Scalable Interconnects for Reconfigurable Spatial Architectures

Yaqi Zhang
yaqiz@stanford.edu
Stanford University
Stanford, CA

Alexander Rucker
acrucker@stanford.edu
Stanford University
Stanford, CA

Matthew Vilim
mvilim@stanford.edu
Stanford University
Stanford, CA

Raghu Prabhakar
raghup17@stanford.edu
Stanford University
Stanford, CA

William Hwang
williamhwang@stanford.edu
Stanford University
Stanford, CA

Kunle Olukotun
kunle@stanford.edu
Stanford University
Stanford, CA

ABSTRACT

Recent years have seen the increased adoption of Coarse-Grained Reconfigurable Architectures (CGRAs) as flexible, energy-efficient compute accelerators. Obtaining performance using spatial architectures while supporting diverse applications requires a flexible, high-bandwidth interconnect. Because modern CGRAs support vector units with wide datapaths, designing an interconnect that balances dynamism, communication granularity, and programmability is a challenging task.

In this work, we explore the space of spatial architecture interconnect dynamism, granularity, and programmability. We start by characterizing several benchmarks' communication patterns and showing links' imbalanced bandwidth requirements, fanout, and data width. We then describe a compiler stack that maps applications to both static and dynamic networks and performs virtual channel allocation to guarantee deadlock freedom. Finally, using a cycle-accurate simulator and 28 nm ASIC synthesis, we perform a detailed performance, area, and power evaluation across the identified design space for a variety of benchmarks. We show that the best network design depends on both applications and the underlying accelerator architecture. Network performance correlates strongly with bandwidth for streaming accelerators, and scaling raw bandwidth is more area- and energy-efficient with a static network. We show that the application mapping can be optimized to move less data by using a dynamic network as a fallback from a high-bandwidth static network. This static-dynamic hybrid network provides a 1.8x energy-efficiency and 2.8x performance advantage over the purely static and purely dynamic networks, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Interconnection architectures**; • **Hardware** → *Hardware accelerators.*

ISCA '19, June 22–26, 2019, Phoenix, AZ

KEYWORDS

interconnection network, reconfigurable architectures, hardware accelerators, CGRAs

ACM Reference Format:
Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable Interconnects for Reconfigurable Spatial Architectures. In *ISCA '19: The 46th International Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ*. ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

Spatially reconfigurable architectures are programmable, energy efficient application accelerators offering the flexibility of software and the efficiency of hardware. Architectures such as Field Programmable Gate Arrays (FPGAs) achieve energy efficiency by providing statically reconfigurable compute elements and on-chip memories in a bit-level programmable interconnect; this interconnect can be configured to implement arbitrary datapaths. FPGAs are used to deploy services commercially [20, 41, 48] and can be rented on the AWS F1 cloud [2]. However, FPGAs suffer from overhead incurred by fine-grained reconfigurability; their long compile times and relatively low compute density have hampered widespread adoption for several years [5, 7, 33, 45]. Therefore, recent spatial architectures use increasingly coarse-grained building blocks, such as ALUs, register files, and memory controllers, distributed in a programmable, word-level static interconnect. Several of these Coarse-Grained Reconfigurable Arrays (CGRAs) have recently been proposed [14, 17, 18, 27, 32, 34, 35, 43, 46].

Applications are mapped to CGRAs by distributing computations spatially across multiple processing blocks and executing them in a pipelined, data-driven fashion. On traditional Networks on Chip (NoCs), communication is the result of explicit message passing by parallel workers or cache misses; these are bursty and relatively infrequent. On CGRAs, however, applications are distributed by parallelizing and pipelining; pipelining introduces frequent and throughput-sensitive communication. Because different applications are parallelized and pipelined differently, they have different communication requirements.

CGRAs need the right amount of interconnect flexibility to achieve high resource utilization; an inflexible interconnect constrains the space of valid application mappings and hinders resource utilization. Furthermore, in the quest to increase compute density,

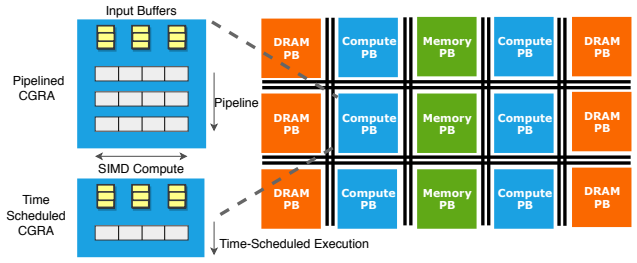


Figure 1: Abstract machine model of our target CGRA, with compute, memory, and DRAM Physical Blocks (PBs).

CGRA data paths now contain increasingly coarse-grained processing blocks such as pipelined, vectorized functional units [17, 39, 46]. These data paths typically have a vector width of 8–16x [46], which necessitates coarser communication and higher on-chip interconnect bandwidth to avoid creating performance bottlenecks. Although many hardware accelerators with large, vectorized data paths have fixed local networks [13], there is a need for more flexible global networks to adapt to future applications. Consequently, interconnect design for these CGRAs involves achieving a balance between the often conflicting requirements of high bandwidth and high flexibility.

