Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign

Tony Nowatzki* Newsha Ardalani† Karthikeyan Sankaralingam‡ Jian Weng*
University of California, Los Angeles* SimpleMachines, Inc.† University of Wisconsin-Madison‡
tjn@cs.ucla.edu

ABSTRACT

Recent programmable accelerators are faster and more energy efficient than general purpose processors, but expose complex hardware/software abstractions for compilers. A key problem is instruction scheduling, which requires sophisticated algorithms for mapping instructions to distributed processing elements, routing of operand dependences, and timing the arrival of operands to enable high throughput.

The complex dependences between mapping, communication and timing make prior scheduling techniques insufficient. Optimization-based approaches are too slow, and heuristic-based approaches cannot achieve high quality. Our first insight is that the algorithm can be solved in a series of phases with overlapping responsibilities to reduce complexity. Second, it is possible to combine optimization-based and stochastic-heuristic based search strategies, to exploit the best features of both. This leads to the two primary techniques we explore, phase overlapping and hybridization.

In this work we explore the codesign of scheduling algorithms with a challenging-to-schedule programmable accelerator. We show we can improve its area by 35% by trimming its scheduling-friendly structures, using a scheduling algorithm that is 5× faster than the state-of-the-art optimization-based scheduler, with up to 2× better throughput.

ACM Reference Format:
Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, Jian Weng. 2018. Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243176.3243212

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243212

1 INTRODUCTION

With the slowing of exponential technology scaling, it has become clear that processors with traditional instruction sets are limited in terms of their energy efficiency and hence performance scalability. To address this problem, programmable accelerators add a rich interface between the software (compiler or programmer) and hardware layers, enabling more efficient and exposed microarchitectures while retaining some level of programmability and generality [10, 11, 22, 29, 34, 42, 44, 46, 48, 55, 56]. Essentially, these accelerators have repurposed spatial architecture principles for the specialization era, where such a rich interface is acceptable.

In an attempt to achieve ASIC-like energy efficiency, several recent spatial architectures completely eschew control capabilities from each processing element (PE) [12, 13, 15, 16, 20, 35, 36], hearkening back to classic systolic arrays [8, 50], but with a more flexible network. We refer to such designs as *dedicated-PE* accelerators.

The primary challenge in adopting these exposed and restrictive microarchitectures is that they require increasingly demanding compiler support. One of the most recurring and difficult problems in these designs is the problem of instruction-scheduling: deciding when and where each computation and operand communication should be performed. More specifically, this includes the highly dependent problems of *mapping* instructions to resources, *routing* dependent values between instances of instructions, and the critical task of assigning the *timing* of concurrent instructions to achieve a high execution rate (hardware utilization). We identify and demonstrate that the main challenge in attaining high throughput is precisely matching arrival times of instruction operands, avoiding delay-mismatch.

Though scheduling algorithms exist for dedicated-PE accelerators, they inadequately address the above timing problem, imposing a stark hardware/software co-design tradeoff: either sacrifice area/power efficiency by adding structures to the hardware to ease the burden of scheduling, or have a lean accelerator and suffer exorbitant scheduling times of a rigorous schedule-space search.

Fast scheduling algorithms are typically based on a variety of heuristics around minimizing communication distances and resource consumption. They have extreme difficulty

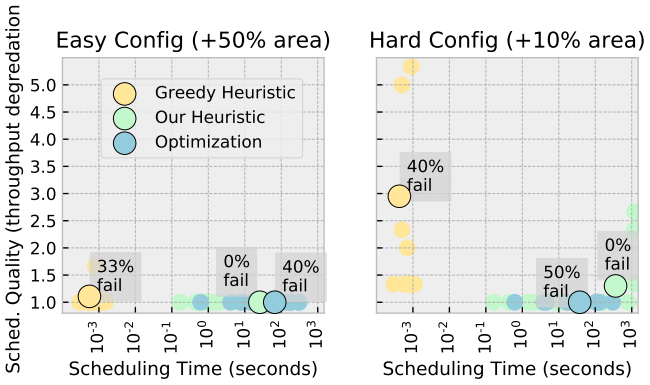

Figure 1: State-of-art scheduler effectiveness. Lower is better in both axes. Max time: 20 Minutes, Benchmarks/methodology in Section 6 (page 9)

constructing a schedule with matching arrival times on all inputs, as their timing is dependent on the source instructions' mapping and routing of operands. The more rigorous approach is to solve the problem mathematically, commonly as a mixed integer linear program (MILP) or using satisfiability modulo theory (SMT) [38]. While this can lead to optimal solutions, and require less complex hardware, they are far too slow when applied to even modestly sized dedicated-PE accelerators.

The fundamental codesign tradeoff can be seen in Figure 1, which shows the scheduling results for the accelerator we target in this work, stream-dataflow [35]. The x-axis is the scheduling time in a log scale, and y-axis is the expected throughput degradation (big circles are centroids, small circles are benchmarks). Existing simple heuristics fail due to a small search space, and existing optimization techniques fail due to too-large a search space. The sophisticated heuristic scheduler we develop, in green, always succeeds. However, it either must suffer performance loss (up to 2.5 ×) on the lean and "hard" hardware configuration, or spend 1.5× area to achieve good performance on the "easy" hardware configuration.

Goal and Insight: The goal of this work is to discover scheduling techniques which are *both* reasonably fast and find high-performance solutions on lean hardware configurations. To this end, we have two main insights. First, we observe that it is *neither* necessary to solve the problem in a series of independent phases for mapping, routing and timing, nor to solve them all simultaneously. Rather, these different responsibilities can be solved in *overlapped phases*, which can prevent poor greedy decisions early in the algorithm. For example, an overlap-phased algorithm would perform the mapping and routing together, in order to find a good mapping, but would not fix the routing before performing the routing and timing together. Our second insight is that it is possible to combine heuristic-based and optimization-based methods into a *hybrid* scheduling algorithm, by applying them on the phases they are best suited for.

In this work, we explore a design space of instruction scheduling algorithms for an extremely-lean dedicated-PE array from the stream-dataflow [35] accelerator, a representative programmable accelerator. We then explore how these algorithms can tradeoff scheduling time, solution quality (performance) and hardware overhead.

Specifically, our contributions are:

- Identification of *delay mismatch* as the bottleneck in dedicated-PE scheduling; mitigation with delay-FIFOs.
- Developing the principles of **phase-overlapping** (overlapped execution of scheduling responsibilities) and **hybrid-scheduling** (integration of heuristic phases), which can be applied to spatial scheduling in general.
- Evaluation on a representative dedicated-PE accelerator, demonstrating phase-overlapping/hybrid-scheduling effectiveness on extremely restrictive hardware.

As a minor contribution, we construct a sophisticated heuristic scheduler based on stochastic search for use in hybrid scheduling, and augment a prior MILP scheduling formulation with advanced delay-matching techniques.

Findings: First, solving *any* phase independently, even with sophisticated optimization based search, conclusively fails on a dedicated-PE accelerator. Second, this work demonstrates the importance of compiler/architecture codesign – we show we can improve the area-efficiency of a high performance accelerator by 35% by trimming its scheduling-friendly structures, using a scheduling algorithm that is 5× faster than the state-of-the-art optimization-based scheduler, with up to 2× better throughput.

Paper Organization: We first discuss challenges and constraints for dedicated-PE accelerator scheduling (§2), then overview the explored and developed scheduling techniques (§3). In the following section, we discuss techniques to enable phase overlapping for an optimization-based scheduler (§4). This motivates the discussion of our heuristic scheduler and its integration with optimization phases (§5). Finally, we then discuss the evaluation methodology (§6), results (§7), related work (§8), and conclude (§9).

2 SCHEDULING CHALLENGES

In this section, we elucidate the challenges of scheduling for dedicated-PE accelerators. First, we describe the essentials of spatial scheduling, then elaborate on our scheduling setting through contrasting with traditional CGRAs, and finally describe the fundamental challenges and potential techniques.

