SCALING A RECONFIGURABLE DATAFLOW ACCELERATOR

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yaqi Zhang

May 2020

Abstract

The rise of “dark silicon” in the semiconductor industry due to the end of Dennard scaling started the multicore era in processor design since 2005. Nonetheless, the performance scaling in multicores is soon coming to a limit, which motivates a new class of spatial architecture–reconfigurable dataflow accelerators (RDAs)–that delivers high throughput and energy efficiency in computation to keep up with the performance demand. To adapt to the compute intensity in modern data-analytic workloads, particularly in deep learning, RDAs are increasing to a scale that was unprecedented before. The increase in scale introduces new challenges in on-chip network design to maintain the energy efficiency and scalability of RDAs. Furthermore, managing and using RDAs at this scale also require new strategies in mapping, memory management, and flexible control to saturate compute throughput of the accelerator.

In this talk, I will talk about architectural design and compilation techinques that improves the scaling efficiency of a RDA developed at Stanford–Plasticine. Staring with a static-dynamic hybrid network, we show that the hybrid network can improves energy efficiency while providing guaranteed success in placement and routing. Next, I will talk about compiler techinques that convert applications’ complex control hierarchies, such as nested loops and branch conditions, into a streaming dataflow representation that can be efficiently executed by Plasticine with distributed on-chip resources. The compiler implements (a) a peer-to-peer (p2p) control paradigm inferred from an imperative programming style that minimizes synchronization overhead, and (b) a mapping strategy that decomposes the computation and memory in a program across a distributed heterogeneous resources.

By applying these techniques, we show that Plasticine is able to outperform state of art accelerators, such as GPUs and FPGAs, in both performance and performance/Watt in various dense, sparse, and streaming applications.

Acknowledgements

I would like to thank my mother and the little green men from Mars.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

2 Background **2**

2.1 Execution Schedule of Spatial Architectures 2

2.2 Plasticine 5

2.3 Spatial 5

3 Architecture **7**

3.1 Generic Banknig Support 8

3.2 Plasticine Specialization for RNN Serving 9

3.2.1 Mixed-Precision Support 9

3.2.2 Sizing Plasticine for RNN Serving 12

3.3 On-chip Network 13

3.3.1 Application Characteristics 13

3.3.2 Design Space for Network Architectures 18

3.3.3 Performance, Area, and Energy Modeling 20

3.3.4 Network Architecture Exploration 23

4 Compiler **34**

4.0.1 Native Low-level Programming Interface of Plasticine 35

4.1 Compiler Overview 38

4.2 Distributed Control Flow 40

4.2.1 Synchronization 41

4.2.2 Data-Dependent Control Flow 44

4.3 Program Decomposition 46

4.3.1 Virtual Memory Decomposition 47

4.3.2 Virtual Compute Decomposition 49

4.4 Optimizations 52

4.5 Register Allocation 54

4.6 Debugging and Instrumentation Support 55

5 Conclusions **56**

6 Future Work **57**

A Appendix **58**

Bibliography **59**

List of Tables

2.1 Concurrency level explored by different architectures 2

3.1 Benchmark summary 21

3.2 Network design parameter summary. 26

4.1 Names and definitions used in the solver-based partitioning. 48

4.2 Solver formulation for partitioning*. 48

4.3 Table 4.1 extension for solver-based merging, which is a generalization of the partitioning problem. 48

4.4 Table 4.2 extension for solver-based merging*. The Retiming Partition objective is not used for merging. 48

List of Figures

2.1 Hiearchical pipelining and parallelization on spatial architecture 3
2.2 Average utilization vs. peak compute density tradeoff 4
2.3 High-level performance model of spatial architectures 4
2.4 Example of outer product in Spatial pseudocode. 5

3.1 Plasticine PCU SIMD pipeline and low-precision support. Red circles are the new operations. Yellow circles are the original opertaions in Plasticine. In (d) the first stage is fused $1^{st}, 2^{nd}$ stages, and the second stage is fused 3^{nd}, 4^{th} stages of (b). 10
3.2 Variant configuration of Plasticine for serving RNN. 12
3.3 Physical resource and bandwidth utilization for various applications. 16
3.4 Application communication patterns on pipelined (a,b) and scheduled (c,d) CGRA architectures. (a) and (c) show the activation rate distribution of logical links at runtime. Links sorted by granularity, then rate; darker boxes indicate higher rates. The split between green and pink shows the ratio of logical vector to scalar links. (b) and (d) show the distribution of broadcast link fanouts. 16
3.5 Characteristics of program graphs. 17
3.6 Switch and router power with varying duty cycle. 22
3.7 Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit). 24

3.8 Area breakdown for all network configurations. 24
3.9 Switch and router power with varying duty cycle. 25
3.10 Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit). 25
3.11 Area breakdown for all network configurations. 25
3.12 Performance scaling with increased CGRA grid size for different networks. . . 27
3.13 Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.) 29
3.14 Impact of bandwidth and flow control strategies in switches. 29
3.15 Impact of VC count and flit widths in routers. 29
3.16 Geometric mean improvement for the best network configurations, relative to the worst configuration. 29
3.17 Normalized performance for different network configurations. 30
3.18 (a,d): Normalized performance/watt. (b,e): Percentage of compute and memory PBs utilized for each network configuration. (c,f): Total data movement (hop count). 31

4.1 Performance comparison with V100 GPU 39
4.2 (a) Access dependency graph. (b) Synchronization of two accesses on the same memory. (c) Single-cycle special case. (d) Actors uses local states of controller hierarchy to determine when to send a token. 42
4.3 An example of splitting a memory to serve parallel requesters. 47
4.4 (a) An example of an illegal partition. (b) The cost of a partition. 51

Chapter 1

Introduction

With the end of Dennard Scaling [?], the amount of performance one can extract from a CPU is reaching a limit. To provide general-purpose flexibility, CPU spends the majority of energy on overheads, including dynamic-instruction execution, branch prediction, and a cache hierarchy, and less than 20% of the energy on the actual computation [?]. Even worse, the power wall is limiting the entire multicore family to reach the doubled performance improvement per generation enabled by technology scaling in the past[?].

Chapter 2

Background

2.1 Execution Schedule of Spatial Architectures

The biggest advantage of reconfigurable accelerators, compared to processor-based architectures such as CPUs and GPUs, is the ability to explore pipeline parallelism at multiple granularity.

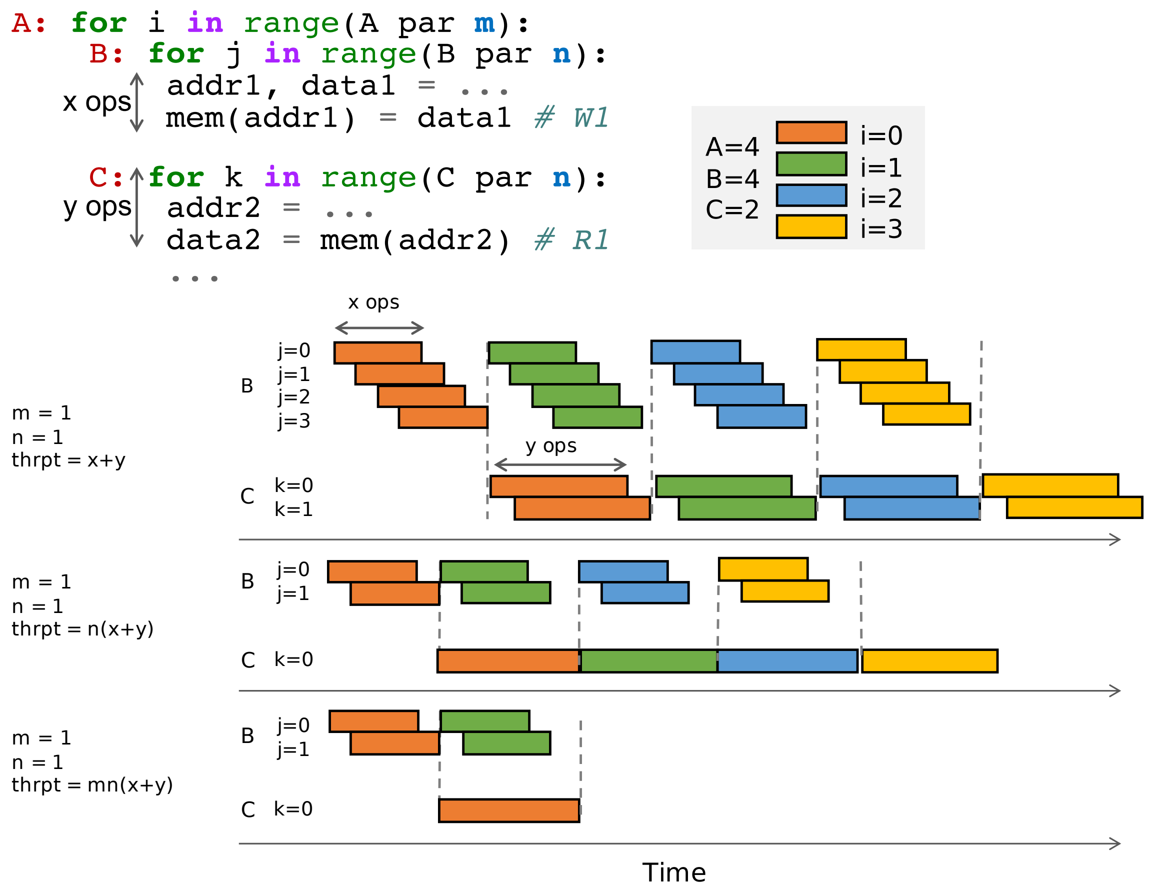
In traditional Von Neumann architectures, a computer consists of a processing unit that performs computation, a memory unit that stores the program states, and a control unit that tracks execution states and fetch the instruction to execute. This computing model inheritely assumes that instructions with in a program are executed in time.

Unlike traditional Von Neumann architectures, which inheritely assumes program is executed in time, reconfigurable data-flow architecutes can statically program

Figure 2.1 shows an example program executed on the

Concurrency Level	Instruction	Data	Task/Kernel
Parallelsim	CPU,RDA	CPU,GPU,RDA	CPU,RDA
Pipelining	RDA	RDA	RDA

Table 2.1: Concurrency level explored by different architectures



```
A: for i in range(A par m):
    B: for j in range(B par n):
         addr1, data1 = ...
         mem(addr1) = data1 # W1

    C: for k in range(C par n):
         addr2 = ...
         data2 = mem(addr2) # R1
         ...
```

Figure 2.1: Hiearchical pipelining and parallelization in spatial architecture. (a) illustrates the runtime and throughput of a hiearchically pipelined and parallelized program on a reconfigurable spatial architecture. At inner level, instructions within each basic block are fine-grained pipelined across iterations of the inner most loop. At outer level, the inner loops are coarse-grained pipeliend across the outer loop iterations. Exploting multiple levels of pipeline parallelism gives a total throughput of $x + y$ operations per cycle. (b) Vectorizing the inner most loops B and C by n increases the throughput to $(x + y)n$. (c) Parallelizing the outer loop A by m furhter increases the throughput to $(x + y)mn$.

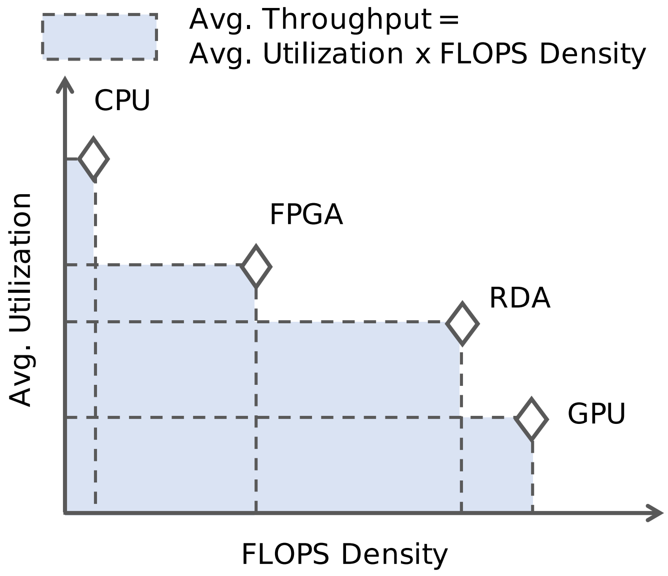


Figure 2.2: Average utilization vs. peak compute density tradeoff among different architectures.

$$\text{thrpt}_{\text{app}} = \min\left(\begin{array}{c}\text{thrpt}_{\text{comp}},\\ \dfrac{\text{comp}}{\text{access}_{\text{off}}}\text{BW}_{\text{off}},\\ \dfrac{\text{comp}}{\text{access}_{\text{on}}}\text{BW}_{\text{on}},\\ \dfrac{\text{comp}}{\text{trans}_{\text{net}}}\text{BW}_{\text{net}}\end{array}\right)$$

Throughput	Proportional To
Compute	P, D
Off-chip Memory	P, D
On-chip Memory	P
On-chip Network	P^{-}, D

Application-specific
Hardware-specific
P: Parallelization factor
D: Pipelining depth

Figure 2.3: High-level performance model of spatial architectures

```
// Host to accelerator register for scalar input with
// user annotated value
val N = ArgIn[Int]; bound(N) = 1024
// 1-D DRAM size in N
val vecA, vecB = DRAM[T](N)
// 2-D DRAM size in NxN
val matC = DRAM[T](N, N)
// Loop unrolling factors
val op1, op2, ip:Int = ...
// Blocking sizes of vecA and vecB
val tsA, tsB:Int = ...

// Accelerator kernel
C0: Accel {
  // C1 is parallelized by op1
  C1: Foreach(min=0, step=tsA, max=N, par=op1){ i =>
    // Allocate 1-D scratchpad size in tsA
    val tileA = SRAM[T](tsA)
    // Load range i to i+tsA of vectorA from off- to
    // on-chip parallelized by ip
    C2: tileA load vecA(i::i+tsA par ip)
    C3: Foreach(min=0, step=tsB, max=N, par=op2) { j =>
      val tileB = SRAM[T](tsB)
      C4: tileB load vecB(j::j+tsB par ip)
      // 2-D scratchpad
      val tileC = SRAM[T](tsA, tsB)
      C5: Foreach(min=0, step=1, max=tsA){ ii =>
        Foreach(min=0, step=1, max=tsB, par=ip) { jj =>
          tileC(ii, jj) = tileA(ii) * tileB(jj)
        }
      }
      // Store partial results to DRAM
      C6: matC(i::i+tsA, j::j+tsB par ip) store tileC
    }
  }
}
```

Figure 2.4: Example of outer product in Spatial pseudocode.

2.2 Plasticine

2.3 Spatial

We use Spatial, an open source domain specific language for reconfigurable accelerators, to target spatial architectures [1]. Spatial describes applications with nested loops and an explicit memory hierarchy that captures data movement on-chip and off-chip. This exposes design parameters that are essential for achieving high performance on spatial architectures, including blocking size, loop unrolling factors, inner-loop pipelining, and coarse-grained pipelining of arbitrarily nested loops. To enable loop-level parallelization and pipelining, Spatial automatically banks and buffers intermediate memories between loops. An example of outer product—element-wise multiplication of two vectors resulting in a matrix—in Spatial is shown in Figure 2.4. For spatial architectures, Design Space Exploration (DSE) of parameters (e.g., *op1*, *op2*, *ip*, *tsA*, *tsB*) is critical to achieve good resource utilization and performance [2].