Interconnects can be classified into two broad categories: *static* and *dynamic*. Static interconnects use switches programmed at compile time to reserve high-bandwidth links between communicating units for the lifetime of the application. CGRAs traditionally employ static interconnects [9, 55]. In contrast, dynamic interconnects, or NoCs, contain routers that allow links to be shared between more than one pair of communicating units. NoC communication is typically packet-switched, and routers use allocators to fairly share links between multiple competing packets. Although static networks are fast, they require over-provisioning bandwidth and can be underutilized when a dedicated static link is reserved for a logical link that is not 100% active. While dynamic networks allow link sharing, the area and energy cost to transmit one bit of data is higher for routers than for switches, making bandwidth scaling more expensive in dynamic networks than in static networks.

In this paper, we start by detailing the key considerations involved in building a CGRA network, including those arising from network design, CGRA architecture, and the characteristics of spatially mapped applications. Network designs must be carefully considered because vectorization magnifies inefficiencies: the increased network area of a vectorized design ensures that any overhead has a significant impact. Next, we evaluate the performance, area, and power requirements of several interconnection designs using cycle-accurate simulation and ASIC synthesis of a switch and router with a 28 nm industrial technology library. We then explore a variety of design points, including static, dynamic, and hybrid networks, decreased flit widths and VC counts for dynamic networks, and different flow-control strategies for static networks.

We show that CGRA network designs must consider application characteristics and the execution model of the underlying architecture. Performance scales strongly with network bandwidth, with an 8x average performance gap between the best and worst configurations. The hybrid network gives the best network energy-efficiency: a 1.83x average improvement over the static network. On pipelined architectures, hybrid networks can also match the performance per area of higher bandwidth, purely static networks with less than 8% performance loss.

The key contributions of this paper are:

(1) An analysis of key communication patterns exhibited by spatial architectures.
(2) A network-aware compiler flow that efficiently targets static, dynamic, and hybrid networks with varying granularities.
(3) A quantitative analysis of the performance, area, and energy trade-offs involved in choosing a CGRA network, using benchmarks drawn from various application domains.

The rest of the paper is organized as follows: Section 2 provides background on the communication patterns that motivate interconnect design and describes the network design space explored in this paper. Section 3 describes our compiler flow for a vectorized CGRA, including how we physically map applications. Section 4 details our evaluation methodology and experimental results. Section 5 discusses related work, and Section 6 offers concluding remarks.

2 BACKGROUND

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 1. 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 1. 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 [46].

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 [13] and DaDianNao [8].

2.1 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:

2.1.1 Vectorized communication. Recent hardware accelerators use large-granularity compute tiles (e.g., vectorized compute units and SIMD pipelines) for SIMD parallelism [39, 46], 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.

2.1.2 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.

2.1.3 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 [13, 40, 46]. 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 multi-processors, 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 [49]. For spatial accelerators, however, compute performance is limited by network throughput, and latency is comparatively less important.

2.1.4 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.

2.2 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.

2.2.1 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 [58], 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.

2.2.2 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 [10]. 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 [21]. 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 [58]. 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

```
// 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
    }
  }
}
```

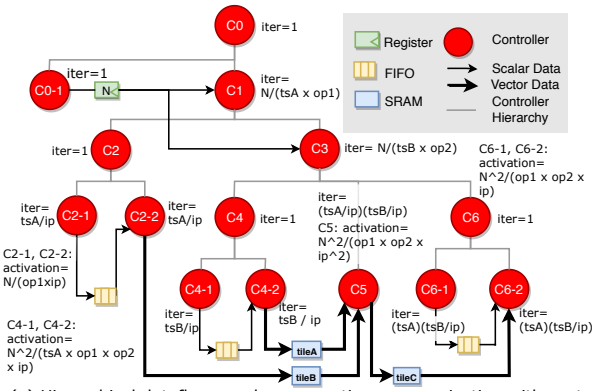
Figure 2: Example of outer product in Spatial pseudocode.

ensure that all logical paths traversing the same physical link are placed into separate buffers.

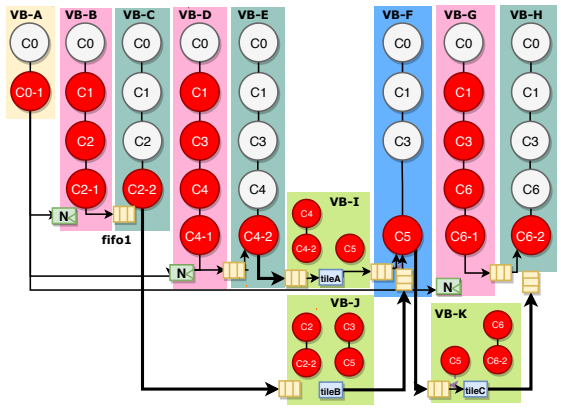
2.2.3 Hybrid networks. Finally, we explore hybrids between static and dynamic networks that run each network in parallel. During static place and route, the highest-bandwidth logical links from the program graph are mapped onto the static network; once the static network is full, further links are mapped 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.