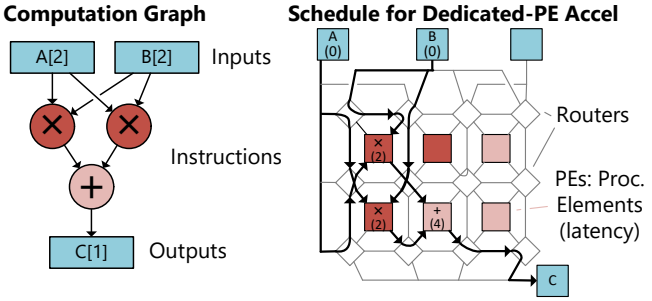

Figure 2: Example Scheduling Problem

2.1 Spatial Scheduling Essentials

We define a spatial architecture as one which exposes the management of hardware resources for execution, routing, storage, and/or timing of operations to the software. Spatial scheduling is the task of deciding an execution plan for a dataflow graph of computations of interest to this rich software interface. Figure 2 shows an example of mapping a graph of instructions onto a hardware graph, for an abstract dedicated-PE architecture. In the accelerator context, dataflow graphs correspond to the computations from a loop or loop nest. There are three main scheduling responsibilities, which are highly interdependent:

- **Mapping** Assign computation elements (instructions, inputs, outputs) to compatible resources.
- **Routing** Assign dependences between computation elements (their communication) to network resources.
- **Timing** Coordinate the timing of computation and communication to maximize the potential throughput.

Depending on the particular architecture and its capabilities, the above scheduling problem can tend to be either trivially easy, or very difficult. Typically, hardware which is optimized for area or power has fewer structures to ease the burden of scheduling.

2.2 Dedicated versus Shared PEs

CGRA Styles: Programmable accelerators often embed a coarse grain reconfigurable architecture (CGRA), a networked collection of processing elements (PEs) that can be reconfigured for different computations. One defining feature of the PE is in how instructions are mapped to it – either the PE can be *shared* by multiple instructions and time-multiplexed (eg. [10, 22, 30, 42–44, 56]), or it can be *dedicated* to one static instruction (eg. [12, 13, 16, 20, 35, 36]). In terms of their execution model, both CGRA types attempt to pipeline multiple iterations of the computation graph, but a dedicated PE CGRA necessarily attempts to fully-pipeline the computation.

PE Examples: Figure 3 shows a prototypical example of a shared PE and dedicated PE. These PEs have a very similar structure in terms of routing values from neighboring

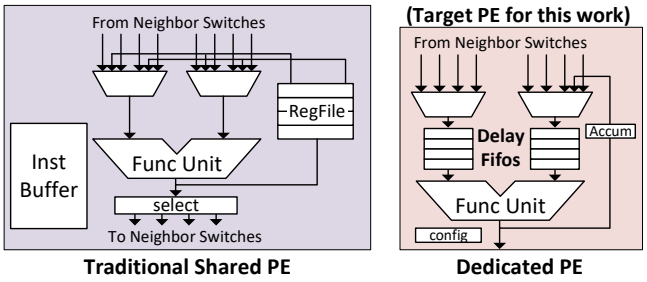

Figure 3: Traditional Versus Dedicated PEs

switches (routers) in the network into a functional unit, and maintaining state for both instruction configuration and local reuse. However, since a shared-PE maps multiple instructions, it must contain a configuration or instruction buffer to remember which instructions to time-multiplex, and at what point in the execution. To allow communication between instructions residing on the same PE, a register file is added. On the other hand, a dedicated-PE array only maps a single instruction, and pipelines multiple instances of that instruction over time, with perhaps a single register for reduction. The "delay FIFO" is a critical hardware resource which eases the scheduling burden, which we explain next in Section 2.3.

Broadly, the advantage of the shared-PE array is the flexibility in mapping large computation graphs, at a cost of requiring some extra hardware support. Extra support is required in the compiler for a dedicated-PE array to *resize* the computation graph to the hardware, typically through vectorization [20]. Note that we assume the compiler/programmer has already resized the graph to meet the resource constraints of the CGRA. A combined vectorizer/scheduler is beyond the scope of this work.

2.3 Scheduling Dedicated-PE CGRAs

Scheduler Goal: As for most accelerator architectures, the goal of the instruction scheduler is to achieve the maximum throughput for a given input computation graph. In a traditional shared-PE CGRA, this is accomplished by minimizing the "initiation interval" II – the number of cycles between iterations of the loop. A secondary objective would be to reduce the total latency of the schedule, to reduce the penalty of filling/draining the pipeline.

For a dedicated-PE CGRA, an II=1 schedule is intuitively guaranteed just by having fully-pipelined functional units[1], and implies 100% utilization of any mapped PEs. However, when taking into account the timing and limited buffering, this is not quite the case.

Affect of Delay-Mismatch: If the relative delay of two inputs arriving at a PE do not match, the throughput achievable

[1]We ignore recurrences besides accumulation for this discussion, these are handled in the architecture-specific Section 6.1.

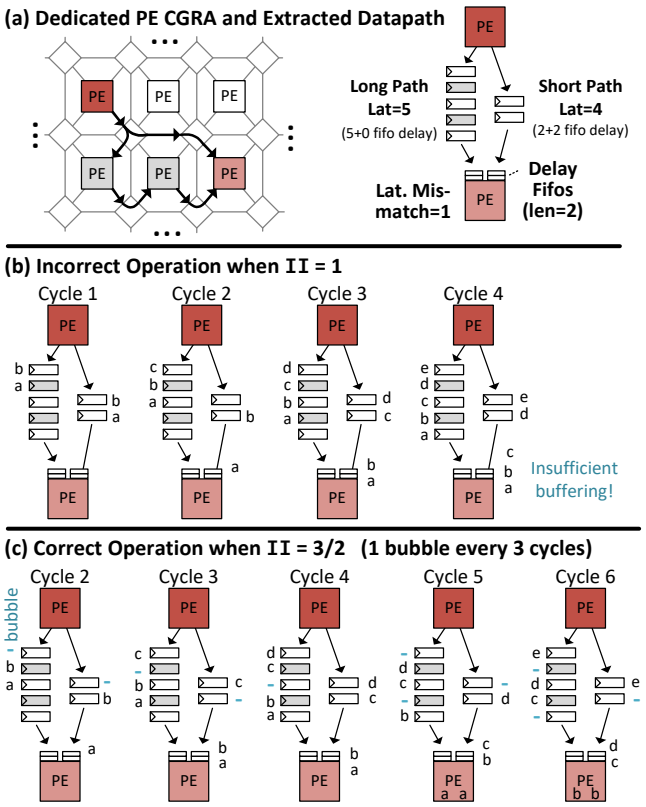

Figure 4: Dedicated PE CGRA Timing Challenge

is reduced. Figure 4 explains this effect. Figure 4(a) shows a generic dedicated PE CGRA and the extracted datapath, highlighting the source and destination PE, as well as the buffer stages between them (5 on the long path, 2 on the short path). Note that the destination PE has two slots in its delay FIFO, so even though there is an overall delay mismatch of 3, the delay FIFO can lengthen the short path, and get a mismatch of 1. In Figure 4(b) and (c), we show the datapath operation by examining consecutive outputs (a,b,c,d,e,...) from the source node. In Figure 4(b) we show that if we assume II=1, there will be insufficient buffering on cycle 4, because the corresponding inputs on the long path will not yet have arrived.

A simple solution sets II to (1 + delay mismatch), which guarantees a mismatch number of bubbles behind a value while it is waiting for its corresponding input down the long path. However, we can do better if we have buffering at the PE inputs, specifically the delay-FIFOs mentioned earlier. These not only reduce the mismatch, they can also hold multiple inputs while waiting for the long path. This can be seen in Figure 4(c), which shows the correct pipeline operation using a *fractional* II of 3/2 (one bubble every 3 cycles). In general, the best II is:

$$\text{II} = \max_{PE \in PEs} \frac{\text{FIFO\_LEN} + \text{mismatch}_{PE}}{\text{FIFO\_LEN}} \quad (1)$$

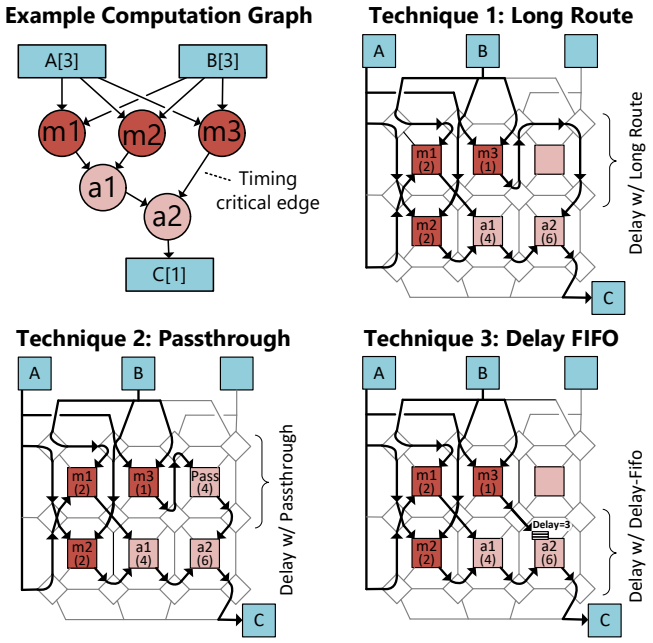

Figure 5: Three delay-matching techniques

2.4 Delay Matching Techniques

Mismatched delays, which reduce maximum throughput, occur either because an instruction's producers may have been mapped at different relative distances (mapping/routing issue), or because the computation graph contains communication between instructions at different depths. Figure 5 shows several legal example schedules for a computation graph. The number underneath each instruction indicates which cycle that instruction will be performed at. This delay estimation assumes one cycle per instruction and network hop.