Chapter 3

Architecture

In this section, we discuss the architectural advancement on top of the original Plasticine architecture introduced in [3]. These architectural additions helps increase the application coverage or improve the mapping strategies of existing applications by supporting new language constructs, data types, and improves the utilizations of the hardware. Specifically, Section 3.1 lay outs the datapath changes in order to support more flexible banking schemes required by general access patterns supported in Spatial; Section 3.2 discusses the hardware specialization and architectural sizing for machine learning applications; **??** provides an extensive study on on-chip network selection reconfigurable spatial architectures.

3.1 Generic Banknig Support

3.2 Plasticine Specialization for RNN Serving

To show efficient execution of the loop and parallel pattern constructs, we map our implementation onto a spatial architecture, Plasticine. **Foreach** at Line $17, 19$ and **Reduce** at Line $22, 23$ are mapped to PCUs on Plasticine. When the application size is small, these constructs are executed using pipelined SIMD lanes within a single PCU. When the application size is large, multiple PCUs can be used to parallelize and pipeline the dot product across PCUs. Element-wise operations can be executed in a deep pipeline formed by chaining multiple PCUs.

To fit an RNN's weights on-chip, we execute our application with low-precision arithmetics. In this section, we propose the necessary micro-architectural changes to support low-precision arithmetics on Plasticine. We also discuss architectural parameter selection for Plasticine to serve RNN applications efficiently.

3.2.1 Mixed-Precision Support

Previous works [4, 5] have shown that low-precision inference can deliver promising performance improvements without sacrificing accuracy. In the context of reconfigurable architectures such as FPGAs, low-precision inference not only increases compute density, but also reduces required on-chip capacity for storing weights and intermediate data.

To support low-precision arithmetics without sacrificing coarse-grained reconfigurability, we introduce two low-precision struct types in Spatial: a tuple of 4 8-bit and 2 16-bit floating-point numbers, `4-float8` and `2-float16` respectively. Both types packs multiple low-precision values into a single precision storage. We support only 8 and 16-bit precisions, which are commonly seen in deep learning inference hardwares. Users can only access values that are 32-bit aligned. This constraint guarantees that the microarchitectual change is only local to the PCU. Banking and DRAM access granularity remains intact from the original design.

Figure 3.1 (a) shows the original SIMD pipeline in a Plasticine PCU. Each FU supports both floating-point and fix-point operations. When mapping applications on Plasticine, the

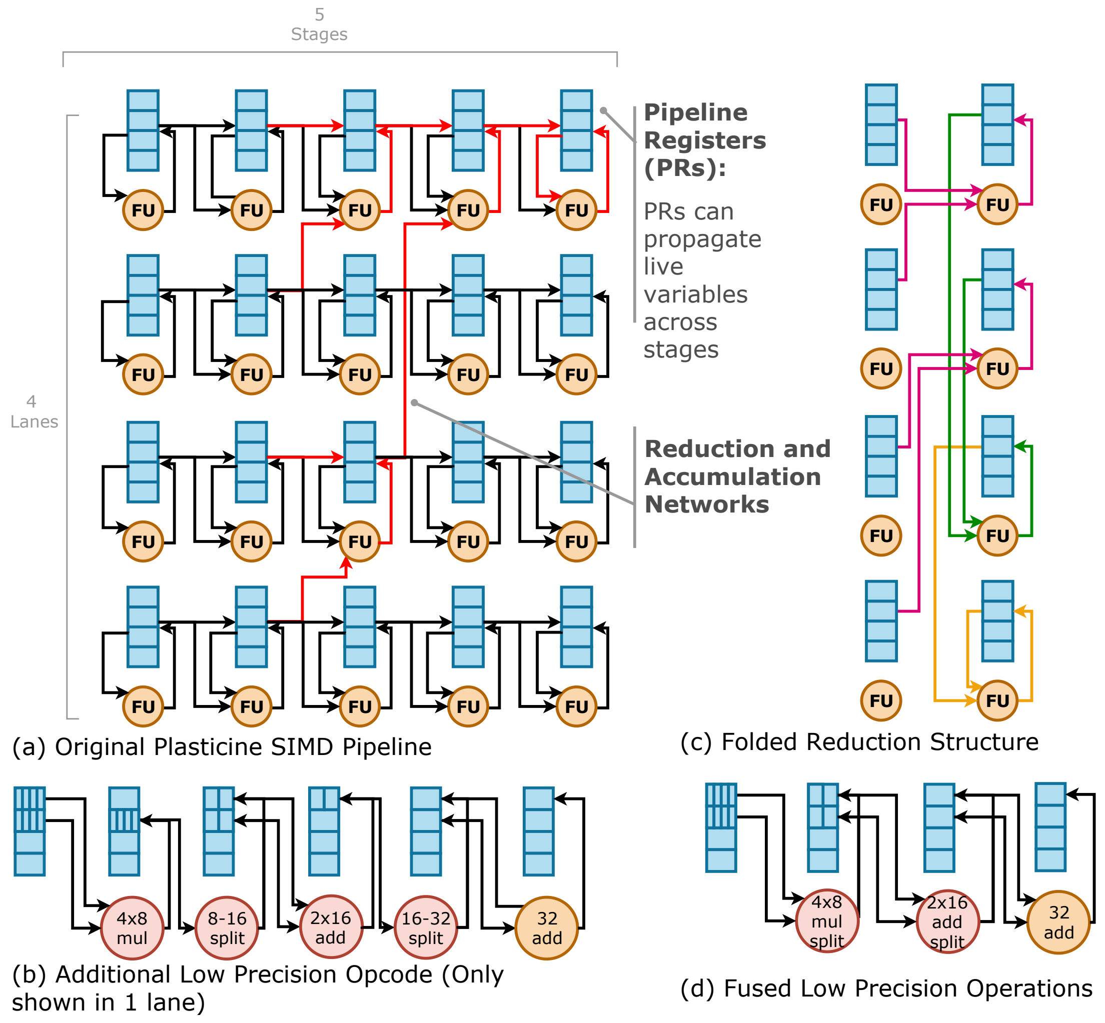

5
Stages
4
Lanes
FU
Pipeline Registers (PRs):
PRs can propagate live variables across stages
Reduction and Accumulation Networks
(a) Original Plasticine SIMD Pipeline
(c) Folded Reduction Structure
4x8 mul
8-16 split
2x16 add
16-32 split
32 add
(b) Additional Low Precision Opcode (Only shown in 1 lane)
4x8 mul split
2x16 add split
(d) Fused Low Precision Operations

Figure 3.1: Plasticine PCU SIMD pipeline and low-precision support. Red circles are the new operations. Yellow circles are the original opertaions in Plasticine. In (d) the first stage is fused $1^{st}, 2^{nd}$ stages, and the second stage is fused 3^{rd}, 4^{th} stages of (b).

inner most loop body is vectorized across the lanes of the SIMD pipeline, and different operations of the loop body are mapped to different stages. Each pipeline stage contains a few pipeline registers (PRs) that allow propagation of live variables across stages. Special cross-lane connections as shown in red in Figure 3.1 enable reduction operations. To support 8-bit element-wise multiplication and 16-bit reduction, we add 4 opcodes to the FU, shown in Figure 3.1 (b). The 1^{st} and 3^{rd} stages are element-wise, low-precision operations that multiply and add 4 8-bit and 2 16-bit values, respectively. The 2^{nd} and 4^{th} stages rearrange low-precision values into two registers, and then pad them to higher precisions. The 5^{th} stage reduces the two 32-bit value to a single 32-bit value using the existing add operation. From here, we can use the original reduction network shown in Figure 3.1 (a) to complete the remaining reduction and accumulates in 32-bit connection.

With 4 lanes and 5 stages, a PCU first reads 16 8-bit values, performs 8-bit multiplication followed by rearrangement and padding, and then produce 16 16-bit values after the second stage. The intermediate values are stored in 2 PRs per lane. Next, 16 16-bit values are reduced to 8 16-bit values and then rearranged to 8 32-bit value in 2 PRs per lane. Then, the element-wise addition in 32-bit value reduces the two registers in each line into 4 32-bit values. These values are fed through the reduction network that completes the remaining reduction and accumulation in two plus one stages.

In a more aggressive specialization, we can fuse the multiply and rearange into the same stage. We also fuse the first low-precision reduction with the next rearange as shown in Figure 3.1 (d). In this way, we can perform the entire low-precision map-reduce in 2 stages in addition to the original full precision reduction. In order to maximize hardware reuse, we assume that it is possible to construct a full precision FU using low-precision FUs. In addition, we observe that the original reduction network in the SIMD lanes could lead to low FU utilization. To improve FU utilization, we fold the entire tree structure in a single stage. Figure 3.1 (c) shows the folded reduction accumulation structure. Specifically, latter reductions in the tree are mapped to earlier stages in the pipeline. In this setup, the entire reduction plus accumulation is still fully pipelined in $\log_2(\#_{LANE}) + 1$ cycles with no structural hazard. With fused reduced-precision multiplication and reduction, and folded

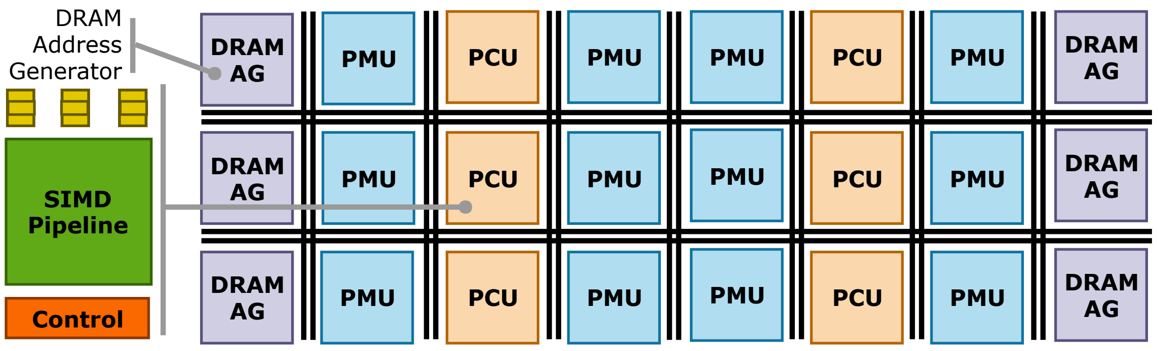


Figure 3.2: Variant configuration of Plasticine for serving RNN.

reduction tree, a PCU is able to perform all map-reduce that accumulates $4\#_{LANE}$ 8-bit values using 4 stages. All the operations are completed in $2 + \log_2(\#_{LANE}) + 1$ cycles.

3.2.2 Sizing Plasticine for RNN Serving

Evaluating an RNN cell containing N hidden units and N input features requires $2N^2$ computations and $N^2 + N$ memory reads. With large N, the compute to memory ratio is 2:1. The original Plasticine architecture uses a checkerboard layout with 1 to 1 ratio between PCU and PMU. A PCU has 6 stages and 16 lanes, and a PMU has 16 banks. This provides a 6:1 ratio between compute resource and on-chip memory read bandwidth. As a result of this layout, on-chip memory read bandwidth becomes the bottleneck for accelerating RNN serving applications. Given that RNNs cover a wide range of important applications, we select a Plasticine configuration tailored for RNN serving. Specifically, we choose a 2 to 1 PMU-PCU ratio with 4 stages in each PCU. Figure 3.2 shows the layout of this Plasticine variant.

3.3 On-chip Network

This section discusses communication characteristics common in applications that have been spatially mapped to CGRAs. Because CGRAs encompass a broad range of architectures, we first describe the abstract machine model of our target CGRA for this study, shown in Figure 3.2. The CGRA contains Physical Blocks (PBs) corresponding to distributed hardware resources, including compute units, scratchpads, and DRAM controllers. The communication between PBs, sent over a reconfigurable network, is purely streaming. Compute PBs have a simple control mechanism: they wait on input data dependencies and stall for backpressure from the network. The network guarantees exactly-once, in-order delivery with variable latency, and communication between PBs can have varying granularities (e.g., 512-bit vector or 32-bit scalar).

In this study, we focus on two categories of CGRA architectures. The first architecture uses pipelining in compute PBs, as shown in Figure 3.2. To provide high throughput, each stage of the pipeline exploits SIMD parallelism, and multiple SIMD operations are pipelined within a PB. Plasticine, a recently proposed CGRA, is an example of a pipelined architecture [3].

The second architecture uses time-scheduled execution, where each PB executes a small loop of instructions (e.g., 6) repeatedly. The scheduling window is small enough that instructions are stored as part of the configuration fabric, without dynamic instruction fetch overhead. This execution model creates more interleaved pipelining across PBs with communication that is tolerant of lower network throughput, which provides an opportunity to share links. Many proposed CGRAs and domain-specific architectures use this *time-scheduled* form of computation, including Brainwave [?] and DaDianNao [?].

3.3.1 Application Characteristics

The requirements of an interconnection network are a function of the communication pattern of the application, underlying CGRA architecture, and compilation process. We identify the following key characteristics of spatially mapped applications:

Vectorized communication

Recent hardware accelerators use large-granularity compute tiles (e.g., vectorized compute units and SIMD pipelines) for SIMD parallelism [3, ?], which improves compute density while minimizing control and configuration overhead. Coarser-grained computation typically increases the size of communication, but glue logic, reductions, and loops with carried dependencies (i.e., non-parallelizable loops) contribute to scalar communications. This variation in communication motivates specialization for optimal area- and energy-efficiency: separate networks for different communication granularities.

Broadcast and incast communication

A key optimization for spatial reconfigurable accelerators is the parallelization of execution across PBs. This parallelization involves unrolling outer loop nests in addition to the vectorization of the inner loop. For neural network accelerators, this corresponds to parallelizing one layer across different channels. By default, pipeline parallelism involves one-to-one communication between dependent stages. However, when a consumer stage is parallelized, the producer sends a one-to-many broadcast to all of its consumers. Similarly, when a producer stage is parallelized, all partial results are sent to the consumer, forming a many-to-one incast link. When both the producer and the consumer are parallelized, the worst case is many-to-many communication, because the parallelized producers may dynamically alternate between parallelized receivers.

Compute to memory communication

To encourage better sharing of on-chip memory capacity, many accelerators have shared scratchpads, either distributed throughout the chip or on its periphery [3, ?, ?]. Because the compute unit has no local memory to buffer temporary results, the results of all computations are sent to memory through the network. This differs from the NoCs used in multiprocessors, where each core has a local cache to buffer intermediate results. Studies have shown that for large-scale multi-processor systems, network latency—not throughput—is the primary performance limiter [?]. For spatial accelerators, however, compute performance is limited by network throughput, and latency is comparatively less important.

Communication-aware compilation

Unlike the dynamic communication of multi-processors, communication on spatial architectures is created statically by compiling and mapping the compute graph onto the distributed PB resources. As the compiler performs optimization passes, such as unrolling and banking, it has static knowledge about communication generated by these transformations. This knowledge allows the compiler to accurately determine which network flows in the transformed design correspond to throughput-critical inner-loop traffic and which correspond to low-bandwidth outer-loop traffic.