2.3 High-Level Abstraction

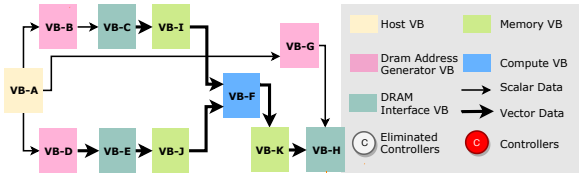
We use Spatial, an open source domain specific language for reconfigurable accelerators, to target spatial architectures [30]. Spatial describes applications with nested loops and an explicit memory hierarchy that captures data movement on-chip and off-chip. This exposes design parameters that are essential 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. 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 [31].



(a) Hierarchical dataflow graph representing communication with nested controllers. Using annotated input sizes, our compiler derives the activation counts of loop nests and logical links. (iter: iteration of a loop. activation: total number of iterations the controller will be active)



(b) VB Allocation. Controllers are duplicated and distributed to VBs. Controller signals are used to enqueue/dequeue inputs/outputs; the compiler eliminates duplicated controllers whose output signals are not used.



(c) VB dataflow graph. This program graph will be placed and routed on the global network.

Figure 3: Target specific compilation: the hierarchical control graph is transformed into a distributed dataflow graph for the outer product example shown in Figure 2 .

3 METHODOLOGY

3.1 Compilation

Figure 4 shows the overall compilation and mapping process. We start by taking the Intermediate Representation (IR) output by the Spatial compiler and lowering it into a target-specific IR, which is a distributed data flow graph consisting of *Virtual Blocks* (VBs). A single VB corresponds to an arbitrarily-sized computation or memory block, and edges between VBs capture data dependencies that are routed across the global network. Next, the VB graph is mapped onto a PB graph that describes the underlying architectural

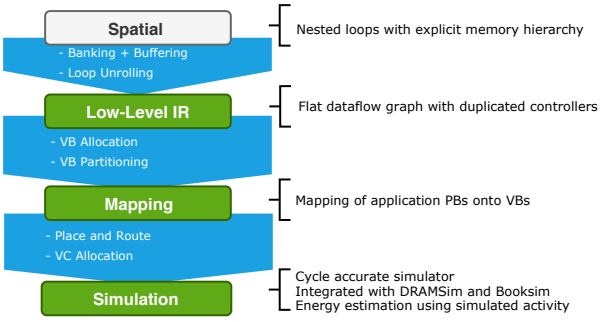


Figure 4: Compilation and mapping flow.

constraints, such as scratchpad sizes and pipeline depths (Figure 1). Finally, the mapped VB graph is placed and routed onto the physical array and associated network.

3.1.1 Virtual block allocation. The first compilation step is converting nested loop constructs into a distributed, streaming VB dataflow graph. Figure 3(a) shows the input to our compiler: the hierarchical controller-based representation of the Spatial program shown in Figure 2. For simplicity, the graph shown in the diagram is not unrolled. We allocate one VB per program basic block and extract SIMD parallelism by vectorizing the inner loop within a single VB. This vectorization corresponds to the inner loop parallelization expressed in the program. Parallelization at arbitrary loop levels (beyond SIMD parallelism) is achieved by loop unrolling, where a single basic block is duplicated and mapped to multiple VBs.

To convert from the hierarchical model to a streaming dataflow graph, we duplicate and distribute nested loop controllers for each VB, as shown in Figure 3(b). The distributed controllers are used to match communication between VBs without the need to send explicit control information. Because the same parent controller is duplicated at both ends of a link, the receiver knows exactly how much data to expect. Instead of requiring an explicit control signal, the receiver can simply count the number of input packets received. The output of this step is the VB dataflow graph shown in Figure 3(c), where edges indicate single-bit, scalar, or vector data dependencies between VBs; these links are then mapped to the network.

3.1.2 Resource pruning and VB partitioning. Software can contain arbitrarily large basic blocks that do not fit in fixed-size hardware. For a given hardware specification, our compiler partitions a VB whenever its compute resources, on-chip memory capacity, or network bandwidth exceed hardware constraints. For example, computation partitioning occurs for basic blocks longer than the number of stages in a pipelined architecture or the scheduling window of a time-scheduled architecture.

Partitioning can also result from scratchpad bandwidth and capacity constraints. During parallelization, the on-chip memory used by parallelized readers and writers to communicate must be banked to provide the necessary bandwidth. When a memory VB has more banks than a memory PB, we use multiple PBs to map that logical memory and support higher access bandwidth. A similar process is used for memory VBs that exceed hardware capacity.

Finally, our compiler partitions VBs to keep the in- and out-degrees of nodes manageable. When a VB in a pipelined CGRA consumes too many inputs or produces too many *distinct* outputs, we partition it to reduce its degree and meet the input/output bandwidth constraints of a purely static network. For dynamic and hybrid networks, partitioning is not strictly necessary, but it improves performance by decreasing congestion at the network ejection port associated with a PB. We do not partition broadcasts with high output degrees because they are handled natively within the network.

We use a bipartite graph between VBs and PBs to indicate valid matches between the two, where an edge between a VB and PB suggests a valid mapping. With this representation, we support reusing specialized PBs (e.g., address generators) for general purpose VBs, instead of forcing a one-to-one mapping between VBs and PB types. Initially, all VBs and PBs are connected, forming a complete bipartite graph; we then remove edges between VBs and PBs that violate a resource constraint. After pruning, we recursively partition any VB with no valid PBs into multiple VBs until all VBs match at least one PB.