This figure demonstrates three different strategies for enabling delay matching. Each of these has the same computation vertex to hardware node mapping, and most of the routing is similar as well. The only difference is in how the $m3 \rightarrow a2$ edge is delayed, which needs three additional cycles of delay. Technique 1 is to use a *longer route* between the mapped location of $m3$ and $a2$; this consumes extra routes. Technique 2 uses a PE as a no-op, which we refer to as *pass-through*. While this consumes less routing resources than the previous schedule, it takes an extra PE resource. Schedule 3 balances timing by using the *delay-FIFO* to add delay on $a2$'s input.

In general, all three techniques are required to balance delay, and are useful in different situations as they complement each other. This is partly because the delay-FIFOs are expensive in hardware, making it is desirable to keep these as small as possible. Other techniques can help compensate. It is also possible to combine the techniques, ie. to use two

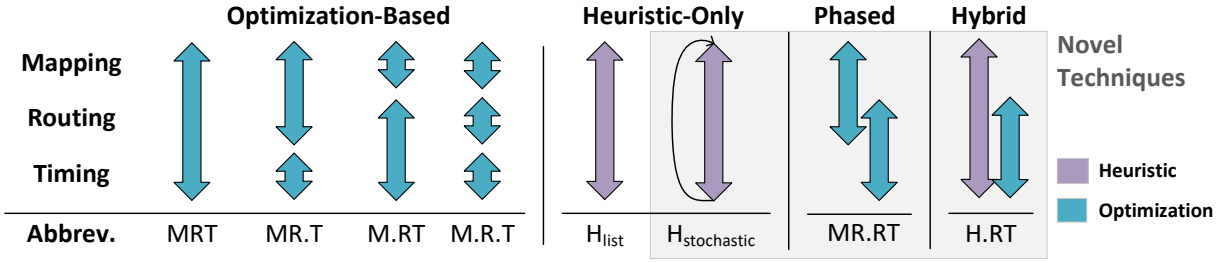

Figure 6: Spatial Scheduling Algorithm Overview

passthrough nodes, adding configurable delay to each one, and also using an extra long route. Passthrough is also useful purely for increasing routing bandwidth, provided there are extra PEs available.

CGRA Scheduling Objective: Overall, because throughput is critical, we place a higher priority on minimizing the maximum mismatch, and a lower priority on minimizing latency. Minimizing latency is marginally helpful for reducing setup and drain time, which is mostly an effect for short phases. Also, reducing latency helps lessen the impact of recurrences (a reuse of a value on a subsequent iteration), which also limit the II. In practice, though, we find that reduction recurrences are not common, and can often be taken off the critical path through loop transformations.

Codesign Implications: A major factor in area overhead comes from the delay FIFOs. Later evaluation will show up to 50% overhead for a relevant accelerator. As we will see in the analysis in subsequent sections, the FIFO-length plays the primary role in the difficulty of the problem, motivating the need to mitigate with advanced scheduling techniques.

3 SCHED. TECHNIQUES OVERVIEW

We apply two main techniques in this work, phase overlapping (performing scheduling through phases with overlapping responsibilities) and hybrid scheduling (integrating heuristic phases with optimization phases). Here we provide a conceptual framework to reason about their integration; a visual depiction with algorithm abbreviations is shown in Figure 6.

Technique 1: Phase Overlapping: The most powerful and slow scheduler jointly solves the mapping, routing and timing problems (termed MRT in Figure 6). A natural way to reduce the complexity would be to break the problem into phases, either performing them all independently (M.R.T), mapping and routing together (MR.T), or routing and timing together (M.RT).

While faster, the tight dependence between scheduling responsibilities makes this approach less effective. A change in the mapping from computation vertex to resource node may make routing easier because an open path exists, but might come at the expense of making the timing constraint more difficult to solve. A change in the routing might open up a new space for routing other edges, but may prevent certain mappings from even being possible. Therefore, fixing one phase before moving onto the next may prove irrecoverable.

We propose here to break the scheduling problem into *overlapping phases*. An initial phase may address some responsibilities, commit to a subset of the decisions, and the next phase would addresses an overlapping set of responsibilities. In this context, the phase-overlapped scheduler performs joint mapping and routing, then fixes the mapping decisions, then performs joint routing and timing (MR.RT). This scheduler is decidedly faster, because the search space is vastly reduced.

Technique 2: Hybrid Scheduling: As the results will show, employing phase-overlapping to the optimization-based scheduler is much faster, but not sufficiently fast enough to be practical (we are aiming for a few minutes at most). The main question becomes how to improve the scheduling time without sacrificing the ability to find solutions or the solution quality. The principle here is to replace part of the branch-and-bound based search that an optimization solver would apply, with a heuristic-based search which can more quickly find legal solutions. For example, the mapping and routing phase of MR.RT can be substituted with a heuristic. We refer to this as a hybrid scheduler. Notationally, we denote the heuristic as the H phase, and the overall hybrid scheduler as H.RT. Note this is distinct from inputting an initial solution to an ILP solver as a way of attaining an upper bound for a minimization problem.

Finally, the goal of this phase is to select a solution for the subsequent phase which would maximize its chances of success. For example, a heuristic for the MR phase should find a mapping and routing which maximizes the chances of the RT phase to find a legal schedule.

	#	Constraints	Description
Mapping	1,2	$\forall_{v\in V} \sum_{n\in C_{vn}} M_{vn} = 1, \quad \forall_{v\in V} \sum_{n\notin C_{vn}} M_{vn} = 0$	All vertices mapped to exactly one compatible node.
	3	$\forall_{v_1v_2\in G\cap n_1,n_2\in N}\ D_{v_1v_2} \geq DIST_{n_1n_2}(M_{v_1n_1} + M_{v_2n_2} - 1)$	Calculation of distance, D, between each node. $DIST$ is a pre-computed parameter describing hardware node distance.
Routing	4 5	$\forall_{ve\in G,n\in N}\ \Sigma_{nl\in H} M_{el} = M_{vn} + P_{en}$ $\forall_{ev\in G,n\in N}\ \Sigma_{ln\in H} M_{el} = M_{vn} + P_{en}$	Each input (output) edge must be mapped to an input (output) link on the assigned node, or the node is a passthrough.
	6 7	$\forall_{e\in E,r\in R}\ \Sigma_{lr\|H}, M_{el} = \Sigma_{rl\in H} M_{el}\ \forall_{e\in E,r\in R}\ \Sigma_{rl\|H}, M_{el} \leq 1$	Mapped links entering router should equal those leaving router, and should only enter the router once.
	8 9	$\forall_{ve\in G,l\in L}\ M_{el} \leq M_{vl} \qquad \forall_{v\in V,l\in L}\ \Sigma_{ve\in G} M_{el} \geq M_{vl}$	An edge mapped to link implies vertex mapped to link, and a vertex mapped to link implies some edge mapped to link.
	10	$\forall_{l\in L}\ \Sigma_{v\in G} M_{vl} \leq 1$	Only allow one vertex to be mapped per link.
	11 12	$\forall_{e\in E,ve\in G}\ T_e = T_v + \text{LAT}_v + (\Sigma_{l\in L} M_{el}) + X_e$ $\forall_{e\in E,ev\in G}\ \ T_v >= T_e$	Timing Equations: Vertex latency greater than dependent vertex, plus instruction latency, plus number of links
Timing	13	$\forall_{e\in E}\ \ X_e \leq \text{FIFO\_LEN}(1 + \Sigma_{n\in N} P_{en})$	Constrain the maximum FIFO delay, $X(e)$.
	14	$\forall_{e\in E,ev\in G}\ \ mismatch >= T_v - T_e$	Compute mismatch at each vertex.
	15	$\forall_{l_1l_2\in H,e\in E}\ \ O_{l_1} + \text{MLAT}(M_{el_1} + M_{el_2} - 1) \leq O_{l_2}$	No fictitious cycles: connected links have increasing orderO_l.