We select a mix of applications from domains where hardware accelerators have shown promising performance and energy-efficiency benefits, such as linear algebra, databases, and machine learning. Table 3.1 lists the applications and their data size. Figure 3.3 shows, for each design, which resource limits performance: compute, on-chip memory, or DRAM bandwidth. DotProduct, TPCHQ6, OuterProduct, and BlackScholes are DRAM bandwidth-bound applications. These applications use few on-chip resources to achieve maximum performance, resulting in minimal communication. Lattice (a fast inference model for low-dimensional regression [?]), GDA, Kmeans, SGD, and LogReg are compute-intensive applications; for these, maximum performance requires using as much parallelization as possible. Finally, LSTM, GRU, and LeNet are applications that are limited by on-chip memory bandwidth or capacity. For compute- and memory-intensive applications, high utilization translates to a large interconnection network bandwidth requirement to sustain application throughput.

Figure 3.4(a,b) shows the communication pattern of applications characterized on the pipelined CGRA architecture, including the variation in communication granularity. Compute and on-chip memory-bound applications show a significant amount of high-bandwidth communication (links with almost 100% activity). A few of these high-bandwidth links also exhibit high broadcast fanout. Therefore, a network architecture must provide sufficient

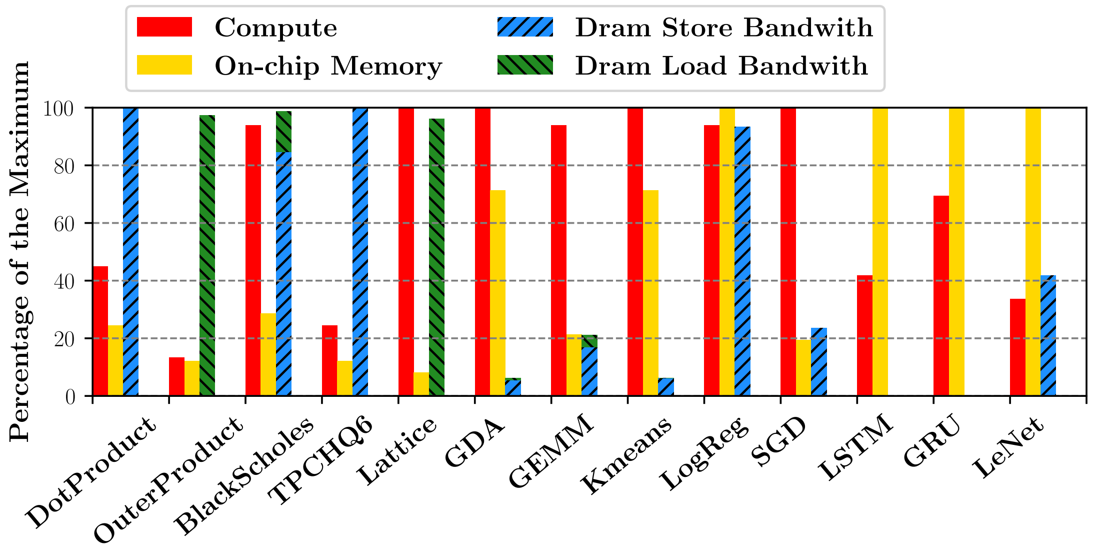
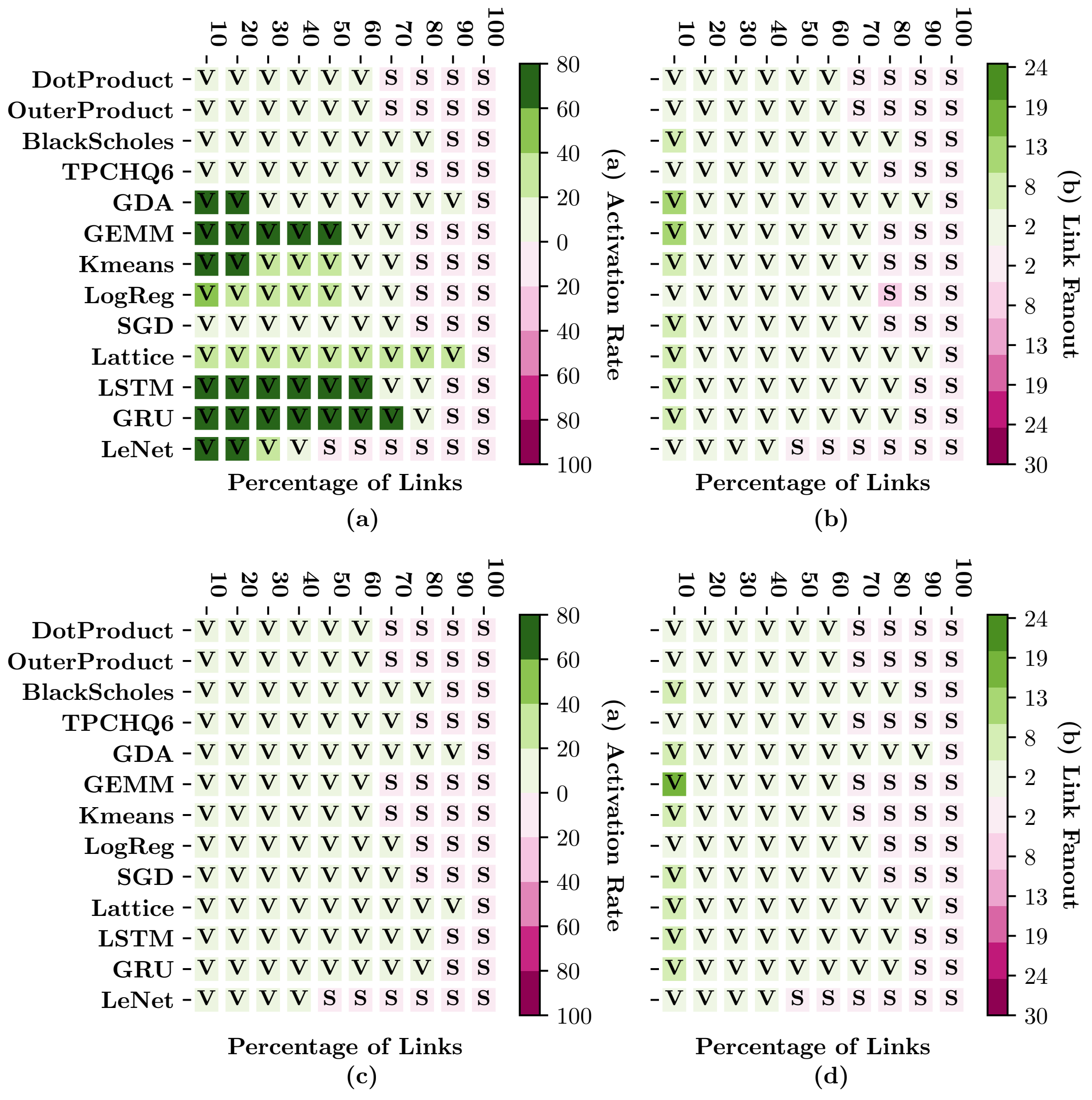


Figure 3.3: Physical resource and bandwidth utilization for various applications.



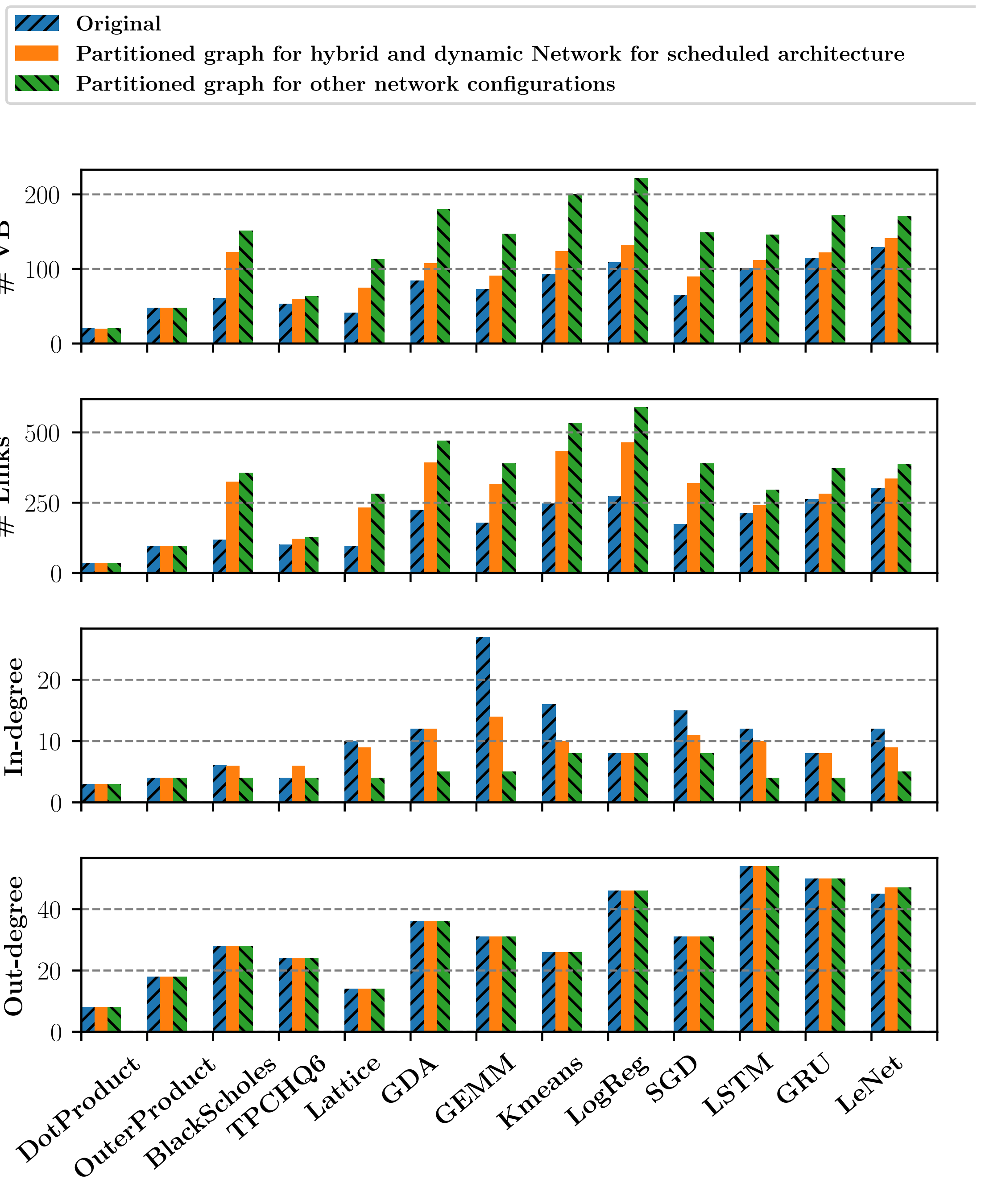


Figure 3.5: Characteristics of program graphs.

bandwidth and efficient broadcasts to sustain program throughput. On the contrary, time-scheduled architectures, shown in Figure 3.4(c,d), exhibit lower bandwidth requirements due to the lower throughput of individual compute PBs. Even applications limited by on-chip resources have less than a 30% firing rate on the busiest logical links; this reveals an opportunity for link sharing without sacrificing performance.

Figure 3.5 shows statistics describing the VB dataflow graph before and after partitioning. The blue bars show the number of VBs, number of logical links, and maximum VB input/output degrees in the original parallelized program; the yellow and green bars show the same statistics after partitioning. Fewer VBs are partitioned for hybrid networks and dynamic networks with the time-scheduled architecture, as explained in Section **??**. The output degree does not change with partitioning because most outputs with a large degree are from broadcast links.

3.3.2 Design Space for Network Architectures

We start with several statically allocated network designs, where each SIMD pipeline connects to several switches, and vary flow control strategies and network bisection bandwidth. In these designs, each switch output connects to exactly one switch input for the duration of the program. We then explore a dynamic network, which sends program data as packets through a NoC. The NoC uses a table-based routing scheme at each router to allow for arbitrary routes and tree-based broadcast routing. Finally, we explore the benefits of specialization by evaluating design points that combine several of these networks to leverage the best features of each.

Static networks

We explore static network design points along three axes. First, we study the impact of flow-control schemes in static switches. In credit-based flow control [**?**], the source and destination PBs coordinate to ensure that the destination buffer does not overflow. For this design point, switches only have a single register at each input, and there is no backpressure between switches. The alternate design point uses a skid-buffered queue with two entries at each switch; using two entries enables per-hop backpressure and accounts for a one-cycle delay in stalling the upstream switch. At full throughput, the receiver will consume data as it is sent and no queue will ever fill up. The second axis studied is the bandwidth, and therefore routability, of the static network. We vary the number of connections between switches in each direction, which trades off area and energy for bandwidth. Finally, we explore specializing static links: using a separate scalar network to improve routability at a low cost.

Dynamic networks

Our primary alternate design is a dynamic NoC using per-hop virtual channel flow control. Routing and Virtual Channel (VC) assignment are table-based: the compiler performs static routing and VC allocation, and results are loaded as a part of the routers' configurations at runtime. The router has a separable, input-first VC and switch allocator with a single iteration and speculative switch allocation [?]. Input buffers are sized just large enough (3 entries) to avoid credit stalls at full throughput. Broadcasts are handled in the network with duplication occurring at the last router possible to minimize energy and congestion. To respect the switch allocator's constraints, each router sends broadcasts to output ports sequentially and in a fixed order. This is because the switch allocator can only grant one output port per input port in every cycle, and the RTL router's allocator does not have sufficient timing slack to add additional functionality. We also explore different flit widths on the dynamic network, with a smaller bus taking multiple cycles to transmit a packet.

Because CGRA networks are streaming—each PB pushes the result to the next PB(s) without explicit request—the network cannot handle routing schemes that may drop packets; otherwise, application data would be lost. Because packet ordering corresponds directly to control flow, it is also imperative that all packets arrive in the order they were sent; this further eliminates adaptive or oblivious routing from consideration. We limit our study of dynamic networks to statically placed and routed source routing due to these architectural constraints. PBs propagate backpressure signals from their outputs to their inputs, so they must be considered as part of the network graph for deadlock purposes [?]. Furthermore, each PB has fixed-size input buffers; these are far too small to perform high-throughput, end-to-end credit-based flow control in the dynamic network for the entire program [?]. Practically, this means that no two logical paths may be allowed to conflict at *any* point in the network; to meet this guarantee, VC allocation is performed to ensure that all logical paths traversing the same physical link are placed into separate buffers.

Hybrid networks

Finally, we explore hybrids between static and dynamic networks that run each network in parallel. During static place and route, the highest-bandwidth logical links from the program graph are mapped onto the static network; once the static network is full, further links are mapped to the dynamic network. By using compiler knowledge to identify the relative importance of links—the link fanout and activation factor—hybrid networks can sustain the throughput requirement of most high-activation links while using the dynamic network for low-activation links.