3.1.3 Link analysis and heuristic generation. In Spatial, the user can annotate runtime-variable input values to assist compiler analysis. We use these programmer annotations to compute the expected number of iterations for each controller statically; this heuristic guidance helps the placer evenly distribute traffic. However, we do not require exact annotations for efficient placement—a rough estimate of data size is sufficient for the placer to determine the relative importance of links. For loop controllers, the number of iterations per parent iteration is $\lceil (max - min)/(step \cdot par) \rceil$; controllers representing glue logic run once per parent iteration. The activation count of a link will be the product of all of its ancestral controllers' iteration counts.

Using the computed activation counts, the placer prioritizes highly used links for allocation to the static network and leaves infrequently used links on the dynamic network. When no annotation is provided, or loop bounds cannot be constant-propagated, the compiler estimates loop iteration counts based on the nesting depth: innermost loops are the most frequently active. This heuristic provides a reasonable estimate of links' priorities for routing purposes.

3.2 Placement and Routing

For purely static networks, the goal is primarily to find a valid placement—because static routes have guaranteed resources reserved, no two routes can conflict, and any placement found is essentially optimal. Some routes may have longer latency, which has a second-order performance impact compared to throughput. However, determining a purely static allocation of routes cannot be done in polynomial time and may be infeasible; therefore, the static network must be over-provisioned to aid placement. Although virtually all placements are valid for the dynamic and hybrid networks, not all valid placements are equal—some have less congestion and therefore run several times faster.

Dynamic network placement is performed with an iterative algorithm using heuristics to rapidly evaluate placements: a penalty score is assigned as a linear function of several subscores. These include projected congestion on dynamic links, projected congestion

at network injection and ejection ports, the average route length, and the length of the longest route. We estimate congestion by normalizing the number of packets on each link to the program link with the highest total packet count. The most active program link sets a lower bound on the program runtime (the highest bandwidth physical link can still only send one packet per cycle), which translates to an upper bound on congestion for other links. We start with random placement of the VBs. Then, a genetic algorithm shuffles the VBs whose links contribute most to congestion, and keeps the new position if it improves the route assignment. By iteratively replacing and re-routing, the mapping process eventually converges to a good placement.

3.2.1 Congestion-aware routing. To achieve optimal performance, we use a routing algorithm that detects congestion and routes around it. Routing starts with the highest-priority routes, as determined by fanout and activation count, which are fixed after they are routed. This makes sure that the static network is used most efficiently. Our scheme searches a large space of routes for each link, using Dijkstra's algorithm [11] and a hop weighting function. Routes are not analyzed on the basis of a single source-destination pair, which would be inadequate for broadcasts: instead, a directed graph is built from the source and all destinations in the route, with edge weights corresponding to the minimal route between each pair of VBs in the broadcast. For example, if the broadcast is from VB-1 to VB-2 and VB-3, four total potential routes are analyzed for congestion: VB-1 to VB-2, VB-1 to VB-3, VB-2 to VB-3, and VB-3 to VB-2. The routes are weighted so that routes mapped on the static network are preferable to those mapped to the dynamic network; within these categories, routes are weighted based on length.

Then, a search algorithm based on Prim's algorithm for minimum spanning trees [47] is run to build a tree for the broadcast, starting with only the source being reached. At every step, the most-preferable route (from the graph built using Dijkstra's algorithm) that adds a new destination VB to the reached set is chosen and added to the broadcast, until all destination VBs are reached. This route can start from either the source of the broadcast tree or any destination currently in the reached set. The algorithm will find a fully static broadcast tree, if one exists, and will only add a non-static route to the broadcast (moving the entire broadcast to the dynamic network) when there are VBs in the tree that cannot be reached from the source VB by *any* static route.

3.2.2 VC allocation for deadlock avoidance. Deadlock is a system pathology in dynamic routing where multiple flits form a cyclic holds/waits dependency on each others' buffers and prevent forward progress. Most data-flow accelerators use a streaming model, where outputs of a producer are sent over the network to one or more consumers without an explicit request; the producer is backpressured when there is insufficient buffer space. While this paradigm improves accelerator throughout by avoiding the round-trip delay of a request-response protocol, it introduces an additional source of deadlock [21].

Figure 5(a) shows a sample VB data-flow graph, which is statically placed and routed on a 2×3 network in (b). Logical links B and C share a physical link in the network. If C fills the buffer shared with B, VB-3 will never receive any packets from B and will not make forward progress. With streaming computation, the program

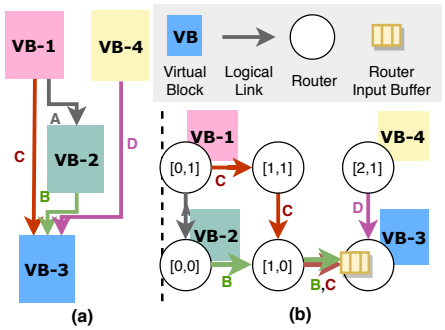


Figure 5: An example of deadlock in a streaming accelerator, showing the (a) VB data-flow graph and (b) physical placement and routes on a 2×3 network. There are input buffers at all router inputs, but only the buffer of interest is shown.

graph must be considered as part of the network dependency graph, which must be cycle-free to guarantee deadlock freedom. However, this is infeasible because cycles can exist in a valid VB dataflow graph when the original program has a loop-carried dependency. Therefore, deadlock avoidance using cycle-free routing, such as dimension-order routing, does not work in our scenario. Allocating VCs to prevent multiple logical links from sharing the same physical buffer is consequently the most practical option for deadlock avoidance on streaming accelerators.