Table 1: MILP Scheduling Formulation of each Phase (objectives not shown)

4 OPTIMIZATION FORMULATION AND PHASE OVERLAPPING

The objective of this section is twofold. The first is to describe the formulation of the scheduling problem as an MILP, particularly how it extends previous descriptions for advanced delay matching. The second objective is to show how it can be broken into either independent or overlapped phases.

4.1 MILP Formulation

The MILP formulation we employ is based on a general formulation for spatial architectures [37], which we briefly recap below. We then discuss modifications for advanced delay matching using passthrough routing and delay-FIFOs, and modifications to enable phase overlapping.

Optimization Problem Notation: Adopting the nomenclature of [37], we express the computation graph as a set of vertices V, and edges E, and the connectivity as an adjacency matrix $G_{V\cup E, V\cup E}$. For example, $v_e \in G$ represents a vertex v and its outgoing edge e, and $v_1v_2 \in G$ represents two connected vertices. Note that lowercase letters generally are members of the uppercase letter's set. The hardware graph is composed of nodes N, routers R and directed links L to connect them. This is represented by a set $H_{N\cup R\cup L, N\cup R\cup L}$. Finally, only certain vertices are compatible with certain nodes, indicated by set C_{VN}. Figure 7 demonstrates the notation.

As mentioned, the scheduling algorithm's job is to map input, computation, and output vertices to input port, computation, and output port nodes, respectively. The restrictions on hardware lead to the following scheduling constraints (complete model in Table 1):

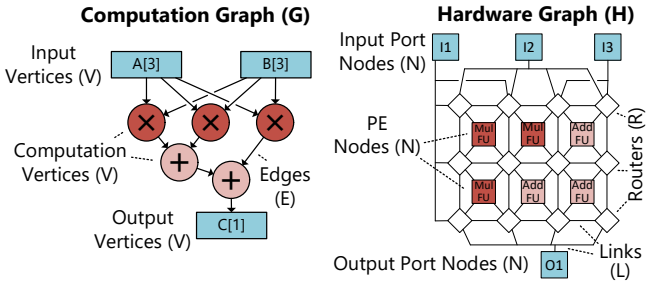

Figure 7: Notation used for MILP Formulation

- **Mapping:** Because we target a dedicated-PE array, all vertices must be mapped to a unique and compatible node. The binary decision variables here indicate vertex to node mapping, M_{VN}.
- **Routing:** For any edge, there must exist a connected path of links, through routers, between the source and destination vertex. Only one value may be mapped per link. The binary decision variables here indicate mapping of edges to links M_{EL}.
- **Timing** Where latency is defined as the cycle-offset from the initial arrival of data, the maximum latency mismatch (difference) of operands on each node should be minimized as the primary objective. Minimizing latency is secondary. The decision variables here indicate execution time of the vertex and edge T_V and T_E, and the FIFO delay X_E.

Support for Delay Matching: The MILP formulation supports all three forms of delay matching:

(1) **Long Paths:** Automatically through MILP search.

(2) **Passthrough:** We add a set of binary variables $P_{e,n}$ indicating if an edge e is being mapped as a passthrough for a node n. Routing equations (4,5 in table 1) are updated to allow edges to be mapped as the inputs to nodes, *either* if the corresponding vertex is mapped to the node, *or* if the edge and node pair is a passthrough.
(3) **Delay FIFOs:** Integer variables X_E indicate extra delay due to delay FIFOs on each edge. Constraint 14 limits X_E to the FIFO_LEN, multiplied by the number of PEs on that path (1 + number of passthroughs).

4.2 Phases and Overlapping

Each set of responsibilities (mapping, routing, timing) may be solved solved jointly, in independent phases, or with overlapped phases. If scheduling independent (non-overlapped) phases, all variables' values are carried into the next phase. For overlapped phases, only a subset of variable values are carried into the subsequent phase. As an example, while a non-overlapped MR.T algorithm performs mapping and routing, then fixes the associated variables (M_{VN} for mapping and M_{EL} for routing), the phase-overlapped MR.RT fixes only the mapping (M_{VN}) after the first phase.

Objective Functions: To support all types of phases in Figure 6, we require several objective functions, as each phase has different available information:

- **MRT, RT, T** In these phases we minimize the maximum mismatch (modeling expected throughput), plus a small factor for the maximum latency of any path (eg. mismatch + latency / 100), as we optimize latency only as a secondary objective. Because mismatch is the most important objective, we allow this phase to terminate early when a zero-mismatch schedule is found.
- **MR, R** Since mismatch is not available, in these phases we minimize the maximum latency alone. Because performance is generally not latency sensitive, we reduce scheduling time by allowing these phases to terminate early if they reach within 10% of an optimal schedule.
- **M** Here, even latency is not available, as there is no routing. Instead we simply minimize total distance between dependent vertices on the grid, intuitively minimizing the "distortion" of the input graph when mapped to the hardware topology. In an offline phase, we calculate the distance between all pairs of nodes (ie. distance without considering routing contention) with an additional variable D_{VV} (see constraint 3). We use a similar 10% threshold to optimum as the early stopping criterion.

Phase Time Allocation: There is a final issue of allocating time to each scheduling phase, given some maximum desired overall time; the early phases should be limited to leave some time to complete the later phases. Empirically we found that this was only an issue for the MR phase, as it is the most expensive in terms of scheduling time, so we allow the MR phase to have 90% of the overall time.

5 HYBRID STOCHASTIC SCHEDULING

The second principle that this work explores is *hybrid scheduling*, where some scheduling phases are performed by a fast heuristic scheduler to reduce the search space, and other phases are performed through optimization to find a guaranteed best solution. Here we explain the design of a sophisticated heuristic scheduler, and how to integrate with optimization-based scheduling phases, particularly for overlapped responsibilities.

We remark that our goal here is to create an effective heuristic scheduler that performs all three forms of delay-matching, which can be used for effective hybrid scheduling. It is beyond the scope of this work to compare various heuristic scheduling techniques.

5.1 Iterative Stochastic Scheduler

Heuristic Scheduler Considerations: In designing a heuristic scheduling algorithm for use in hybrid and overlapped scheduling, both *solution quality* and *time-to-first-solution* are critical. Solution quality is of course useful because it increases the likelihood of the optimization phase being successful. Reducing the time to first-solution is also useful, because if it is low enough, it enables an iterative approach where the optimization phase can be applied multiple times with different partial solutions from the heuristic.

High-level Heuristic Scheduling Approach: To balance the objectives of time-to-first-solution and solution quality, our approach is to use an iterative stochastic search, where all three scheduling responsibilities (mapping, routing, timing) are considered simultaneously. Essentially, we attempt to schedule the software graph onto the hardware graph multiple times, where in each iteration the scheduling decisions are most-often locally optimal. This approach of relying on mostly locally-optimal decisions leads to nearly-satisfactory, reasonable solutions quickly. The best solution at any point can be used in later scheduling phases.

Stochasticity comes into play to improve solution quality, where each iteration introduces some randomness in the ordering of scheduling decisions, as well as sometimes making locally-non-optimal choices. As better solutions are found, ie. those which have a lower delay-mismatch, scheduling iterations which are determined to have a larger mismatch can be exited early; this essentially bounds and speeds up the search during later iterations.