3.3.3 Performance, Area, and Energy Modeling

We use a cycle-accurate simulator to model the pipeline and scheduling delay for the two types of architectures, integrated with DRAMSim [?] to model DRAM access latency. For static networks, we model a distance-based delay for both credit-based and per-hop flow control. For dynamic networks, we integrate our simulator with Booksim [?], adding support for arbitrary source routing using look-up tables. Finally, to support efficient multi-casting in the dynamic network, we modify Booksim to duplicate broadcast packets at the router where their paths diverge. At the divergence point, the router sends the same flit to multiple output ports over multiple cycles. We assume each packet carries a unique ID that is used to look up the output port and next VC in a statically generated routing table, and that the ID is roughly the same size as an address. When the packet size is greater than the flit size, the transmission of a single packet takes multiple cycles.

Benchmark	**Description**	**Data Size**
DotProduct	Inner product	1048576
OuterProduct	Outer product	1024
BlackScholes	Option pricing	1048576
TPCHQ6	TPC-H query 6	1048576
Lattice	Lattice regression [?]	1048576
GDA	Gaussian discriminant analysis	127×1024
GEMM	General matrix multiply	$256 \times 256 \times 256$
Kmeans	K-means clustering	k=64, dim=64, n=8192, iter=2
LogReg	Logistic regression	8192×128, iter=4
SGD	Stochastic gradient descent for a single layer neural network	16384×64, epoch=10
LSTM	Long short term memory recurrent neural network	1 layer, 1024 hidden units, 10 time steps
GRU	Gated recurrent unit recurrent neural network	1 layer, 1024 hidden units, 10 time steps
LeNet	Convolutional neural network for character recognition	1 image

Table 3.1: Benchmark summary

Area and power

To efficiently evaluate large networks, we start by characterizing the area and power consumption of individual routers and switches used in various network configurations. The total area and energy are then aggregated over all switches and routers in a particular network. We use router RTL from the Stanford open source NoC router [?] and our own parameterized switch implementation. We synthesize using Synopsys Design Compiler with a 28 nm technology library and clock-gating enabled, meeting timing at a 1 GHz clock frequency. Finally, we use Synopsys PrimeTime to back-annotate RTL signal activity to the post-synthesis switch and router designs to estimate gate-level power.

We found that power consumption can be broken into two types: inactive power consumed when switches and routers are at zero-load (P_{inactive}, which includes both dynamic and static power), and active power. The active power, as shown in Section 3.3.3, is proportional to the amount of data transmitted. Because power scales linearly with the amount of data movement, we model the marginal energy to transmit a single flit of data (flit energy,

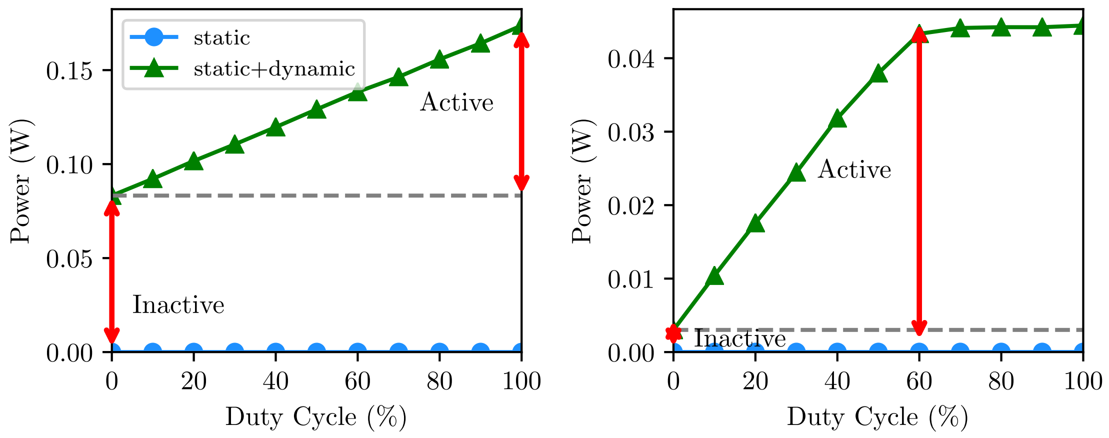


Figure 3.6: Switch and router power with varying duty cycle.

E_{flit}) by dividing active energy by the number flits transmitted in the testbench:

$$E_{\text{flit}} = \frac{(P - P_{\text{inactive}})\, T_{\text{testbench}}}{\#\text{flit}} \tag{3.1}$$

While simulating an end-to-end application, we track the number of flits transmitted at each switch and router in the network, as well as the number of switches and routers allocated by place and route. We assume unallocated switches and routers are perfectly power-gated, and do not consume energy. The total network energy for an application on a given network (E_{net}) can be computed as:

$$E_{\text{net}} = \sum_{\text{allocated}} P_{\text{inactive}} T_{\text{sim}} + E_{\text{flit}} \#\text{flit}, \tag{3.2}$$

where P_{inactive}, E_{flit}, and #flit are tabulated separately for each network resource.

Figure 3.9 shows that switch and router power scale linearly with the rate of data transmission, but that there is non-zero power at zero-load. For simulation, the duty cycle refers to the amount of offered traffic, not accepted traffic. Because our router uses a crossbar without speedup [?], the testbench saturates the router at 60% duty cycle when providing uniform random traffic. Nonetheless, router power still scales linearly with accepted traffic.

A sweep of different switch and router parameters is shown in Figure 3.10. Subplots (d,e,f) show the energy necessary to transmit a single bit through a switch or router. Subplot (a) shows the roughly quadratic scaling of switch area with the number of links between adjacent switches. Vector switches scale worse with increasing bandwidth than scalar switches, mostly due to increased crossbar wire load. At the same granularity, a router consumes more energy a switch to transmit a single bit of data, even though the overall router consumes less power (as shown in Figure 3.9); this is because the switch has a higher throughput than the router. The vector router has lower per-bit energy relative to the scalar router because it can amortize the cost of allocation logic, whereas the vector switch has higher per-bit energy relative to the scalar switch due to increased capacitance in the large crossbar. Increasing the number of VCs or buffer depth per VC also significantly increases router area and energy, but reducing the router flit width can significantly reduce router area.

Overall, these results show that scaling static bandwidth is cheaper than scaling dynamic bandwidth, and a dynamic network with small routers can be used to improve link sharing for low bandwidth communication. We also see that a specialized scalar network, built with switches, adds negligible area compared to and is more energy efficient than the vector network. Therefore, we use a static scalar network with a bandwidth of 4 for the remainder of our evaluation, except when evaluating the pure dynamic network. The dynamic network is also optimized for the rare instances when the static scalar network is insufficient. When routers transmit scalar data, the high bits of data buffers are clock-gated, reducing energy as shown in (f). Figure 3.11 summarizes the area breakdown of all the network configurations that we evaluate.

3.3.4 Network Architecture Exploration

We evaluate our network configurations in five dimensions: performance (perf), performance per network area (perf/area), performance per network power (perf/watt), network area efficiency (1/area), and network power efficiency (1/power). Among these metrics, performance is the most important: networks only consume a small fraction of the overall accelerator

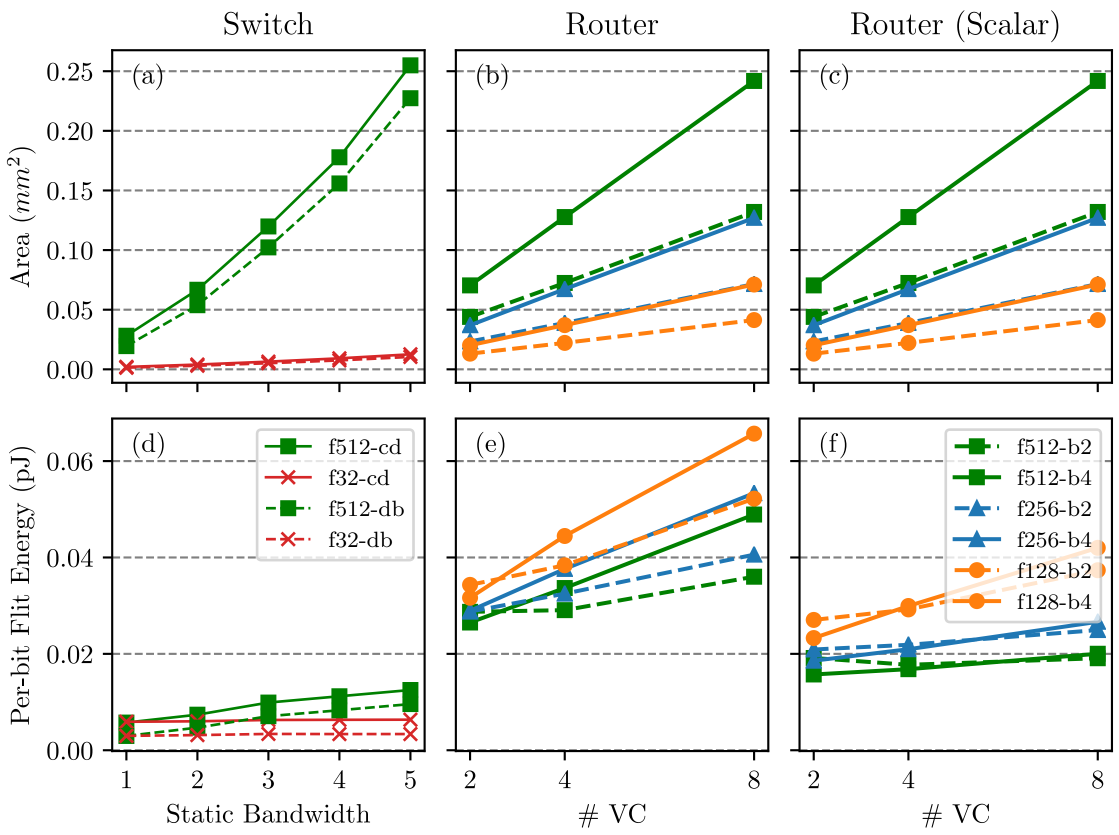


Figure 3.7: Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit).

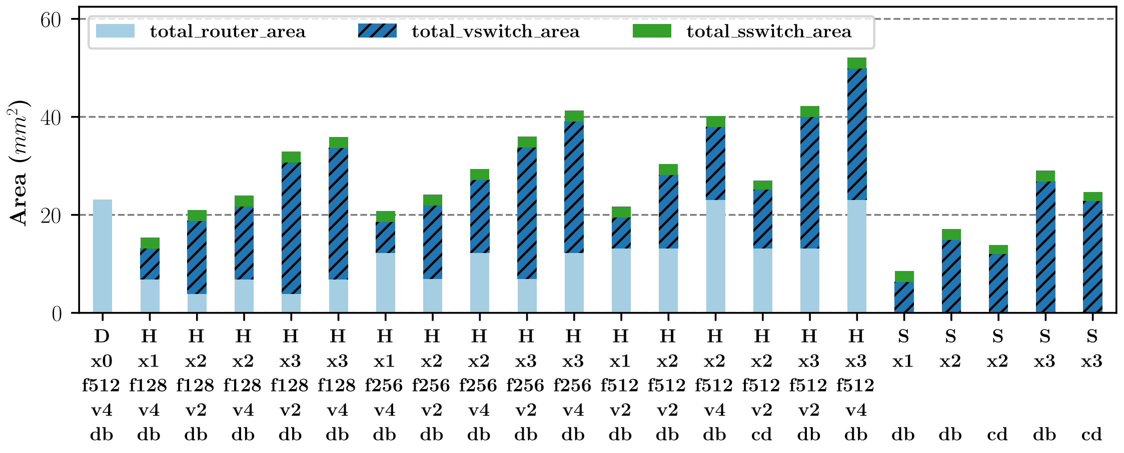


Figure 3.8: Area breakdown for all network configurations.

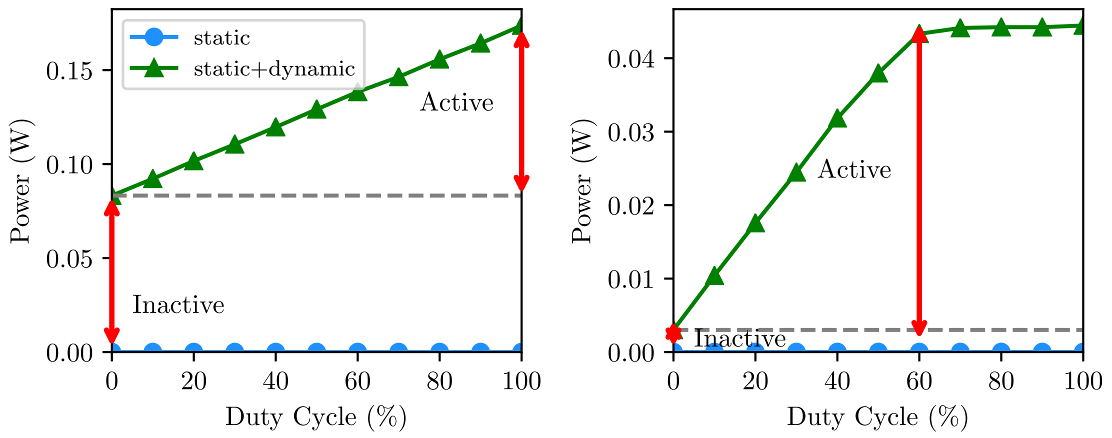


Figure 3.9: Switch and router power with varying duty cycle.

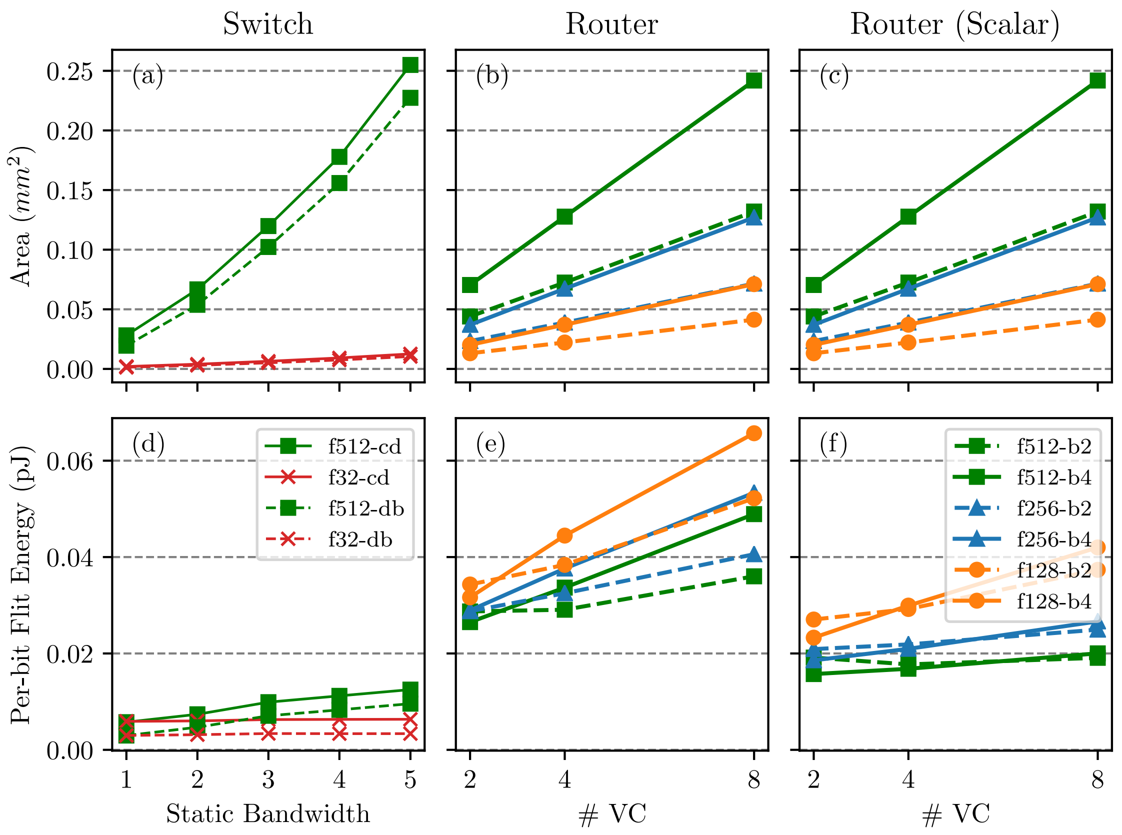
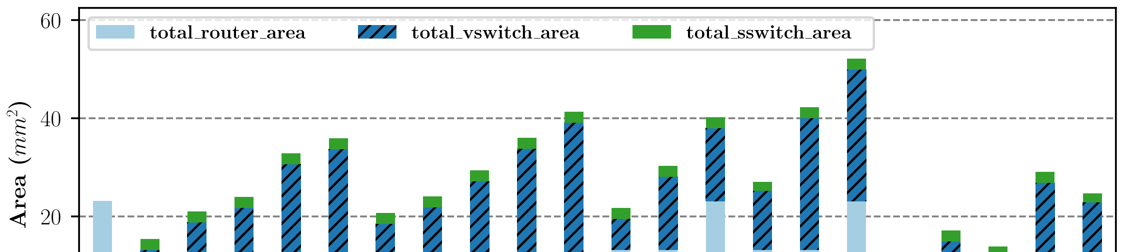


Figure 3.10: Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit).