3.3 Simulation

We use a cycle-accurate simulator to model the pipeline and scheduling delay for the two types of architectures, integrated with DRAMSim [57] 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 [25], 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.

3.3.1 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 [4] 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.

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 [15]	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 1: Benchmark summary

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

$$E_{\text{flit}} = \frac{(P - P_{\text{inactive}})\, T_{\text{testbench}}}{\#\text{flit}} \tag{1}$$

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:

$$E_{\text{net}} = \sum_{\text{allocated}} P_{\text{inactive}} T_{\text{sim}} + E_{\text{flit}} \#\text{flit}, \tag{2}$$

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

4 EVALUATION

To evaluate our compilation flow and network architectures, we use a set of benchmarks implemented in Spatial. We start with Spatial's output IR (Section 3.1), and transform it into a graph of distributed, streaming VBs. Our compiler then performs place and route for a target architecture before generating a configuration for cycle-accurate simulation. During simulation, we track the amount of data moved by each switch and router, which we integrate with synthesis results to produce estimates of area and power.

For each application, we find the highest-performing parallelization and tiling factors; for DRAM-bound applications, this is the configuration that saturates memory bandwidth. The optimum parameters for each network configuration may vary, as high parallelization does not improve performance on a low bandwidth network. We start with benchmark characterizations (Section 4.1), analyzing application characteristics and communication patterns to identify how they interact with networks. Next, we characterize the area and energy of network primitives, which we use to calculate the total network area and energy in Section 4.2. Finally, Section 2.2 presents a design space study over all network dimensions for both pipelined and scheduled architectures. Table 2 summarizes the notation we use to describe network configurations in the remainder of this section.

4.1 Application Characterization

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 1 lists the applications and their data size. Figure 6 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 [15]), 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 7(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 bandwidth and efficient broadcasts to sustain program throughput. On the contrary, time-scheduled architectures, shown in Figure 7(c,d), exhibit lower bandwidth requirements due to the lower throughput

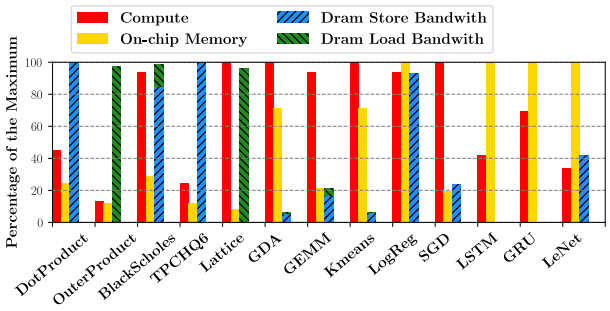


Figure 6: Physical resource and bandwidth utilization for various applications.

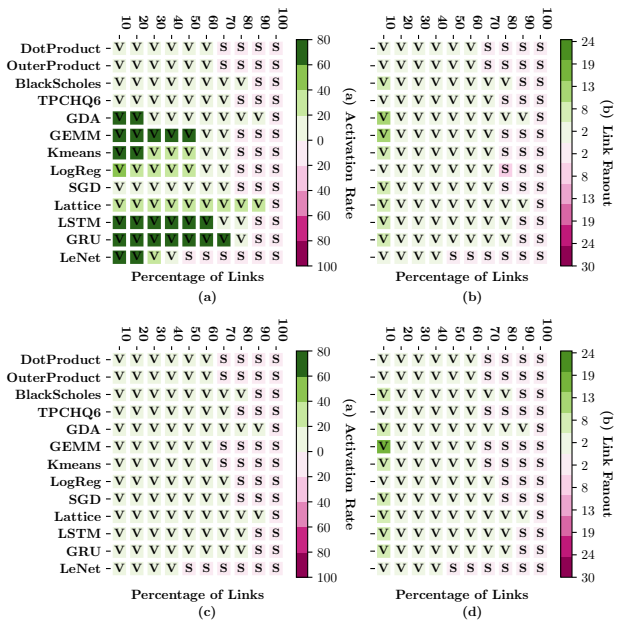


Figure 7: Application communication patterns on pipelined (a,b) and scheduled (c,d) CGRA architectures. (a) and (c) show the activation rate distribution of logical links at runtime. Links sorted by granularity, then rate; darker boxes indicate higher rates. The split between green and pink shows the ratio of logical vector to scalar links. (b) and (d) show the distribution of broadcast link fanouts.