Algorithm 1: ScheduleIteration(G,H)

```
   input  : Computation Graph G, Hardware Graph H
   output : Schedule S
1  list = RandomTopologicalSort(G)
2  for v ∈ list do
3      best_routing = Nill
4      if do_rand_pick() then
5          while (!success && attempt_more()) do
6              n = randompick(CompatibleNodesLeft(v,S))
7              success = RouteInEdges(v,n,cur_routing)
8          end
9          if success then best_routing = cur_routing
10     else
11         for n ∈ CompatibleNodesLeft(v,S) do
12             cur_routing = Nill
13             success = RouteInEdges(v,n, cur_routing)
14             if success then best_routing = cur_routing
15         end
16     end
17     if best_routing == Nill then return Nill
18     else if mismatch(best_routing) > mismatch(best_schedule)
         then return Nill
19     else IncorperateRouting(S,best_routing)
20 end
21 Procedure RouteInEdges(v, n, cur_routing)
       input  : Vertex v, Node n, Routing cur_routing
       output : Update cur_routing
22     success = true
23     for e ∈ IncommingEdges(v) do
24         success &&= ShortestPath(e,n,cur_routing,H)
25         && r_score(cur_routing) < r_score(best_routing)
26         if !success then break
27     end
28     return success
```

Algorithm Core: At the heart of the algorithm is a list-style scheduler: ScheduleIteration (see Algorithm 1). We first explain its operation without discussing the stochastic aspects (stochastic elements are highlighted in blue). Note that this algorithm is similar to what is described for previous dedicated PE architectures [20, 21], but is updated to perform all three forms of delay-matching.

At the top level, the algorithm iterates over the computation graph in forward topological order. For each instruction vertex, it attempts to route its input dependences on each compatible hardware node (loop on line 11), with procedure RouteInEdges(). This procedure calls a shortest path algorithm on each incoming edge. Forward topological order is chosen because it enables the delay mismatch to be computed at each step. This allows the algorithm to choose the node n with the minimal mismatch. Hardware nodes which have equivalent mismatch are prioritized first by (minimizing) the new value of the maximum delay mismatch, and second by the number of routing resources it would consume to map to them (function r_score on line 25). Once chosen, n's routing will be incorporated into the final schedule for this iteration.

Algorithm 2: HeuristicSchedule(G,H)

```
   input  : Computation Graph G, Hardware Graph H
   output : Schedule S
1  best_schedule=Nill, i=0
2  while keep_going(++i,best_schedule) do
3      new_schedule = ScheduleIteration(G, H)
4      if score(new_schedule) < score(best_schedule) then
5          best_schedule = new_schedule
6      end
7  end
```

Adding Stochasticity: There are two main sources of Stochasticity which we found to be useful in scheduling (highlighted in Algorithm 1). The first is to iterate in a *random* topological order (line 1), to enable the different instruction dependence paths to be prioritized on different iterations. The second source of stochasticity is to sometimes choose an arbitrary compatible node to schedule to, rather than one that minimizes resource use (lines 4-9). The function do_rand_pick() decides how often to perform the stochastic versus the locally-optimal mapping. Based on empirical measurements, we set this parameter to be around 10% of mapping decisions being stochastic. We then simply call ScheduleIteration multiple times, keeping the best current schedule (Algorithm 2).

Because the stochastic algorithm is iterative, the lowest mismatch is used to bound further search. Here, the ScheduleIteration function can be terminated early if it becomes impossible to schedule a node with higher mismatch (line 18). Function mismatch() computes the lowest possible mismatch given the mapping/routing decisions and delay-FIFO length (using max/min arrival times).

Support for Delay Matching: Overall, the stochastic scheduler supports all three forms of delay matching:

(1) **Long Paths:** The random selection of a seemingly arbitrary node can sometimes provide the extra latency for a path that needs to be lengthened.
(2) **Passthrough:** The shortest path routing algorithm allows routes through functional units. To prevent excessive use of node resources as passthroughs, the cost is inversely proportional to the number of not-needed nodes left.
(3) **Delay FIFOs:** At each step of the algorithm, the max/min arrival times are maintained for computing maximum mismatch, which are used to prioritize mapping decisions throughout each iteration.

Algorithm 3: ScheduleHybrid(G,H)

```
input  : Computation Graph G, Hardware Graph H
output : Schedule best_schedule
while time_left() && mismatch(best_schedule) != 0 do
    heur_sched=HeuristicSchedule(G,H)
    new_sched=SchedMILP(G,H,
      init_guess=heur_sched,Mvn=heur_sched.Mvn)
    if score(new_sched) < score(best_schedule) then
        best_schedule = new_sched
    end
end
```

5.2 Hybrid Scheduler Integration

Because our later experiments indicate that the MR phase of the hybrid (MR.RT) scheduler dominates scheduling time, we discuss here how to replace the MR phase with the heuristic phase to create a hybrid scheduler (H.RT).

Our strategy is simply to call the stochastic scheduler for a modest number of iterations, then use this solution both as an initial guess (which aids the traditional branch-and-bound search of MILP solvers by providing a lower bound) and also as the fixed mapping M_{VN} to implement the overlapped scheduling. This process repeats until the algorithm finds a legal schedule where mismatch is zero (see Algorithm 3). We also found that we do not require much time in the RT phase to find a good solution, but sometimes the optimizer "wastes time" by trying to prove optimality. Therefore, we set a timeout of 5 minutes in the RT phase, which allows the algorithm to attempt more initial solutions from the heuristic.

Finally, to improve the likelihood of the optimization scheduler's success, our heuristic's objective ("score") adds a second-order term for the total mismatch across nodes (ie. it optimizes for total mismatch as a secondary objective). This intuitively reduces the total amount of correction required by the optimization scheduler.

6 EVALUATION APPROACH

6.1 Target Architecture

Architecture Overview: For context, we briefly describe the target architecture for performance analysis. The stream-dataflow accelerator [35] (Figure 8) consists of a control core, dedicated-PE CGRA, and stream-engines to coordinate data movement (for scratchpad, memory, recurrence). Data is accessed from memory (cache hierarchy) or scratchpads in decoupled streams from the computation; this combined with wide cache/scratchpad ports and wide buses enable feeding the CGRA fast enough to sustain fully-pipelined execution (provided the scheduling algorithm can achieve a high-quality schedule).

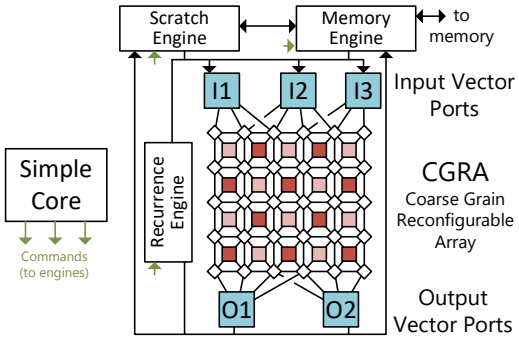

Figure 8: Stream-Dataflow Accelerator

Data flows through the CGRA deterministically with static delays through all paths, foregoing any flow-control internally. Data is fired synchronously into the CGRA when enough is available for one instance of the computation. CGRA firing logic maintains a running tally of whether the last (FIFO_LEN + mismatch) cycles had less than (FIFO_LEN) occurrences, in order to determine if firing a computation instance is legal.

Provisioning: The CGRA has a 64-bit datapath and circuit switched configurable routers. The PE grid size is 5x5, PEs are heterogeneous and support fixed-point and subword SIMD, and (besides for special-function units) are fully pipelined to enable up to II=1 on all computations.

6.2 Evaluation Methodology

Optimization and Simulation Software: Optimization-based formulations use GAMS [1] version 24.7.3 with IBM CPLEX solver, version 12.6.0.0 [2]. For estimating performance, we use a gem5 [7] based cycle-level simulator of stream-dataflow.

Benchmark Computation Graphs: The computation graphs in this study are from the previous evaluation of the stream-dataflow accelerator [35]. These include workloads from MachSuite [49], representing codes for which ASICs are typically built, and deep neural network layers, based on the implementation in DianNao [9]. We also implement a set of linear algebra kernels.

Area Estimation: For the area estimation impact of the delay-FIFOs, we use the baseline accelerator numbers from the stream-dataflow paper [35], and synthesize FIFOs written in Chisel [6] (for consistency with the original work) of different sizes, using a 55nm tech library.

Maximum Scheduling Time: The maximum scheduling time per computation graph that we use for the majority of the results is 20 minutes, and later results show sensitivity to increased scheduling time. Speaking broadly, this is reasonable, considering that accelerated workloads are fewer and generally do not need to be recompiled as often. Also,

FIFO_LEN	Sched. Difficulty	Area (mm^2)	Overhead
2	Very Hard	0.528	9.67%
3	Hard	0.543	12.90%
7	Medium	0.606	25.85%
15	Easy	0.736	52.86%

Table 2: Area and Problem Difficulty

as mentioned earlier, all schedulers may terminate early if the maximum mismatch becomes zero (or if an alternate objective is fulfilled, as in Section 4.2).