Notation	Description
[S,H,D]	Static, hybrid, and dynamic network
x#	Static bandwidth on vector network (#links between switches)
f#	Flit width of a router or vector width of a switch
v#	Number of VC in router
b#	Number of buffers per VC in router
[db,cd]	Buffered vs. credit-based flow control in switch

Table 3.2: Network design parameter summary.

area and energy (roughly 10-20%). Because the two key advantages of hardware accelerators are high throughput and low latency, we filter out a network design point if it introduces more than 10% performance overhead. This is calculated by comparing to an ideal network with infinite bandwidth and zero latency.

For metrics that are calculated per application, such as performance, performance/watt, and power efficiency, we first normalize the metric with respect to the worst network configuration for that application. For each network configuration, we present a geometric mean normalized across all applications. For all of our experiments, except Section 3.3.4, we use a network size of 14×14 end-point PBs. All vector networks use a vectorization factor of 16 (512 bit messages).

Bandwidth scaling with network size

Figure 3.12 shows how different networks allow several applications to scale to different numbers of PBs. For IO-bound applications (BlackScholes and TPCHQ6), performance does not scale with additional compute and on-chip memory resources. However, the performance of compute-bound applications (GEMM and SGD) improves with increased resources, but plateaus at a level that is determined by on-chip network bandwidth. This creates a trade-off in accelerator design between highly vectorized compute PBs with a small network—which would be underutilized for non-vectorized problems—and smaller compute PBs with limited performance due to network overhead. For more finely grained compute PBs, both more

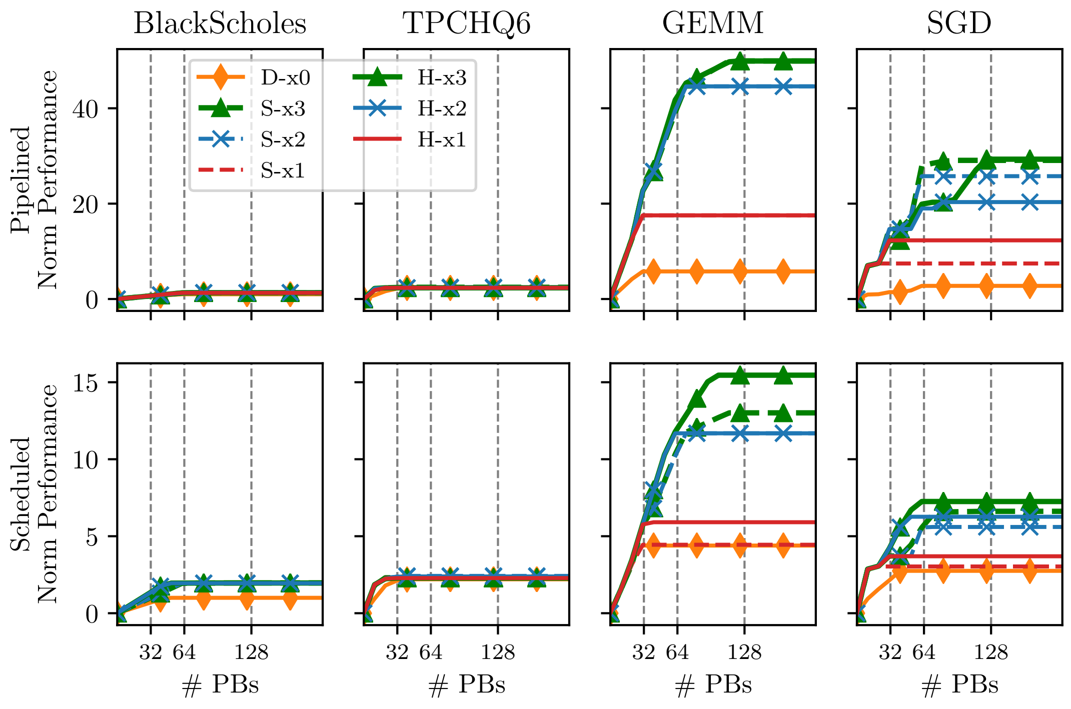


Figure 3.12: Performance scaling with increased CGRA grid size for different networks.

switches and more costly (higher-radix) switches must be employed to meet application requirements.

The scaling of time-scheduled accelerators (bottom row) is much less dramatic than that of deeply pipelined architectures (top row). Although communication between PBs in these architectures is less frequent, the scheduled architecture must use additional parallelization to match the throughput of the pipelined architecture; this translates to larger network sizes.

For pipelined architectures, both hybrid and static networks provide similar scaling with the same static bandwidth: the additional bandwidth from the dynamic network in hybrid networks does not provide additional scaling. This is mostly due to a bandwidth bottleneck between a PB and its router, which prevents the PB from requesting multiple elements per cycle. Hybrid networks tend to provide better scaling for time-scheduled architectures; multiple streams can be time multiplexed at each ejection port without losing performance.

Bandwidth and flow control in switches

In this section, we study the impact of static network bandwidth and flow control mechanism (per-hop vs. end-to-end credit-based). On the left side of Figure 3.14, we show that increased static bandwidth results in a linear performance increase and a superlinear increase in area and power. As shown in Section 3.3.4, any increase in accelerator size must be coupled with increased network bandwidth to effectively scale performance. This indicates that network overhead will increase with the size of an accelerator.

The right side of Figure 3.14 shows that, although credit-based flow control reduces the amount of buffering in switches and decreases network area and energy, application performance is significantly impacted. This is the result of imbalanced data-flow pipelines in the program: when there are parallel long and short paths over the network, there must be sufficient buffer space on the short path equal to the product of throughput and the difference in latency. Because performance is our most important metric, credit-based flow control is not feasible, especially because the impact of bubbles increases with communication distance, and therefore network size.

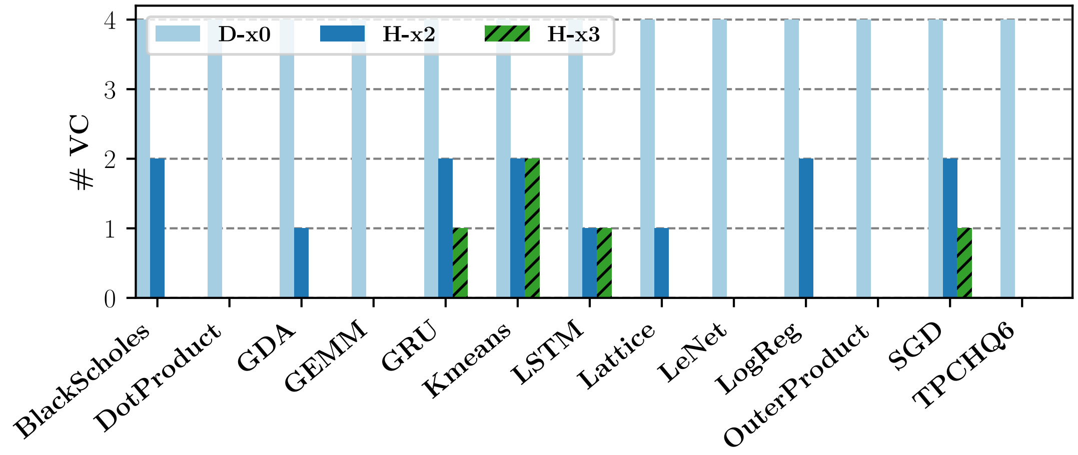


Figure 3.13: Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.)

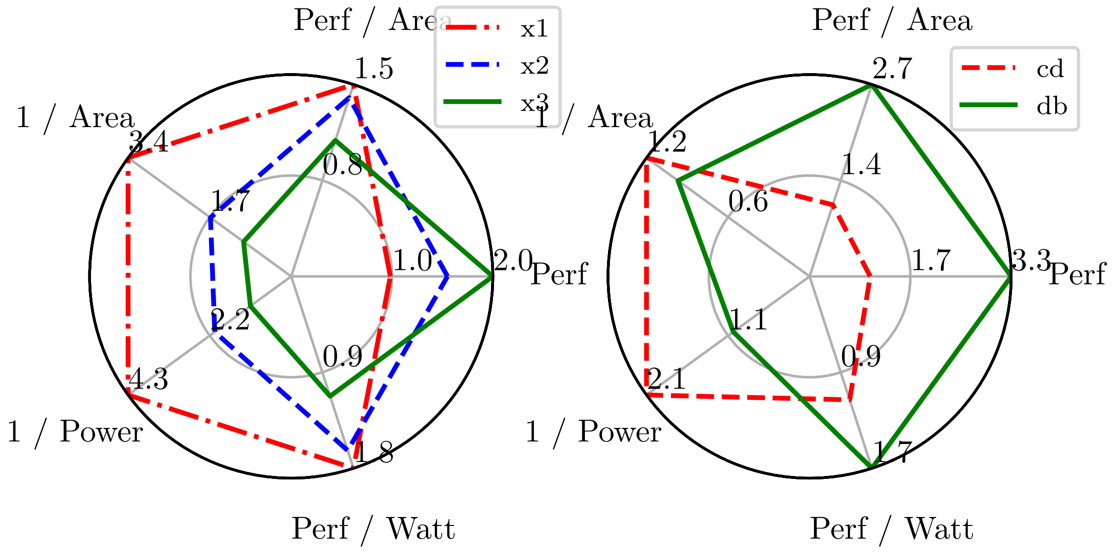
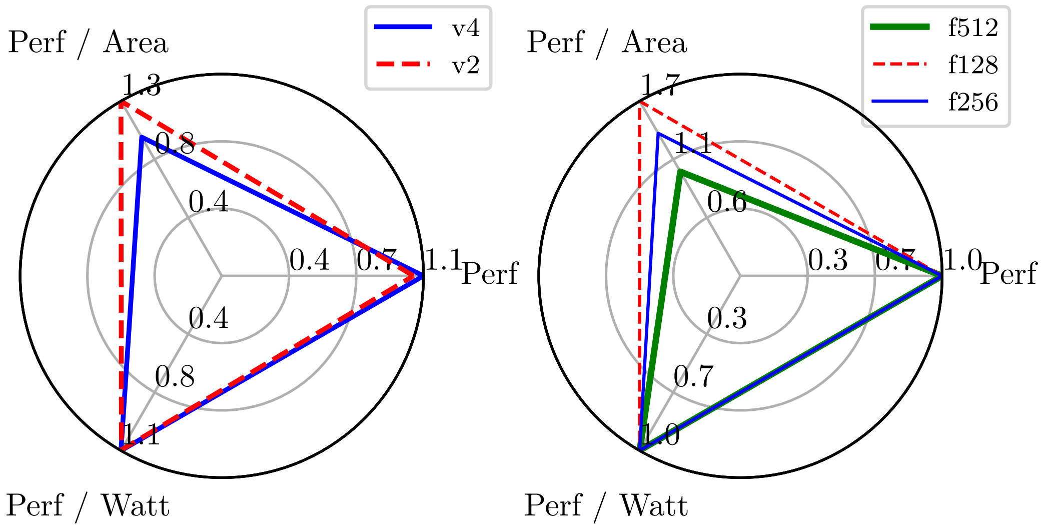


Figure 3.14: Impact of bandwidth and flow control strategies in switches.



