the internal utilization of our PEs, and translates to fewer idle datapaths and compute units. For most RADISH generated designs, we find that router interconnect power and area consumes on average between 40.1% of the power and 47.4% of the area for linear algebra benchmarks and image processing benchmarks. For ALU-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

based designs, routing interconnects still consume between 22.5% to 27.4% of the total accelerator power. This is because ALU-based CGRAs require less routing between routers since there are fewer inputs per PE. However, since more ALUs are required to implement an application, they require more routers than our generated DSRA designs. The high routing power and area component of our DSRA imposes an Amdahl's limitation on the gains that can be achieved by improving the PEs in the fabric. Finally, in terms of PE count, our generated DSRA designs use between 1.1× to 6.3× fewer PEs when using our generated blocks than when using ALUs.

In terms of latency, we find that our RADISH-generated CGRAs on average have similar latencies as ALU-generated block (Fig. 8b and Fig. 8e). This is because the critical path of our PEs are longer because they contain many more operators and configuration logic. However, in terms of energy efficiency, RADISH-generated designs achieve a geometric mean energy improvement of 1.7× and 2.5× for image processing and linear algebra respectively. In some cases, energy efficiency improvements can reach up to 11.7× for image processing and 28.9× due to both power and latency improvements afforded by RADISH designs.

VI. Related Work

Domain-specific accelerators have an established history which dates back to the 1960s [19] and has been a subject of continuous interest. Many prior works rely on manually identifying common reconfigurable *processing elements* (PEs) and design patterns [20] to better abstract the commonly used hardware structures. However, these approaches rely on designer insights to identify PEs, and cannot automatically identify and exploit common computational structures within an application domain. Most recent CGRA-based work relies on manually specified PEs such as ALUs or compound ALUs [21]. Prior work such as automatic instruction set customization [22], [23] and GreenDroid [24] are limited to basic blocks boundaries when identifying potential fragments of computation to accelerate. Template-based approaches such as [25], [26] histogram the number of edges between different type of nodes in the dataflow graph; frequent edges transitions between operators are used to build up templates. As a result, template-based approaches are limited to generating smaller sized PEs and do not scale as well to capture larger more global computation structures.

The most similar work to ours is Totem [27] which proposes a custom reconfigurable array generator for a target domain of applications. However, Totem combines datapaths at the netlist level which only contains coarser-grained elements such as ALUs, registers, and RAMs; RADISH combines datapaths at the operator level which allows more efficient fusions at the arithmetic operator level. Brisk et al. [16] propose datapath fusion using ILP to generate domain-specific accelerators which is later used by Stojilovic et al. [17]. RADISH's datapath fusion algorithm is similar to theirs; however, unlike these prior works, we also provide a compiler-in-the-loop that guides the search using higher-level cost metrics.

The PE identification and refinement process proposed in this work is similar to the task of instruction set customization (each PE is a new instruction). Huang et al. [28] propose an instruction cosynthesis formulation using simulated annealing to generate an instruction set and map code to generated instructions. Clark et al. [29], [22], [30] propose generating custom instructions which are realized using a tightly coupled coprocessor unit with a general-purpose core. Park et al. [31] propose CGRA Express which dynamically fuses functional units to operate in the same cycle but still relies on general-purpose interconnects to connect the fused elements. Unlike these prior works, our work uses a higher-level cost function which includes compilation coverage, block utilization, and communication reduction.

More recently, there has been a resurgence of interest in automatically generating domain-specific accelerators for emerging application domains and increasingly complex workloads. GreenDroid [24] and conservation cores [32] both propose a sea of fixed-function accelerators architecture to accelerate portions of the Android kernel. However, unlike our work, these works limit acceleration to basic blocks identified by hot code profiling and do not consider more global structures across applications. More recently, Prabhakar et al. [20] propose generating reconfigurable hardware from parallel patterns but their technique still requires designer expertise to identify patterns. Nowatzki et al. [23] proposed a new ILP-based technique for automatically discovering instruction set customizations, Their work also limits their search within program basic blocks which limits the size and scope of the PEs that are discovered. Cong et al. [33] propose accelerator-rich CMPs which is a similar sea of accelerator architecture for domain-specific acceleration. Coole et al. [34] approach a similar problem to ours with a different approach; they use a clustering-based algorithm, whereas we use a genetic algorithm guided by a high-level cost function.

Finally, our work uses dataflow graphs for simplicity and to expose sufficient information to the ILP datapath fusion algorithm. Other approaches such as modulo scheduling [35] or parallel patterns [36] offer more compact representations but the coarser granularity abstracts away datapath details required for fusion. Basic block representations such as those used in [24] are limiting because it limits fusion to within the basic block; by unrolling the datapath graph, RADISH is not limited to basic blocks.

VII. Conclusion

We presented RADISH, a tool for automatically generating reconfigurable PEs from a given set of applications. We use a genetic algorithm to combine a novel PE fusion algorithm and a compiler-in-the-loop for evaluating PEs. The algorithm iteratively improves the solution quality, resulting in PEs that outperform traditional CGRA PEs.