7 EVALUATION

The broad objective of the evaluation is to determine the efficacy of phase-overlapping and hybrid scheduling. We formulate this as the following questions:

Q1. Does delay-FIFO size impact accelerator area?
Q2. Time/performance tradeoffs with overlapping?
Q3. Does the initial guess improve scheduling time?
Q4. Intuitively, why does the hybrid approach work?
Q5. Which algorithm is best for a given FIFO_LEN?
Q6. How do the results change with scheduling time?
Q7. Are results similar across benchmarks?
Q8. Does delay-mismatch correlate with performance?

Q1. Does delay-FIFO size impact accelerator area?: The area breakdown of the accelerator with different FIFO sizes is in Table 2. *The configuration with FIFO_LEN of 15 has about 50% area overhead over the design with no delay FIFOs, which is a significant portion.*

Q2. What are the scheduling time and performance tradeoffs with phase overlapping?:

Figure 9 shows the expected throughput (inverse of initiation interval) and scheduling time of all optimization-only schedulers, including non-overlapped (M.RT, MR.T, M.R.T), overlapped (MR.RT), and joint (MRT), for the "easy" problem with FIFO_LEN=3. While the MR.RT algorithm performs the best (solving 17/20 inputs), and M.RT performs second best (solving 16/20 inputs), no algorithms succeed across all workloads. This means that *optimization alone is insufficient to find good schedules quickly.*

Different phases fail for different reasons. The M.R.T and MR.T schedulers fail because the timing phase is difficult to solve independently of the others. The M.RT fails for the same reason, except here it is difficult to perform routing independently of mapping. The joint (MRT) and overlapped (MR.RT) have the ability to consider mapping decisions based on routing constraints, and to consider routing decisions

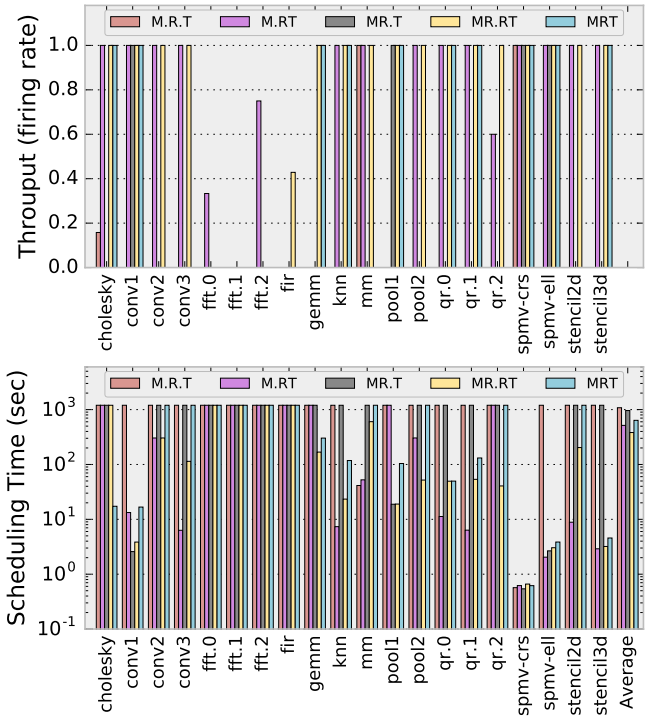

Figure 9: Optimization-only schedulers' throughput and scheduling time (FIFO_LEN=3)

based on timing constraints, but they sometimes fail anyways because the combined mapping and routing is expensive within the solver, and if no solution is found here the algorithm cannot continue.

On the positive side, not only does MR.RT succeed most often, it is also much faster (2× over MRT and 3× over M.R.T on average). This may be surprising, given that the MR.RT scheduler uses more complex phases which have more constraints than solving each phase individually. However, the reduced scheduling time can be explained by the fact that the schedulers are allowed to terminate early if they find an acceptably-optimal solution. Note that sometimes a scheduler will run out of time if it could not find the optimal solution, because it is performing a time-consuming search over the possible decisions to *prove* infeasibility; in such a case it can still return the best known solution at the conclusion of the algorithm.

We also looked at increasing the scheduling time to an hour (data not shown here). The MR.RT scheduler found valid schedules across all workloads, while the M.RT scheduler did not solve any additional inputs. *Overall, making mapping decisions before considering routing is intractable, as is making routing decisions before timing – overlapped scheduling is necessary.*

Q3. Does the initial guess help optimization techniques?: The traditional way of using a heuristic to

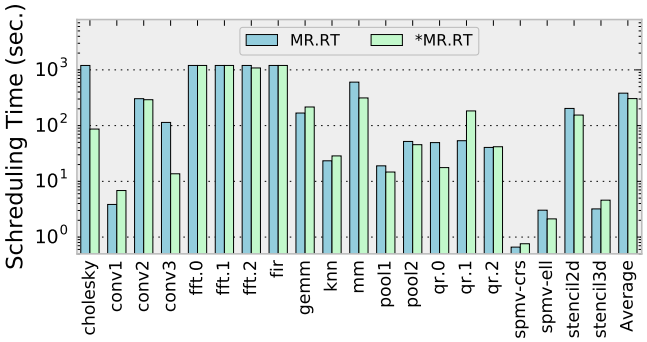

Figure 10: Effect of initial guess (*) on MR.RT.

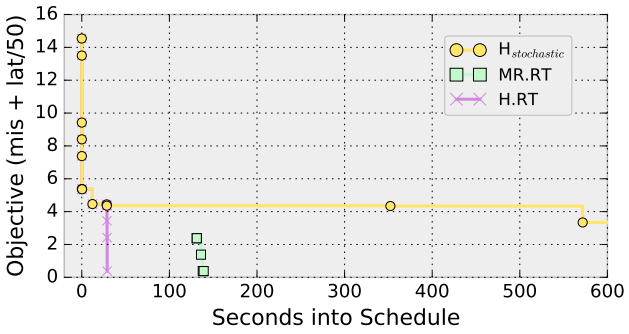

Figure 11: Solution quality achieved over 10 minutes (32-1 Reduction Kernel).

augment an optimization-based scheduler is to use it to generate an initial guess to bound the search space. Figure 10 shows the solution times for each benchmark for MR.RT, for FIFO_LEN=3 and 20 minute scheduling time bound, with and without the initial guess from the heuristic solver (*MR.RT uses an initial guess based on our heuristic scheduler). Sometimes the solution time is slightly worse, because of the additional time to compute the heuristic. However, in several cases the solution time is cut dramatically. Overall, the scheduling time is only slightly better on average, by 25%.

For this reason, we will always assume that the initial guess is employed for all algorithms tested going forwards in the evaluation. That said, as the difference is overall very small, *using the heuristic to generate an initial guess is insufficient alone to reduce scheduling time.*

Q4. Intuitively, why does the hybrid approach reduce scheduling time?: Figure 11 shows the objective function of three schedulers (stochastic heuristic, overlapped, and hybrid) over time. For the stochastic scheduler, it initially quickly finds many solutions of increasing quality. However, it quickly tapers-off and cannot further improve. The overlapped-phase scheduler (MR.RT) has two phases. During the MR phase, no solution is found for two minutes. However, once one is found, the RT phase is entered, and within 3

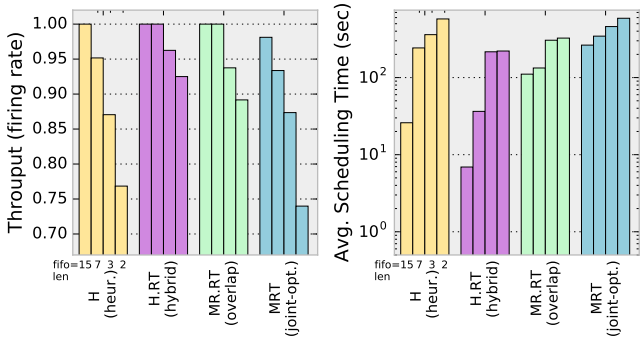

Figure 12: Overall Scheduler Effectiveness. (FIFO_LEN decreases from left to right bar; initial guess from H is enabled.)

seconds the zero-mismatch solution is found. Achieving the best of both worlds, *the hybrid scheduler (H.RT) first utilizes the heuristic scheduler to quickly attain an initial schedule that has a reasonable set of mapping decisions, then uses these to bound the solution space for the RT phase, quickly getting a zero-mismatch solution in the optimization phase.* Overall, this takes only around 30 seconds.