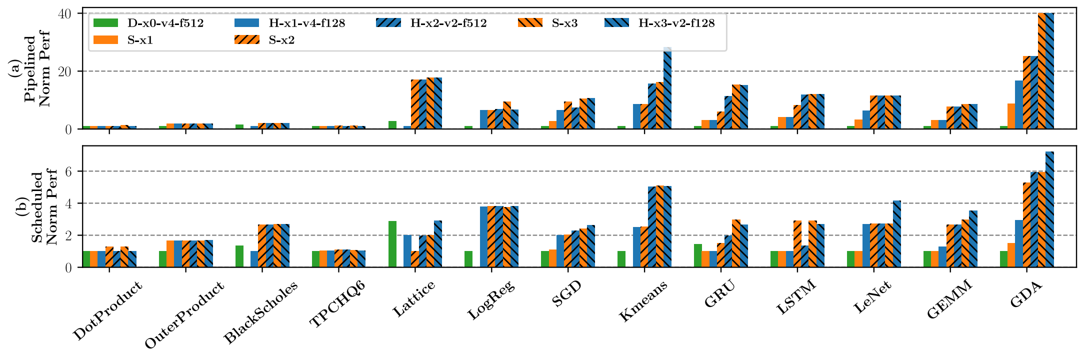


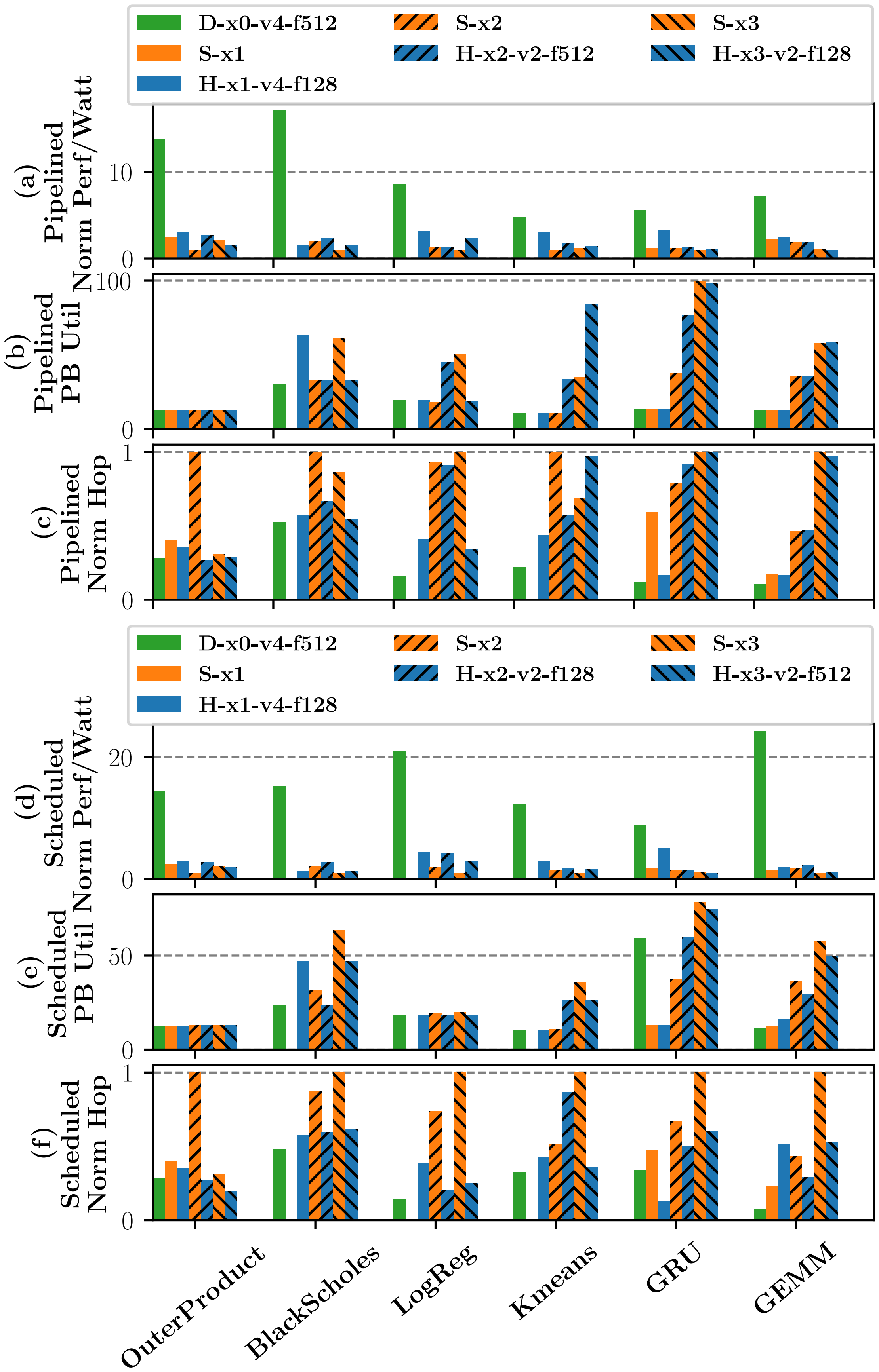
Figure 3.17: Normalized performance for different network configurations.

VC count and reduced flit width in routers

In this experiment, we study the area-energy-performance trade-off between routers with different VC counts. As shown in Section 3.3.3, using many VCs increases both network area and energy. However, using too few VCs may force roundabout routing on the dynamic network or result in VC allocation failure when the network is heavily utilized. Nonetheless, the left side of Figure 3.15 shows minimal performance improvement from using more VCs.

Therefore, for each network design, we use a VC count equal to the maximum number of VCs required to map all applications to that network. Figure 3.13 shows that the best hybrid network configurations with 2x and 3x static bandwidth require at most 2 VCs, whereas the pure dynamic network requires 4 VCs to map all applications. Because dynamic network communication is infrequent, hybrid networks with fewer VCs provide both better energy and area efficiency than networks with more VCs, even though this constrains routing on the dynamic network.

We also explore the effects of reducing dynamic network bandwidth by using smaller routers; as shown in Section 3.3.3, routers with smaller flits have a much smaller area. Ideally, we could scale static network bandwidth while using a low-bandwidth router to provide an escape path and reduce overall area and energy overhead. The right side of Figure 3.15 shows that, for a hybrid network, reducing flit width improves area efficiency with minimal performance loss.


D-x0-v4-f512
S-x1
H-x1-v4-f128
S-x2
H-x2-v2-f512
S-x3
H-x3-v2-f128
(a) Pipelined Norm Perf/Watt
(b) Pipelined PB Util
(c) Pipelined Norm Hop
D-x0-v4-f512
S-x1
H-x1-v4-f128
S-x2
H-x2-v2-f128
S-x3
H-x3-v2-f512
(d) Scheduled Norm Perf/Watt
(e) Scheduled PB Util
(f) Scheduled Norm Hop
OuterProduct
BlackScholes
LogReg
Kmeans
GRU
GEMM

Static vs. hybrid vs. dynamic networks

Figure 3.17 shows the normalized performance for each application running on several network configurations. For some applications, the bar for S-x1 is missing; this indicates that place and route failed for all unrolling factors. For DRAM-bound applications, the performance variation between different networks is trivial because only a small fraction of the network is being used. In a few cases (Kmeans and GDA), hybrid networks provide better performance due to slightly increased bandwidth. For compute-bound applications, performance primarily correlates with network bandwidth because more bandwidth permits a higher parallelization factor.

The highest bandwidth static network uses the most PBs, as shown in Figures 3.18(b,e), because it permits more parallelization. It also has more data movement, as shown in (c,f), because PBs can be distributed farther apart. Due to bandwidth limitations, low-bandwidth networks perform best with small unrolling factors—they are unable to support the bisection bandwidth of larger program graphs. This is evident in Figures 3.18(b,e), where networks D-x0-v4-f512 and S-x2 have small PB utilizations.

With the same static bandwidth, most hybrid networks have better energy efficiency than the corresponding pure static networks, even though routers take more energy than switches to transmit the same amount of data. This is a result of allowing a small amount of traffic to escape onto the dynamic network: with the dynamic network as a safety net, static place and route tends to converge to better placements with less overall communication. This can be seen in Figures 3.18(c,f), where most static networks have larger hop counts than the corresponding hybrid network; hop count is the sum of all runtime link traversals, normalized per-application to the network configuration with the most hops. Subplots (e,f) show that more PBs are utilized with static networks than hybrid networks. This is because the compiler imposes less stringent IO constraints on PBs when partitioning for the hybrid network (as explained in Section **??**), which results in fewer PBs, less data movement, and greater energy efficiency for hybrid networks.

In Figure 3.16, we summarize the best perf/watt and perf/area (among network configurations with $<10\%$ performance overhead) for pipelined and scheduled CGRA architectures. Pure dynamic networks are not shown because they perform poorly due to insufficient bandwidth. On the pipelined CGRA, the best hybrid network provides a 6.4x performance increase, 2.3x better energy efficiency, and a 6.9x perf/area increase over the worst network configuration. The best static network provides 7x better performance, 1.2x better energy efficiency, and 6.3x better perf/area. The hybrid network gives the best perf/area and perf/watt, with a small degradation in performance when compared to the static network. On the time-scheduled CGRA, both static and hybrid networks have an 8.6x performance improvement. The hybrid network gives a higher perf/watt improvement at 2.2x, whereas the static network gives a higher perf/area improvement at 2.6x. Overall, the hybrid networks deliver better energy efficiency with shorter routing distances by allowing an escape path on the dynamic network.

Chapter 4

Compiler

In this section, we introduce the compiler framework—SARA—that targets Plasticine architecture from high-level programs described in the Spatial language.

Front-end Selection Although SARA takes Spatial as front-end, the compilation techniques in SARA can be equally applied to other imperative langauges with nested loop constructs, such as C-based high-level syntehsis language, the backend of Halide IR, and other DSLs at the similar abstraction level. Using Spatial as our front-end language has the advantage that the language is designed for reconfigurable hardware, such that it can express valid execution schedule that can be exploit by spatial architectures natively in the language.

The data structure reprsented most exisitng imperative languages IR often marries to the CPU's virtual memory model. LLVM-based compilers, for example, treats data collections as pointers to a shared global address space, without distinguising what data are stored on-chip vs. off-chip. This is CPU provides the memory abstraction that any data within the global address space can be equally accessed and the hardware implicitly manage the data movement between on and off-chip access by brining Accelerators on the other hand, has explicitly managed on-chip scratchpads, that is not a cached version of main memory data implicitly managed by the hardware. The idea is to have the algorithm, which has better understanding of the data characteristics, to explicitly control what data gets moved on and off-chip to maximize locality. Other important static analysis in synthesis compilers, such as banking analysis, requires the compiler to have the global view all access patterns on a data structure. Therefore, modeling the data structures as disjoint memory space is much more suitable than pointers to a shared memory for reconfigurable architectures.

Without lost of generality, we use python-style pseudo code to represents the front-end programming abstraction the the rest of our discussion.

4.0.1 Native Low-level Programming Interface of Plasticine

Similar to FPGA high-level synthesis tools, SARA provides a high-level imperative programming abstration and synthesizes the program to execute on a reconfigurable accelerator. Targeting an RDA, however, is much more challenging than targeting FPAGs due to RDA's stringent mapping constraints. Unlike FPGAs, RDAs cannot map arbitrary RTL functionality.

Taking Plasticine as an example, the hardware has collection of memory tiles (PMUs) and compute tiles (PCUs). The mesh global network can be statically configured to connect any tiles with guaranteed in-order transmission of packets over arbitrary network latency.

Pattern Compute Unit (PCU)

As the major compute work horse of the architecture, a PCU contains a 6-stage SIMD pipeline with 16 SIMD lanes. Unlike a processor core, the PCU can only statically configure six vector instructions throughout the entire execution. Additionally, the six instructions must be branch-free in order to be fully pipelined across stages. At runtime, the SIMD pipeline executes the same set of instructions over different input data. Software can configure the SIMD pipeline to depend on a set of input streams and produce a set of output streams. There are three types of streams–single-bit control streams, 32-bit word scalar streams, and 16-word vector streams–corresponding to three types of global networks. Execution of the PCU is triggered by the arrival of its input dependencies and back pressured by the downstream buffers. The PCU also contains configurable counters, which can be chained to produce the values of nested loop iterators used in the datapath.

We refer to the program graph that can be executed by the SIMD pipeline as a **compute context** or simply **context**. A context includes the branch-free instructions mapped across SIMD stages, the input and output streams, and associated counter states and control configurations. TODO: shows example of simplified pseudo assembly code to program a PCU context. The control signals of the configurable counters, such as counter saturation or **counter done** signals, can be used to dequeue and enqueue the input and output streams, respectively. The counter bounds (i.e. min, max, and stride) can also be data-dependent using values from the scalar input streams. Compared to other data-flow architectures, the data-flow engine in Plasticine is more flexible in that it allows dynamic enqueue and dequeue window for its input and output streams. *This feature enable Plasticine to support complex control hierarchy, such as nested loops and branch statements, across contexts even though individual contexts can only executes instructions that are control-free.*

There is no global scheduler to orchestrate the execution order among contexts—the execution is purely streaming and data-flow driven. The only way to order the execution of two contexts without a data-dependency is to introduce a control **token** between two contexts acting like a dummy data-dependency. This restriction eliminates the need of long-travelling wires and communication hot spot caused by a centralized scheduler, which again improves the clock frequency and scalability of the architecture.

Pattern Memory Unit (PMU)

PMUs hold all distributed scratchpads available on-chip. Each PMU contains 16 SRAM banks with 32-word access granularity. The PMU also contains pipeline stages specialized for address computation. Unlike SIMD pipelines in PCUs, these pipeline stages are non-vectorized and can only perform integer arithmetics. The address produced by the address pipeline is broadcasted to 16 banks with a configurable offset added to each bank. In contrast to PCU SIMD pipeline that has to be programmed atomically, the address pipeline stages within PMUs can be sliced into a write and a read context, triggered independently. In addition to compute stages, resources such as I/O ports, buffers, and counters are also shared across contexts. It is the software's responsibility to make sure total resources consumed by all contexts does not exceed resource limits of the PMU. All contexts within PMU have access to the scratchpads with unprotected order. For instance, to restrict a read context to access the scratchpad after the write context, the software must explicitly allocate a token from the write context to the read context. SARA automatically generates required control tokens across context such that the memory access order is consistent to the one from high-level program order.

Unlike most accelerators at this scale, Plasticine does not have any shared global on-chip memory. This design dramatically improve the memory density and scalability of the architecture by eliminating hardware complexity and overhead to support complex cache coherence protocol. On the other hand, however, the burden of maintaining consistent view of a logical memory mapped across distributed scratchpads is left to the software. The software must explicitly configure synchronizations across PCUs and PMUs, taking into account that the network can introduce variable latency on both control and data path. One of the major contribution of SARA is to hide these synchronization burden from the programmer and still provide a programming abstraction of logical memories with variable bandwidth and arbitrary capacity that fits on-chip.

DRAM Interface

The Plasticine architecture provides access to four channels of the DDR4 off-chip memory on two sides of the unit array. Each side has a column of DRAM address generator (DAG) specialied to generate off-chip requests. Like compute pipeline in PMUs, the pipeline in DAG is also non-vectorized with integer arithmetics. Each DAG can generates a load or a store request streams to the off-chip memory. All streams can access the entire address space of the DRAM, with in-order response within each stream and no ordering across streams.

Abstract Programming Model

4.1 Compiler Overview

In this section, we describe a systematic approach to compile applications with a centralized control hierarchy into a distributed dataflow graph that can run on an RDA. **??** shows our compilation flow. First, SARA takes the input graph and performs a *virtual block allocation*. A virtual block (VB) is our intermediate representation that eventually gets assigned to a PB. Communication across VB tolerates an arbitrary amount of network latency. The *VB allocation* phase allocates compute, memory, and necessary synchronization across VBs to produce the correct result. The output of the *VB allocation* is a VB dataflow graph (VBDFG) that can be executed on an RDA with infinite-sized PBs.

The next phase is the *PB allocation* phase that assigns each VB to execute on a PB. This step addresses VBs that do not satisfy PB constraints by decomposing them into multiple VBs. After all VB fits in at least one PB, we perform a global optimization that merges small VBs into a larger VB to reduce fragmentation in mapping. The output of the *PB allocation* phase assigns a PB type to each VB, which is input to a *placement and routing (PaR)* phase.

The PaR phase maps the VBDFG graph on the accelerator array. PaR on RDA is similar to PaR on an FPGA as studied in many previous works [**?**], which we do not discuss further in this paper. After PaR, SARA generates configuration for PBs which executes on the RDA.

In the following sections, Section 4.2 describes the major challenge addressed during *VB allocation*, which is converting the centralized control hierarchy into distributed control-flow. Section 4.3 details program-partitioning passes that decompose program over distributed resources. In Section 4.4, we talk about optional optimizations that either improves the runtime or reduces resource usages for an application.