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 8 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 3.1.2. The output degree does

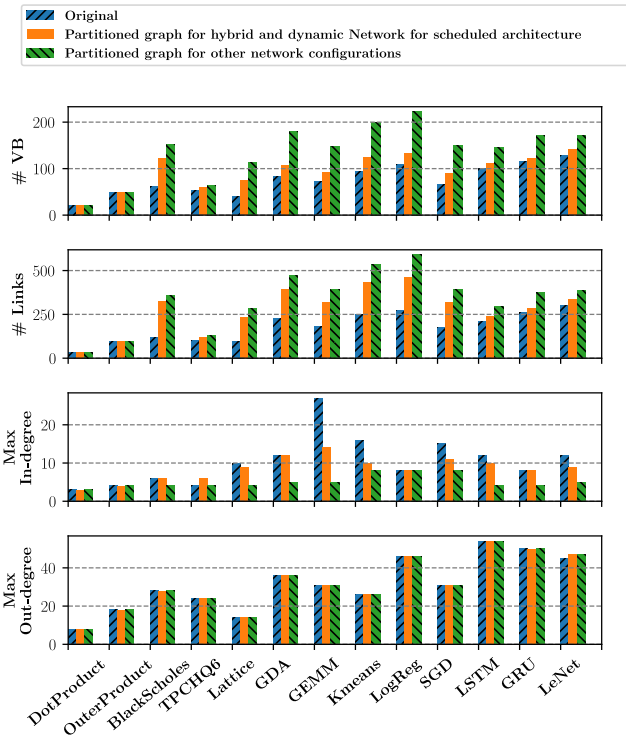


Figure 8: Characteristics of program graphs.

not change with partitioning because most outputs with a large degree are from broadcast links.

4.2 Area and Energy Characterization

Figure 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 [10], 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 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 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

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 2: Network design parameter summary.

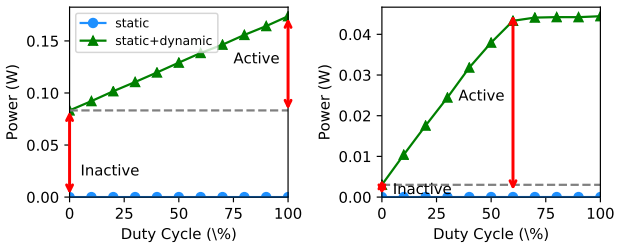


Figure 9: Switch and router power with varying duty cycle.

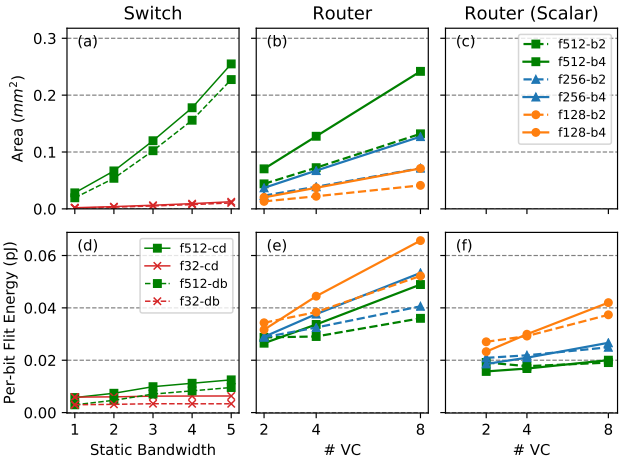


Figure 10: Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit).

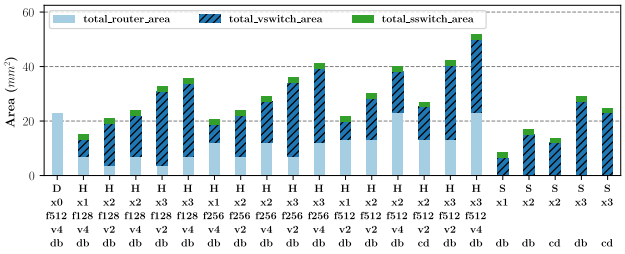


Figure 11: Area breakdown for all network configurations.

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 11 summarizes the area breakdown of all the network configurations that we evaluate.

4.3 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 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 4.3.1, we use a network size of 14×14 end-point PBs. All vector networks use a vectorization factor of 16 (512 bit messages).

4.3.1 Bandwidth scaling with network size. Figure 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 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.

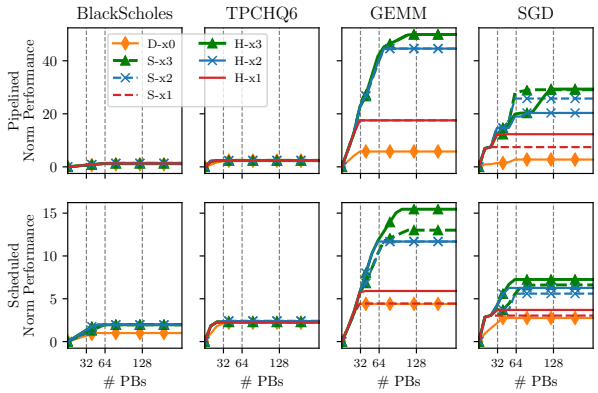


Figure 12: Performance scaling with increased CGRA grid size for different networks.

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.

4.3.2 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 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 4.3.1, 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 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.

4.3.3 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 4.2, 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 15 shows minimal performance improvement from using more VCs.