Q5. Which algorithm is best for a given FIFO_LEN?: Figure 12 shows the overall effectiveness of the approaches in this work across all workloads, in terms of the throughput (left) as well as the average scheduling time (right).

The hybrid scheduler (H.RT) performs the best in terms of both throughput and scheduling time on average across all hardware configurations. While the heuristic scheduler (H) performs reasonably well on the easier configurations (FIFO_LEN=15,7), the overlapped-phase schedulers (H.RT and MR.RT) are required for adequate performance on the remaining configurations, due to the effectiveness of the optimization-based RT phase. The overlapped schedulers are also faster, because they can often meet the early-termination condition faster.

Between the two schedulers which use overlapped phases, the hybrid-phased scheduler (H.RT) is generally also faster than its optimization-based counterpart, by as much as 10× for FIFO_LEN=15 (at equivalent throughput), with more modest gains for FIFO_LEN=3 of around 1.5× (and with 4% better throughput).

The hybrid approach is faster because the heuristic can produce a better quality mapping (the input to the RT phase), as it also considers timing constraints as part of its own objective function. Finally, both overlapped schedulers achieve much higher throughput than the state-of-the-art MRT scheduler on both of the two harder configurations (H.RT gets 9% higher throughput for FIFO_LEN=3 and 21% higher throughput for FIFO_LEN=2).

Q6. How do the results change with scheduling time?: Figure 13 shows a similar graph depicting the sensitivity

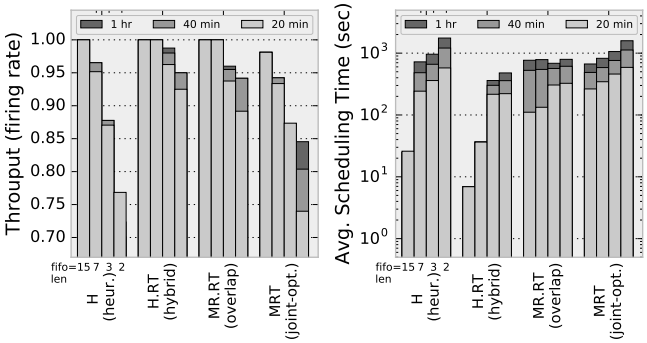

Figure 13: Sensitivity of Scheduler Effectiveness to maximum scheduling time. (FIFO_LEN decreases from left to right bar.).

of throughput and scheduling time to maximum scheduling time. The heuristic scheduler can only rarely make better use of a longer scheduling time, presumably because it does not narrow the search space over time. The joint scheduler also cannot make up for its deficiencies with more time – it fails to reach the average throughput of the overlapped schedulers. Finally, while both the hybrid and optimization-based overlapped schedulers improve with increased scheduling time, the optimization-based scheduler spends much more of its allocated time, even when it does not need to. This is because the MR phase does not reason about timing, and so cannot terminate based on whether the mismatch is zero. *Overall, the hybrid approach is still the best choice, even if more scheduling time is acceptable.*

Q7. How do trends vary across benchmarks?: Figure 14 shows the per-benchmark breakdown of the scheduling time, maximum throughput, and achieved throughput from simulation for the four different schedulers for the "very hard" accelerator configuration (FIFO_LEN=2). The stochastic heuristic (H) and joint (MRT) schedulers are more prone to finding low-throughput schedules, thereby losing significant performance – up to 4× worse on some workloads.

The other two algorithms fare better. In terms of solution quality, the H.RT hybrid scheduler fares just slightly better overall in terms of throughput, as compared to MR.RT. This is because the stochastic scheduler seeds the hybrid with a mapping which is known to have a low total mismatch across all nodes, where the MR.RT scheduler just receives a latency-optimized mapping from its first phase.

Q8. Does delay-mismatch correlate with performance?: The bottom two graphs in Figure 14 show the correlation between the scheduler's view of throughput loss due to delay-mismatch, and the actual performance loss measured by the simulator. Note that we show the overall performance in one bar for `fft` and `qr`, and that the hybrid scheduler achieves up to 2× performance over MRT (on `fir`). Overall, the trends between measured and predicted

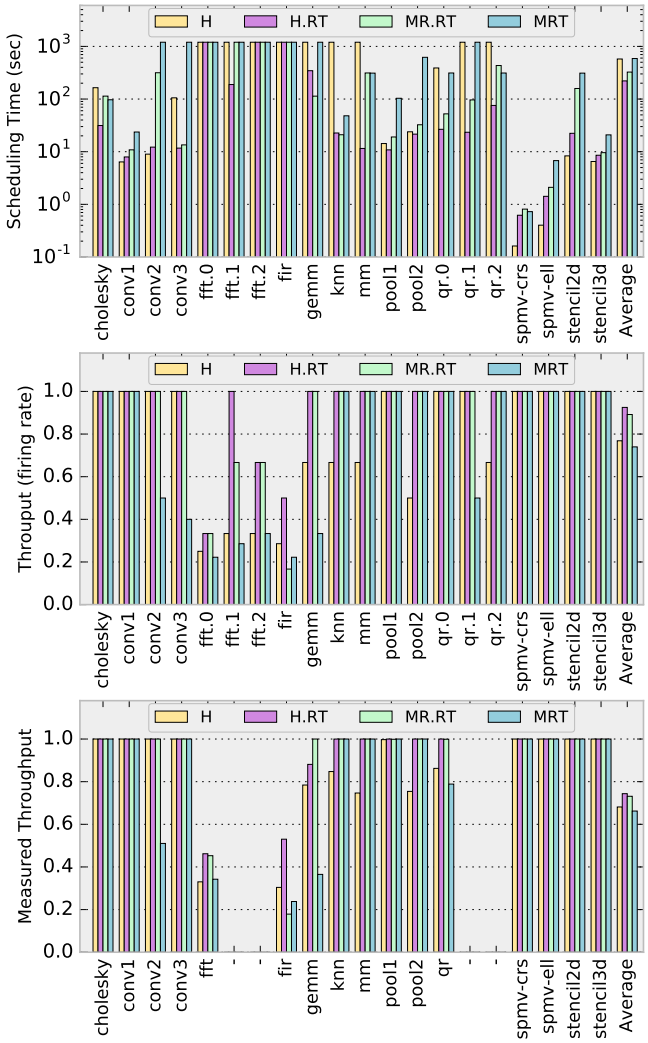

Figure 14: Per-benchmark breakdown of Scheduling Time (Top), scheduler's view of throughput (Middle), and measured throughput relative to best case (bottom) with FIFO_LEN=2.

performance are similar, which *validates the schedulers' abstraction of performance – using maximum delay-mismatch as a proxy for throughput.*

Results Summary: Overall, what we found was that the overlapped-phase (MR.RT) and hybrid (H.RT) schedulers were the most effective, even with highly resource-constrained hardware. The primary difference is that MR.RT produces schedules of lower performance on average (3-4%), and takes longer, especially on easier problems or when the maximum scheduling time is longer.

As far as the overall benefits, the scheduler enables hardware/software codesign. Assuming a desired throughput reduction of less than 5%, the hybrid scheduler can be employed

on a delay-FIFO size of down to 3. This means that the hardware can be designed with 3 delay-FIFO slots instead of 15, leading to 35% area savings.

Compared to the original optimization-based scheduler (MRT), which took an average of 171 seconds for FIFO_LEN 3, and did not always produce a valid schedule, the hybrid scheduler takes on average only 32 seconds (5× speedup), and always produces a high-quality schedule.

8 RELATED WORK

Heuristic Schedulers: There is a long history of heuristic algorithms for resource-exposed architectures. A few examples are the BUG VLIW scheduler [18], its improved Unified Assign and Schedule for clustered VLIWs [40], and the CARS code-generation framework which combines cluster assignment, register allocation, and instruction scheduling [25]. Also, fractional initiation intervals were previously utilized for software pipelining on traditional VonNeumann architectures [3].

Many schedulers have been developed for shared-PE CGRAs for both accelerators and general purpose architectures. A prominent example is the edge-centric modulo scheduler [43] and its extension for sub-cycle scheduling [45]. Another example is the DRESC modulo scheduling algorithm [31], which is based on simulated annealing. Those targeting more general purpose processors include the space-time scheduler [28] for RAW [54], the multi-granular scheduler [32] for Wavescalar [53], and SPS [17] and SPDI [33] for TRIPS [51]. Further work is required to determine if phase-overlapping and hybrid schedulers would be useful for shared-PE designs. Stochastic iterative search techniques also seem plausible for these architectures.