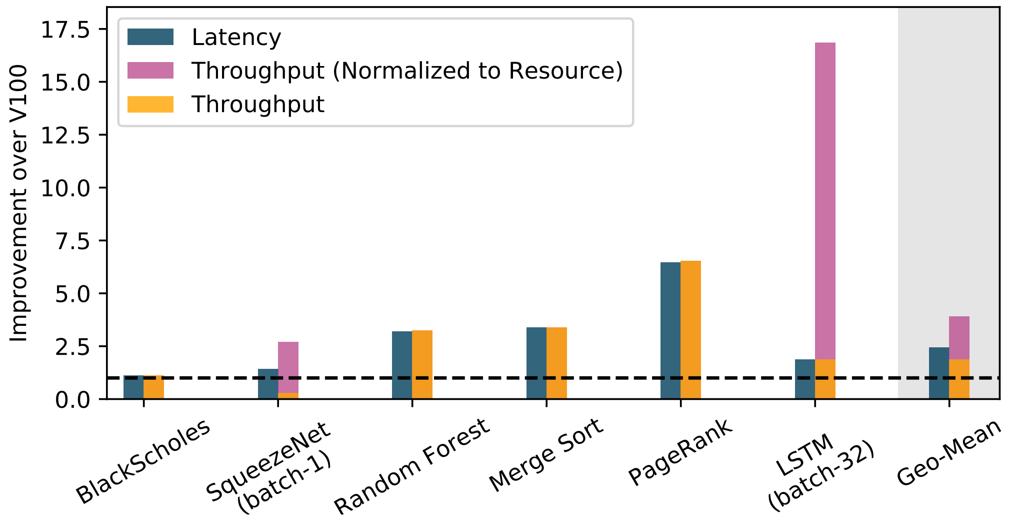


Figure 4.1: Plasticine's latency and throughout improvement over V100 GPU. The evaluated Plasticine architecture has area footprint of $352mm^2$ at 28nm. V100 GPU has area footprint of $815mm^2$ at 12nm. Both platforms have the same off-chip bandwidth at 1TB/s with HBM technology. Yellow and blue bars show the raw measured speedup in throughput and latency, respectively. To account for the resource discrepancy, the pink bar shows the normalized throughput for compute-bound application–SqueezeNet and LSTM, which scales performance with additionally on-chip resources.

4.2 Distributed Control Flow

In this section, we detail how SARA converts the control-input graph (**??**) into a distributed VBDFG.

In a naïve approach, we can map each controller in the hierarchy into a VB (**??**). This strategy suffers from expensive network round-trip delays between the parent and child controllers. If the parent controller is an unrolled loop, the parent needs to synchronize with all child controllers, which creates an undesired communication hot spot. **??**(a) shows an example where synchronization *just* between parent and child controllers can produce an incorrect result due to unpredictable network latency.

The alternative approach explores a different way to execute the expected control schedule correctly. The minimum required synchronization to produce the correct result is to ensure that the computations access the intermediate results in a consistent matter as if the control schedule is strictly enforced. This can be achieved via p2p synchronizations *only* between computations that access a particular shared memory. The execution order of computations that access different memories does not need to be enforced, as they do not impact the program outcome. Therefore, as long as the compututation is executed with the expected number of iterations and the memories are updated consistently, there is no need for any extra synchronization. Next, we walk through how SARA achieves this in more concrete detail.

Virtual Compute Allocation. As a start, SARA allocates workers to execute the dataflow graph within the *innermost* controllers of the input graph. We refer to our workers as *actors*. An actor can consume a set of input streams produced by other actors, perform operations, and send the output streams to other actors. The operations are from a basic block. Hence, they are *branch-free* and can be pipelined. The actor waits for all of its input dependencies before execution and stalls on downstream backpressure. A VB can contain one or multiple actors, as long as the total resource consumption of all actors satisfy the hardware constraint at the end. SARA assumes single cycle latency between actors within a single

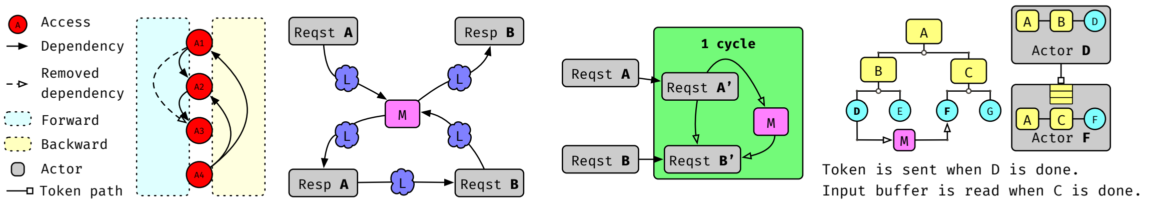
VB and variable latency for actors across VBs. SARA duplicates and distributes the outer controllers for each leaf controller to their corresponding actors, **??**(b). At this point, all actors have the local state to repeat their computation for expected iterations, completely asynchronously.

Virtual Memory Allocation. Next, SARA examines the intermediate results in the control hierarchy and allocates corresponding virtual memories. For each virtual memory, SARA generates synchronization between actors whose inner controllers read or write the memory within the input graph. SARA allocates a single-bit control token as an access grant to the shared memory and passes it between actors. The control token is no different from a regular boolean data-dependency. By controlling *where*, *how*, and *when* to pass the token, SARA is able to maintain a consistent update ordering between the pipelined and parallelized actors that access the shared memory.

4.2.1 Synchronization

We refer to an access to the memory in the input graph as a *declared access*, as supposed to accesses executed at runtime. For example, multiple accesses across loop iterations are counted as a single declared access.

Where. SARA only allocate resource to synchronize actors if their declared accesses can potentially interfere. Whether two declared accesses interfere depends on the type of accesses, the type of the memory, and location of the accesses in the control hierarchy. For every declared access, SARA checks other accesses of the same memory appeared earlier in the program order for a possible forward dependency, and later in the program order for a possible loop-carried dependency (LCD). Two declared accesses A and B have no dependency if their least-common ancestor (LCA) controller executes only one of the children at anytime (from a branch), or all children in parallel (from an unrolled loop). The LCD exists between B to A, if B occurs later in the program order, and A and B are surrounded by a loop. To detect LCD, SARA checks if a loop exists among two accesses' LCA controller



(a) Access dependency. **(b)** Synchronization. **(c)** Single-cycle special case. **(d)** Token sending mechanism.

Figure 4.2: (a) Access dependency graph. (b) Synchronization of two accesses on the same memory. (c) Single-cycle special case. (d) Actors uses local states of controller hierarchy to determine when to send a token.

and LCA's ancestor controllers. For a dual-ported SRAM, all other accesses need to be synchronized to: share address ports for read-after-read (RAR) and write-after-write (WAW); enforce true data-dependency for read-after-write (RAW); and prevent the overriding of read data for write-after-read (WAR). If the memory is multi-buffered (by the user or high-level compiler), we do not need to synchronize for WAR [?] (the writer writes to a different buffer than the reader does). The DRAM interface permits concurrent read streams and, hence, RAR does not need to be synchronized.

How. For each intermediate memory, SARA builds a dependency graph for all its declared accesses (Figure 4.2(a)). Enforcing all dependencies in this graph may not be necessary as dependencies between A_1 and A_2, and A_2 and A_3 already capture the dependency between A_1 and A_3. Therefore, SARA performs a transitive reduction (TR) on the graph to keep the minimum number of dependency edges that preserve the same ordering [?]. Since TR on a cyclic graph is NP-hard, we perform TRs on the forward and backward LCD graphs, separately. Notice, dependencies between accesses touching different buffers of a multi-buffered memory is less rigid than accesses touching the same buffer. Therefore, we can only remove an edge if all dependencies on the equivalent path have a stronger or equivalent dependency strength than the strength of the removed edge.

To eliminate the round-trip overhead between the memory and the computation, SARA duplicates the local states and expressions required to generate the requests in a separate actor as the one that handles the responses. For write accesses, the memory provides an acknowledgment for each request received, used by SARA for synchronization. The request actor generates requests asynchronously as soon as its data-dependencies are cleared, pushing all requests to memory until back-pressured. To order a declared access A before a declared access B SARA creates a dummy dependency between the actor that accumulates the response of access A ($resp_A$) and the actor that generates requests for access B ($reqst_B$) (Figure 4.2(b)). To enforce LCD from access B to access A, SARA introduces a token from $resp_B$ to $reqst_A$, and initializes the token buffer (input buffer receiving the token) with one element to enable the execution of the first iteration. If the LCD is on a multi-buffered memory, the LCD token is initialized with the buffer depth number of elements to enable A for multiple iterations before blocked by access B.

These are general schemes we use on any types of memory (including DRAM and on-chip memories) with unpredictable access latency.

Special Case: Single-Cycle Access For a memory with guaranteed *single-cycle* access latency, such as registers and statically banked SRAMs that are guaranteed conflict-free, we can simplify the necessary synchronization (Figure 4.2(c)). Instead of synchronizing between $resp_A$ and $reqst_B$, we allocate two stateless actors $reqst'_A$ and $reqst'_B$ within $t$he same VB as the accessed memory that forwards requests from $reqst_A$ and $reqst_B$, respectively. Next, we forward the token going from $reqst_A$ to $reqst_B$ to go through $reqst'_A$ to $reqst'_B$ instead, and configure the token buffer in $reqst'_B$ with the depth of one for serialized schedule and depth of M for multi-buffered schedule. We no longer need to insert the LCD token, as the stiff back pressure from the token buffer in $reqst'_B$ will enforce the expected behavior. This optimization only works if the sender and receiver of the token buffer are physically in a single VB where the memory is located. In this way, when $reqst'_B$ observes $reqst'_A$s token, $reqst'_B$ is guaranteed to observe the memory update from $reqst'_A$ because the memory also has single-cycle access latency.

Memory Localization We perform another specialization on non-indexable memories (registers or FIFOs), whose all accesses have no explicit read enables. Instead of treating them as shared memories, SARA duplicates and maps them to local input buffers in all receivers, no longer requiring tokens. The sender actor pushes to the network when the token is supposed to be sent, and the receiver dequeues one element from the input buffer when the token is supposed to be consumed. This dramatically reduces the synchronization complexity of non-indexable memory in the common case.

When. SARA configures the actors to generate the token using their local states at runtime. For FIFOs, the token is generated and consumed every cycle when the producer and receiver actors are active. For register, SRAM, and DRAM the program order expects that the producer and consumer writes and inspects the memory once per iteration of their LCA controller, respectively. Since the producer and receiver both have their outer controllers duplicated in their local state, they have independent views for one iteration of the LCA controller, which is when the controller in their ancestors (that is the immediate child of the LCA controller) is completed (Figure 4.2(d)). The *done* signals of these controllers are used to produce and consume the token in actors, independently.

4.2.2 Data-Dependent Control Flow

Using the synchronization discussed in Section 4.2.1, we can support control constructs that typically are not supported on most RDAs, such as branch and a while convergence loop. The controllers in the input graph can also have data dependencies, such as loop ranges. The dynamic-loop ranges are handled as data dependencies to actors with *memory localization* described in Section 4.2.1. The branch condition is also treated as a data-dependent enable signal of controllers under branch clauses. If the controller is disabled, it is considered *done* immediately. Output tokens depending on the *done* signal will be immediately sent out. For a memory written inside a branch statement and read outside the branch (with a branch miss), the writer actor immediately sends out the token to the receiver as soon as the branch condition is resolved. With a branch hit, the controller waits until its inner controller completes before raising the *done* signal. A similar scheme is used to implement the while loop, where the while condition is a data-dependency of stop signal of controller X.

The producer of the while condition also consumes its own output as an LCD. The condition is then broadcast to all other actors under the same while loop. The *done* signal of the while loop is raised when the condition's data-dependency evaluates to true. At this point, actors accessing memory within the while loop will send the token to access actors outside of the while loop, and enable them to access the intermediate memory.

After all actors and shared resources are allocated and synchronized, we simply put each actor and shared resource into their own VBs. The actors with single-cycle special case (Section 4.2.1) must be put in the same VB as the shared memory.

4.3 Program Decomposition

The output of the previous step (Section 4.2) is a VBDFG that correctly executes the original program on an RDA without resource constraint. The *PB allocation* phase enforces and addresses constraint violations given a hardware specification of the RDA. In the end, each VB is assigned to a PB type used by PaR to map VBs onto the actual PBs.

In contrast to a combined VB allocation and assignment approach, where each VB is allocated for a specific type of PB, our approach enables maximum resource utilization with specialization in PBs, because we model what resource each VB consumes and each PB processes without tracking what types they are. For example, an application without DRAM access can use the specialized DRAM address generator PB to perform other computations.

To enforce resource constraints, SARA uses a VB-PB bipartite graph to keep track of potential valid assignments between the two. We refer to PBs with connection to a VB as the domain of the VB $dom(VB)$. Starting from a complete bipartite graph, a list of pruning steps, each considers a different type of on-chip resource and incrementally removes the edges that violate their resource constraints. The hard constraints restrict VBs to PBs to assigned resources, such as scratchpads and memory controller interfaces. The soft constraints, such as the number of operations in a PB, are "fixable" by decomposing the VB. The soft constraint pruners contain fallback partitioning transformations that attempt to fix VBs with empty domains. The compilation stops if a VB has an empty domain that cannot be fixed. The partitioning and pruning can be an iterative process. For example, the memory partitioning can introduce new computation that requires compute partitioning. After all VBs have at least one PB in their domain, SARA triggers a merging step that compacts the small VBs with slack room in their PBs into larger VB to improve PB utilization. Next, we perform a quick heuristic check on the bipartite graph to see if there exists a possible assignment with enough PBs, and provide feedback on the critical resources, otherwise. Finally, SARA assigns each VB to a PB type with a backtracking search on the pruned bipartite graph and feeds the VBDFG with assigned PB types to the PaR phase. Section 4.3.1 and Section 4.3.2 detail the two major partitioning on memory and compute.

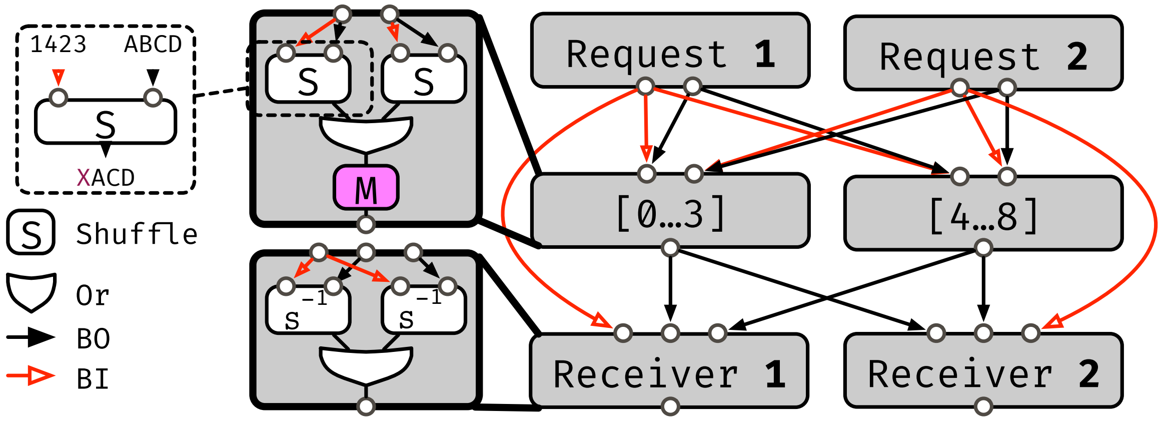


Figure 4.3: An example of splitting a memory to serve parallel requesters.

We have another partitioner encoding valid rule to decompose a BlackBox IP block available on the RDA.