Acknowledgements

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, and OAC-1739419; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Qualcomm, Oracle, Google, Huawei.

REFERENCES

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015.

[2] Amazon, "Ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/, 2018, accessed: 04-01-2018.

[3] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," *SIGPLAN Not.*, vol. 49, no. 4, pp. 255–268, Feb. 2014.

[4] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 1–13.

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*. New York, NY, USA: ACM, 2014, pp. 269–284.

[6] Xilinx, "Vivado design suite user guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf, accessed: 11-7-17.

[7] Altera, "Sdk for opencl," https://www.altera.com/ja_JP/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf, 2013, accessed: 11-7-17.

[8] Mentor, "Catapult high-level synthesis," https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls, 2018, accessed: 11-7-17.

[9] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[10] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A co/processor for streaming multimedia acceleration," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 28–39.

[11] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, 2003, pp. 61–70.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[13] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon, "Genetic approaches for graph partitioning: a survey," pp. 473–480, 2011.

[14] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, Feb 1970.

[15] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2000, pp. 13–23.

[16] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 395–400. [Online]. Available: http://doi.acm.org/10.1145/996566.996679

[17] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Creating domain-specific reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 5, pp. 681–694, May 2013.

[18] P. J. S. Aric A. Hagberg, Daniel A. Schult, "Exploring network structure, dynamics, and function using NetworkX," *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pp. 11–15, Aug 2008.

[19] G. Estrin, "Organization of computer systems: The fixed plus variable structure computer," in *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, ser. IRE-AIEE-ACM '60 (Western). New York, NY, USA: ACM, 1960, pp. 33–40.

[20] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 651–665.

[21] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, "Evaluating programmable architectures for imaging and vision applications," in *MICRO*, 2016.

[22] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 272–283, May 2005.

[23] T. Nowatzki, M. C. Ferris, K. Sankaralingam, C. Estan, N. Vaish, and D. A. Wood, *Optimization and Mathematical Modeling in Computer Architecture*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[24] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, Mar. 2011.

[25] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '02. New York, NY, USA: ACM, 2002, pp. 641–648.

[26] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh, "Instruction generation and regularity extraction for reconfigurable processors," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 262–269.

[27] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, pp. 111–119.

[28] I.-J. Huang, A. M. Despain, and P. Danzig, "Co-synthesis of instruction sets and microarchitectures," 1994.

[29] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 129–.

[30] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 30–40.

[31] Y. Park, H. Park, and S. Mahlke, "Cgra express: Accelerating execution using dynamic operation fusion," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 271–280.

[32] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 205–218.

[33] Y. T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich cmps: From concept to real hardware," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 169–176.

[34] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from opencl using reconfiguration contexts," *IEEE Micro*, vol. 34, no. 1, pp. 42–53, Jan 2014.

[35] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM, 1994, pp. 63–74.

[36] 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 paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 389–402.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2878194, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Max Willsey is a Ph.D. student at the Paul G. Allen School for Computer Science & Engineering at the University of Washington. He received his B.S. in Computer Science from Carnegie Mellon University (2016). He is a recipient of a Qualcomm Innovation Fellowship and an NSF Graduate Research Fellowship honorable mention. His research interests are in programming languages and computer architecture with applications in biology.

Vincent T. Lee (S'18) received his B.S. degree in Electrical Engineering and Computer Science from the University of California, Berkeley (2013). He is currently pursuing a Ph.D. degree at the Paul G. Allen School for Computer Science & Engineering at the University of Washington, and is a recipient of two Qualcomm Innovation Fellowships. His current research interests include stochastic computing, computer architecture, hardware/software codesign, and accelerator design for emerging applications.

Alvin Cheung is an Assistant Professor in the Paul G. Allen School of Computer Science and Engineering at the University of Washington, affiliated with the programming languages and database research groups. He received his undergraduate degrees from Stanford University, and his PhD in Computer Science from MIT. Alvin's research focuses on designing new techniques in programming languages and data management to solve systems and end-user programming problems.

Rastislav Bodík is a Professor in the Paul G. Allen School of Computer Science & Engineering at the University of Washington. He received his Diploma of Computer Engineering from the Technical University of Košice (1992), Košice, Slovakia, and his Ph.D. in Computer Science from the University of Pittsburgh (1999), Pittsburgh, USA, His primary research interests are in computer-aided construction of software using program synthesis.

Luis Ceze (M'03) is a Professor in the Paul G. Allen School of Computer Science & Engineering at the University of Washington and a Venture Partner at Madrona Venture Group. He received his B.S. and M.Eng. in Electrical Engineering from the University of São Paulo, Brazil, and Ph.D. in Computer Science from the University of Illinois at Urbana Champaign. His research focuses on the intersection between computer architecture, programming languages, machine learning and biology. His current focus is on approximate computing for efficient machine learning and DNA-based data storage. He co-directs the Molecular Information Systems Lab (MISL), the Systems and Architectures for Machine Learning lab (SAML) and the Sampa Lab for Hardware/Software Co-design.