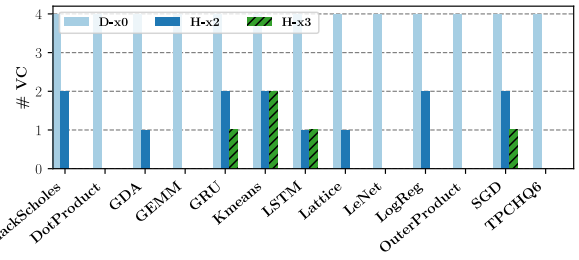


Figure 13: Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.)

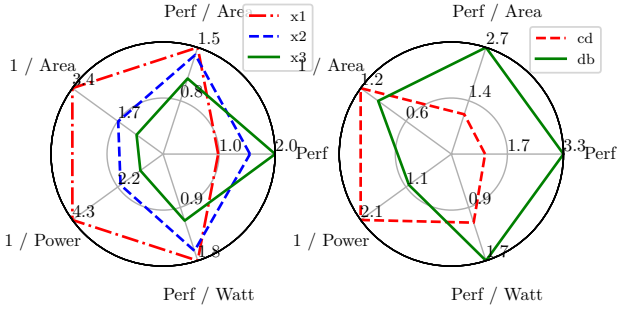


Figure 14: Impact of bandwidth and flow control strategies in switches.

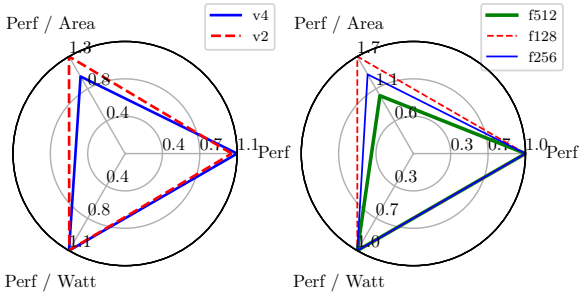


Figure 15: Impact of VC count and flit widths in routers.

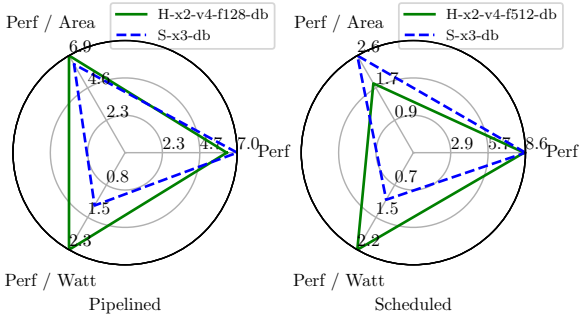


Figure 16: Geometric mean improvement for the best network configurations, relative to the worst configuration.

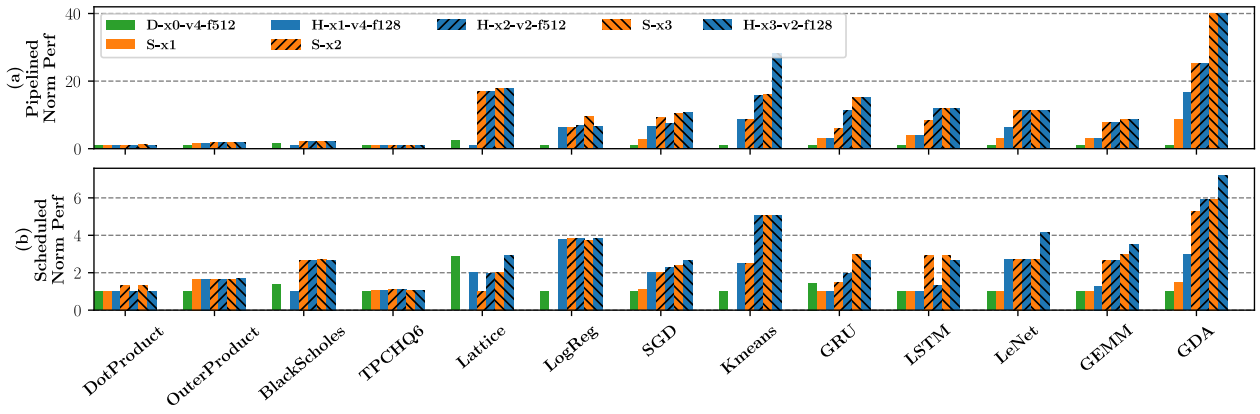


Figure 17: Normalized performance for different network configurations.