4.3.1 Virtual Memory Decomposition

The memory pruner addresses VBs with virtual on-chip scratchpad memories exceeding the physical limits in capacity or number of banks in a PB. Memories in the input graph can have arbitrary size and number of virtual banks. The PBs, on the other hand, contains a small number of fix-sized 1-D scratchpad banks.

To partition the virtual memory, SARA shards the large virtual memory into multiple memory partitioned VBs, and assign each partition with a subset of the virtual banks. Each accessor provides a bank ID (BI) that selects which banks to access and a bank offset (BO) that specifies the address within the bank. BI and BO can be vectorized. SARA can use multiple banks within a PB to form a larger virtual bank. However, if a virtual bank in VB exceeds the total capacity of all physical banks within a PB, SARA further partitions the large virtual banks into multiple sub-banks, such that each sub-bank can fit into the aggregated capacity of a PB. To do so, SARA injects additional calculation to derive the sub-bank ID and sub-bank offset from the BO, and flattens the sub-bank ID with the previous BI to form the new BI and the new BO.

SARA then set ups the crossbar data path between the parallel producers, memory

Name	**Type**	**Description**	**Definitio**
$\mathcal{N}$	Constant	Enumeration of nodes to partition, numbered $\{n_i\}_i$	
N	Constant, $\mathbb{Z}_{\geq 0}$	Number of operations to partition	N
P	Constant, $\mathbb{Z}_{\geq 0}$	Number of partitions to consider	N, or fro
$\mathcal{E}$	Constant, $\{n_i \to n_j\}$	Directed edges representing dependence	
B	Variable, $\{0,1\}^{N \times P}$	Boolean Partitioning Matrix	
$\text{proj}_{\mathbf{B}}(\cdot)$	$\mathbb{Z}_{\geq 0} \to \mathbb{B}$	Function to convert a positive integer into a boolean	Supplemen
$\text{and}(\cdot,\cdot)$	$\{0,1\} \times \{0,1\} \to \{0,1\}$	Boolean and of binary variables	Supplemen
d_p	Variable, $\mathbb{Z}^P_{\geq 0}$	Vector of partition delays	
d_n	Variable, $\mathbb{Z}^N_{\geq 0}$	Vector of node delays	
$\text{dest}(n)$	$\mathcal{N} \to \mathcal{P}(\mathcal{N})$	The set of nodes which depend on n	$\{n' \| n' \in \mathcal{N}\ s.$
c_o	Constant, $\mathbb{Z}_{\geq 0}$	Maximum output arity of a partition	HW
c_i	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input arity of a partition	HW
b_d	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input buffer depth	HW
K	Constant, $\mathbb{R}_+$	Very Large Constant, used for constraint activation	P
α_d	Hyperparameter, $\mathbb{R}_+$	Retime merging probability multiplier	max

Table 4.1: Names and definitions used in the solver-based partitioning.

Type	**Description**	**Expression**
Objective	Allocated Partitions	$\Sigma_i \text{proj}_{\mathbf{B}}(\Sigma_j B_{i,j})$
	Additional Retiming Partitions	$\alpha_d \Sigma_{n_i \to n_j \in \mathcal{E}} \text{proj}_{\mathbf{B}}(\max\{d_n(j) - d_n(i$
Constraint	Partition Assignment	$\forall n_i \in \mathcal{N} :\ \Sigma_j B_{i,j} = 1$
	Dependency Constraint	$\forall n_i \to n_j \in \mathcal{E} :\ d_n(i) + 1[p_i \neq p_j]$
	Output Arity Constraint	$\forall p \in [0, P) :\ \Sigma_{n_s \in \mathcal{N}} \text{and}(B_{s,p}, \text{proj}_{\mathbf{B}}(\max\{(\Sigma_{n_d \in \text{dest}(n_s}$
	Input Arity Constraint (vectorized)	$\Sigma_{n_i \in \mathcal{N}} \max\{\text{proj}_{\mathbf{B}}(\Sigma_{n_j \in \text{dest}(n_i)} B_{j,:}) - B$
	Delay Consistency	$\forall n_i \in \mathcal{N} :\ d_n(i) \leq \min_j (d_p(j) + K -$
		$\forall n_i \in \mathcal{N} :\ d_n(i) \geq \max_j (d_p(j) + B_{i,j}$
	Constant Validity	$\forall n_i \in \mathcal{N} :\ d_n(i) \leq K$
		$\forall i \in [0, P) :\ d_p(i) \leq K$

Table 4.2: Solver formulation for partitioning*.

Name	**Type**	**Description**	**Defi**
$\mathcal{C}_r$	$[\mathcal{N} \to \mathbb{R}_+, \mathbb{R}_+, [\mathbb{R}_+] \to \mathbb{R}_+]$	List of per-node values, limits, and reduction functions for reducible constraints	Suppl
F	$\{0,1\}^{N \times P}$	Feasibility matrix, whether a partition can support a node	

Table 4.3: Table 4.1 extension for solver-based merging, which is a generalization of the partitioning problem.

Type	**Description**	**Expression**
Constraint	Feasibility Constraint	$\forall i, j \in [0, N) \times [0, P) :\ B_{i,j} \leq F_{i,j}$
	Reducible Constraints	$\forall j \in [0, P).\ \forall (c(\cdot), c_v, r(\cdot)) \in \mathcal{C} :\ r([c(n_i) \times B_{i,j}]_{n_i \in \mathcal{N}}) \leq c_v$

Table 4.4: Table 4.2 extension for solver-based merging*. The Retiming Partition objective is not used for merging.

*Expressions are presented using the Disciplined Convex Programming ruleset [?, ?]. Explanations for selected expressions can be found in the supplemental material.

partitions, and parallel consumers. As discussed in Section 4.2.1, each accessor is split into a requester actor and receiver actor. For each memory partition, SARA uses an actor to merge the requests from all parallel requesters, Figure 4.3. The merge actor uses a special shuffle operator that shuffles the BO vector from the order in BI to the order aligned with banks in its partition. If a bank in the partition is not accessed in BI, the output of the shuffle is marked as invalid. We assume the capacity of each physical bank is much smaller than 2^{31}. So we use the first bit of the 32-bit BO to indicate invalid accesses (0 is invalid), which is also used to explicitly disabled access from the program. Next, the merger actor uses a tree of bit-wise OR operators to combine all bank-aligned BOs, and send the combined request vector to its partition. The requests can be trivially ORed because static banking (**??**) guarantees that no two requesters access the same bank in the same cycle. Next, the memory partitions broadcast the respond to all receiver actors. A receiver takes response from all partitions, using the same shuffle operators to align each response back to the requested order in BI, and uses another OR tree to merge the response. The BI is forwarded from the requester to the receiver for the reverse shuffling.

The alternative approach is to reverse the respond to access ordering within the memory partitions before sending them to the requester. This approach does not scale with network bandwidth, as the memory partitions need to send the receiver number of distinct outputs. (The number of receivers is a function of the parallelization factor.) As a result, the amount of output bandwidth at the memory partition limits how much the program can be parallelized, which causes underutilization of the accelerators. In our scheme, each partition sends a single broadcast to all receivers, which is efficiently handled by the network.

The request trees for memory partition and receiver can have high fan-in, which can be partitioned into a tree of VB during the compute partitioning phase in Section 4.3.2.

4.3.2 Virtual Compute Decomposition

Partitioning. The *compute-partitioning* phase addresses VBs using more compute than any PB can provide. If a VB contains multiple actors, SARA first attempts to move the actors into separate VBs. If a single actor exceeds the resource limit, SARA breaks down the large dataflow graph in the actor into multiple actors and put them in separate VBs. During partitioning, SARA maps each subgraph of the large dataflow graph into a new actor, mirrors the control states of the original actor, and streams live variables in between. The PB constraint includes the number of operations, input, and output ports available in the PBs. Because the global network is specialized to handle efficient broadcasts, the in- and out-degree constraint counts the number of broadcast edges, as supposed to edges between partitions (Figure 4.4(a)). In addition, the partitioned subgraphs cannot form *new* cycle between partitions that did not exists in the original VGDFG. Figure 4.4 (b) shows an illegal partitioning that fails the cycle constraint. This is because each actor is enabled atomically by all of its data-dependencies; cyclic dependencies between actors cause deadlock. The original dataflow graph, however, can contain cycles that correspond to LCDs. The back edge of the LCD is initialized with dummy data to enable execution in the first iteration.

Retiming. To pipeline all partitions at full-throughput, SARA must retime the imbalanced data paths between partitions with sufficient buffering. Retiming can introduce new VBs in addition to partitioned VBs. The objective of this phase is to minimize the number of partitions after retiming and minimizing the amount of connectivities between partitions.

The partitioner "fixes" the VB based on a single PB specification, albeit there are many potential PBs the decomposed VB can be mapped to. Currently, we use a heuristic to select a PB type from the $dom(VB)$ right before the compute pruning as a guiding constraint for partitioning. In the following sections, we present a traversal-based solution that provides a decent solution with fast compile time, and a convex optimization solver-based solution that provides an optimum solution with long turn-around time.

Traversal-based Solution To address the cycle constraint, we perform a topological sort of the dataflow graph. The topological traversal ignores the back-edge in the graph and produces a list of traversed nodes. The compiler starts from the beginning of the list, recursively adds nodes into a partition until the partition no longer satisfies the hardware

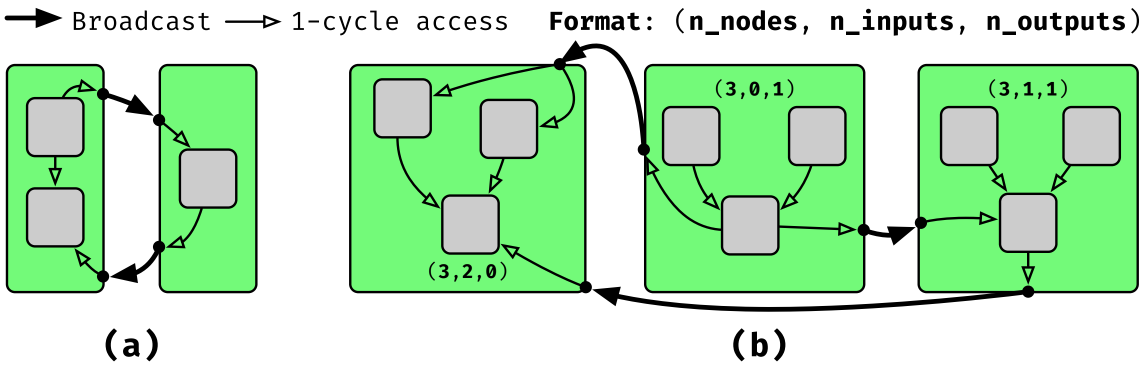


Figure 4.4: (a) An example of an illegal partition. (b) The cost of a partition.

constraint, and repeats the process with a new partition. This approach guarantees that no cycle is introduced with $O(V + E)$ complexity, where V and E are the numbers of vertices and edges. However, the outcome of the partitioning is a function of the traversal order, which does not guarantee an optimum solution, which we experienced with depth-first search (DFS) and breadth-first search (BFS) with forwarding and backward traversals. For DFS, we re-sort the remaining list each time we start with a new partition.

Solver-based Solution Table 4.2 gives our formulation of the partitioning problem. At a high-level, we use a boolean matrix B to keep track of the assignment of nodes in the dataflow graph to partitions. B has dimension of number of nodes to number of partitions, where$B[i, j] == 1$ indicates an assignment of node i to partition j. Each node is constrained to have a single partition assignment. The input and output arity constraints show the formulation of the PB I/O constraints. These are the two most challenging constraints as we need to identify broadcast edges across partitions. To address the cycle constraint, we introduce a delay vector d with size equal to the number of nodes. The delay vector encodes a schedule to execute each node. A node cannot execute earlier than its input dependencies and cannot be scheduled later than its output dependencies (Delay Consistency). The dependency constraint further limits nodes belonging to the same partition to have the same delay. This delay variable is also used to calculate where retiming is required and projection of the amount of retiming VBs introduced. The final object is just the sum of partitioned VBs and retiming VBs. To limit B to a small size, we use the traversal solution to determines the initial column size of B.

4.4 Optimizations

In this section, we discuss a few compiler optimizations that improve the runtime or reduces resource usage in RDAs.

Memory strength reduction (msr). Like traditional strength reduction on arithmetics, SARA replaces expensive on-chip memories with cheaper memories whenever possible. For example, SARA replaces a scratchpad with constant address in all accesses to a un-indexable memory, such as a FIFO. This commonly happens when producer and consumer loops of the memory are fully unrolled.

Route-Through Elimination (rtelm). For patterns where the content of a non-indexable memory (M1) is read and written to another memory (M2), SARA eliminates the intermediate access if the read of M1 and the write of M2 operates in lock-step.

Retiming with scratchpad (retime-mem). By default, SARA uses PB input buffers for retiming purpose. This option enables SARA to use scratchpad memory for retiming that requires large buffer depth.

Crossbar datapath elimination (xbar-elm). Although, crossbars between accessors and memory partitions (Section 4.3.1) are very expensive in the general case, the BI sometimes can be statically resolvable with certain combinations of parallelization on memory accesses. When BI is a constant, SARA can use this information to intelligently assign virtual banks to partitions that reduce the crossbar data path to a partial or a point-to-point connection.

Read request duplication (dupra). During memory partitioning (Section 4.3.1), instead of forwarding BI from the requester to receiver, SARA can also duplicate the BI with local state on receiver side, which eliminates the unbalanced data path at the cost of extra computation.

Global Merging (merge). After all VBs satisfy the hardware constraint, we perform a global optimization to compact small VBs into larger VBs. Merging has very similar problem statement as compute partitioning (Section 4.3.2) except with more constraints. The traversal-based algorithm requires a reference cost for PB to check if the merged VB still satisfy the hardware constraint. At each step of merging, we take the union of the domains of VBs within the current partition, and intersect with the domain of the merging VB. The caveat is that even with a non-empty intersection, the bipartite graph might not have a possible assignment, as merged VB might fit in a larger PB with insufficient quantity. Therefore, we perform a heuristic checking on feasibility of the bipartite assignment at each step of merging. The solver-based solution combines partition assignment with PB type assignment as a joint problem. The output of merging gives both partition assignment as well as a PB type assignment, which eliminates the risk of un-mappable bipartite graph due to merging in the traversal-based solution.

4.5 Register Allocation

4.6 Debugging and Instrumentation Support

Deadlock in Streaming Reconfigurable Architecture

Debugging Support and Performance Instrumentation

Chapter 5

Conclusions

18 For I testify unto every man that heareth the words of the prophecy of this book, If any man shall add unto these things, God shall add unto him the plagues that are written in this book:

19 And if any man shall take away from the words of the book of this prophecy, God shall take away his part out of the book of life, and out of the holy city, and from the things which are written in this book.

Chapter 6

Future Work

Appendix A

Appendix

18 For I testify unto every man that heareth the words of the prophecy of this book, If any man shall add unto these things, God shall add unto him the plagues that are written in this book:

19 And if any man shall take away from the words of the book of this prophecy, God shall take away his part out of the book of life, and out of the holy city, and from the things which are written in this book.

Bibliography

[1] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 296–311, ACM, 2018.

[2] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 115–127, June 2016.

[3] 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*, pp. 389–402, ACM, 2017.

[4] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 1–14, IEEE Press, 2018.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12, IEEE, 2017.