For dedicated-PE schedulers, one example [15] targets a CGRA with PEs arranged in a tree [13]. From the domain of deep neural networks acceleration is MAERI [26], which targets a network structure called the "augmented reduction tree". Also, it can be argued that systolic array scheduling is another example [27, 57]. The dedicated-PE arrays targeted here have a less flexible network than what this work targets.

Optimization-based Schedulers: Optimization-based algorithms also have a rich history in the compiler space, including VonNeumann architectures [24, 41], VLIW processors [19], and multiprocessors [52].

Perhaps the most related work, besides the formulation we build off of here [37, 39], is the MILP scheduling model for the RAW spatial architecture [4]. They consider both ILP-based techniques and heuristics, but do not consider phase-overlapping of responsibilities, or combining heuristics and optimization techniques. The spatial scheduling problem has also been solved using program synthesis techniques [47].

Hardware Synthesis: Optimization techniques have been extensively explored in the past for VLSI and CAD problems [5, 14, 23] for generating application-specific circuits. The nature of these types of problems in the synthesis space in terms of their size (much larger) and types of constraints (typically less constrained) are quite different.

9 CONCLUSION

This work explored the challenge of instruction scheduling on highly-efficient but restrictive dedicated-PE architectures, and evaluated the effectiveness on a recent programmable accelerator. We explored two main principles. The first was overlapping scheduling phases to reduce the search space complexity. The second was integrating heuristic and optimization based schedulers to create a hybrid scheduler, each solving the portion of the problem they are best suited for. These two techniques enabled a faster scheduler by 5×, better throughput, and codesign which could reduce the accelerator's area by 35%.

Overall, the results of this work demonstrate that problem-specific heuristics, as well as optimization based algorithms (which amounts to a branch and bound search) are complimentary, and can be used together. Our insights into CGRA architectures and their interaction with the scheduler can enable aggressive hardware/software codesign, to allow for even more efficient programmable accelerator designs for the specialization era.

REFERENCES

[1] [n. d.]. GAMS, http://www.gams.com/. ([n. d.]).

[2] [n. d.]. IBM ILOG CPLEX, https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex. ([n. d.]).

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. 1995. Software Pipelining. *ACM Comput. Surv.* 27, 3 (Sept. 1995), 367–432. https://doi.org/10.1145/212094.212131

[4] S. Amarasinghe, D. R. Karger, W. Lee, and V. S. Mirrokni. 2002. *A Theoretical and Practical Approach to Instruction Scheduling on Spatial Architectures.* Technical Report. MIT.

[5] S. Amellal and B. Kaminska. 1994. Functional synthesis of digital systems with TASS. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 13, 5 (may 1994), 537 –552.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12).* ACM, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[8] HJ Caulfield, WT Rhodes, MJ Foster, and Sam Horvitz. 1981. Optical implementation of systolic array processing. *Optics Communications* 40, 2 (1981), 86–90.

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[10] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 367–379. https://doi.org/10.1109/ISCA.2016.40

[11] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. 2004. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *MICRO*.

[12] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. 2013. Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity. In *ISLPED*.

[13] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. CHARM: A Composable Heterogeneous Accelerator-rich Microprocessor. In *ISPLED*.

[14] Jason Cong, Karthik Gururaj, Guoling Han, and Wei Jiang. 2009. Synthesis algorithm for application-specific homogeneous processor networks. *IEEE Trans. Very Large Scale Integr. Syst.* 17, 9 (Sept. 2009).

[15] J. Cong, H. Huang, and M. A. Ghodrat. 2016. A scalable communication-aware compilation flow for programmable accelerators. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 503–510. https://doi.org/10.1109/ASPDAC.2016.7428062

[16] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 9–16.

[17] Katherine E. Coons, Xia Chen, Doug Burger, Kathryn S. McKinley, and Sundeep K. Kushwaha. 2006. A spatial path scheduling algorithm for EDGE architectures. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 129–140. https://doi.org/10.1145/1168919.1168875

[18] John R. Ellis. 1985. *Bulldog: a compiler for vliw architectures*. Ph.D. Dissertation.

[19] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. *International Journal of Parallel Programming* 21 (1992), 313–347. Issue 5.

[20] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept. 2012), 38–51. https://doi.org/10.1109/MM.2012.51

[21] Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. In *PACT*.

[22] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*.

[23] Zhining Huang, Sharad Malik, Nahri Moreano, and Guido Araujo. 2004. The design of dynamically reconfigurable datapath coprocessors. *ACM Trans. Embed. Comput. Syst.* 3, 2 (May 2004), 361–384.

[24] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI '02)*. 304–314. https://doi.org/10.1145/512529.512566

[25] Krishnan Kailas, Ashok Agrawala, and Kemal Ebcioglu. 2001. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*. 133–. http://dl.acm.org/citation.cfm?id=580550.876436

[26] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 461–475. https://doi.org/10.1145/3173162.3173176

[27] Monica Sin-Ling Lam. 1987. *A Systolic Array Optimizing Compiler*. Ph.D. Dissertation. Pittsburgh, PA, USA. AAI8814722.

[28] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. 1998. Space-time Scheduling of Instruction-level Parallelism on a Raw Machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, USA, 46–57. https://doi.org/10.1145/291069.291018

[29] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 14–26. https://doi.org/10.1109/HPCA.2016.7446050

[30] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*. Springer, 61–70.

[31] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques* 150, 5 (Sept 2003), 255–61–. https://doi.org/10.1049/ip-cdt:20030833

[32] Martha Mercaldi, Steven Swanson, Andrew Petersen, Andrew Putnam, Andrew Schwerin, Mark Oskin, and Susan J. Eggers. 2006. Instruction scheduling for a tiled dataflow architecture. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS XII)*. 141–150. https://doi.org/10.1145/1168857.1168876

[33] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. 2004. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. 74–84. https://doi.org/10.1109/PACT.2004.26

[34] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. *WaveComputing WhitePaper* (2017).

[35] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 416–429. https://doi.org/10.1145/3079856.3080255

[36] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 27–39. https://doi.org/10.1109/HPCA.2016.7446051

[37] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY,

USA, 495–506. https://doi.org/10.1145/2491956.2462163

[38] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2014. A Scheduling Framework for Spatial Architectures Across Multiple Constraint-Solving Theories. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 2 (Nov. 2014), 30 pages. https://doi.org/10.1145/2658993

[39] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2014. A Scheduling Framework for Spatial Architectures Across Multiple Constraint-Solving Theories. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 2 (Nov. 2014), 30 pages. https://doi.org/10.1145/2658993

[40] Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. 1998. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO 31)*. 308–315. http://dl.acm.org/citation.cfm?id=290940.291004

[41] Jens Palsberg and MpSOC Mayur Naik. 2004. ILP-based Resource-aware Compilation. (2004).

[42] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[43] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*. 166–176. https://doi.org/10.1145/1454115.1454140

[44] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-out Acceleration for Machine Learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 367–381. https://doi.org/10.1145/3123939.3123979

[45] Yongjun Park, Hyunchul Park, and Scott Mahlke. 2009. CGRA Express: Accelerating Execution Using Dynamic Operation Fusion. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*. ACM, New York, NY, USA, 271–280. https://doi.org/10.1145/1629395.1629433

[46] Yongjun Park, Jason Jong Kyu Park, Hyunchul Park, and Scott Mahlke. 2012. Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 84–95. https://doi.org/10.1109/MICRO.2012.17

[47] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[48] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[49] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 110–119.

[50] Roddy Urquhart and Will Moore and Andrew McCabe. 1987. *Systolic Arrays*. Institute of Physics Publishing.

[51] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. 2006. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *MICRO*.

[52] Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. 2007. A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors. In *DATE '07*.

[53] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 291–. https://doi.org/10.1109/MICRO.2003.1253203

[54] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (March 2002), 25–35. https://doi.org/10.1109/MM.2002.997877

[55] J. J. Tithi, N. C. Crago, and J. S. Emer. 2014. Exploiting spatial architectures for edit distance algorithms. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 23–34. https://doi.org/10.1109/ISPASS.2014.6844458

[56] Dani Voitsechov and Yoav Etsion. 2014. Single-graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 205–216. https://doi.org/10.1109/ISCA.2014.6853234

[57] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 29, 6 pages. https://doi.org/10.1145/3061639.3062207