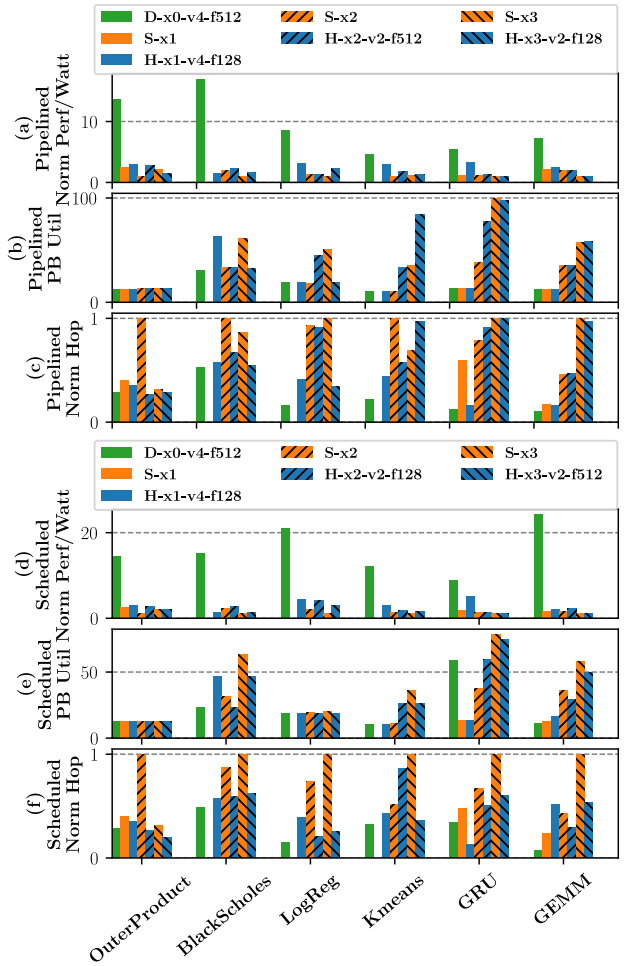


Figure 18: (a,d): Normalized performance/watt. (b,e): Percentage of compute and memory PBs utilized for each network configuration. (c,f): Total data movement (hop count).

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 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 4.2, 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 15 shows that, for a hybrid network, reducing flit width improves area efficiency with minimal performance loss.

4.3.4 Static vs. hybrid vs. dynamic networks. Figure 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 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 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 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 3.1.2), which results in fewer PBs, less data movement, and greater energy efficiency for hybrid networks.

In Figure 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.

5 RELATED WORK

Multiple decades of research have resulted in a rich body of literature, both in CGRAs [9, 55] and on-chip networks [24]. We discuss relevant prior work under the following categories:

5.1 Tiled Processor Interconnects

Architectures such as Raw [53] and Tile [59] use scalar operand networks [54], which combine static and dynamic networks. Raw has one static and two dynamic interconnects: the static interconnect is used to route normal operand traffic, one dynamic network is used to route runtime-dependent values which could not be routed on the static network, and the second dynamic network is used for cache misses and other exceptions. Deadlock avoidance is guaranteed only in the second dynamic network, which is used to recover from deadlocks in the first dynamic network. However, as described in Section 1, wider buses and larger flit sizes create scalability issues with two dynamic networks, including higher area and power. In addition, our static VC allocation scheme ensures deadlock freedom in our single dynamic network, obviating the need for deadlock recovery. The dynamic Raw network also does not preserve operand ordering, requiring an operand reordering mechanism at every tile.

TRIPS [50] is a tiled dataflow architecture with dynamic execution. TRIPS does not have a static interconnect, but contains two dynamic networks [19]: an operand network to route operands between tiles, and an on-chip network to communicate with cache banks. Wavescalar [52] is another tiled dataflow architecture with four levels of hierarchy, connected by dynamic interconnects that vary in topology and bandwidth at each level. The Polymorphic Pipeline Array [44] is a tiled architecture built to target mobile multimedia applications. While compute resources are either statically or dynamically provisioned via hardware virtualization support, communication uses a dynamic scalar operand network.

5.2 CGRA Interconnects

Many previously proposed CGRAs use a word-level static interconnect, which has better compute density than bit-based routing [60]. CGRAs such as HRL [14], DySER [18], and Elastic CGRAs [23] commonly employ two static interconnects: a word-level interconnect to route data and a bit-level interconnect to route control signals. Several works have also proposed a statically scheduled interconnect [12, 38, 56] using a modulo schedule. While this approach is effective for inner loops with predictable latencies and fixed initiation intervals, variable latency operations and hierarchical loop nests add scheduling complexity that prevents a single modulo schedule. HyCube [27] has a similar statically scheduled network, with the ability to bypass intermediate switches in the same cycle. This allows operands to travel multiple hops in a single cycle, but creates long wires and combinational paths and adversely affects the clock period and scalability.

5.3 Design Space Studies

Several prior studies focus on tradeoffs with various network topologies, but do not characterize or quantify the role of dynamism in interconnects. The Raw design space study [36] uses an analytical model for applications as well as architectural resources to perform a sensitivity analysis of compute and memory resources focused on area, power, and performance, without varying the interconnect. The ADRES design space study [6] focuses on area and energy tradeoffs with different network topologies with the ADRES [34] architecture, where all topologies use a fully static interconnect. KressArray Xplorer [22] similarly explores topology tradeoffs with the KressArray [32] architecture. Other studies explore topologies for mesh-based CGRAs [3] and more general CGRAs supporting resource sharing [28]. Other tools like Sunmap [37] allow end users to construct and explore various topologies.

5.4 Compiler Driven NoCs

Other prior works have used compiler techniques to optimize various facets of NoCs. Some studies have explored statically allocating virtual channels [29, 51] to multiple concurrent flows to mitigate head-of-line blocking. These studies propose an approach to derive deadlock-free allocations based on the turn model [16]. While our approach also statically allocates VCs, our method to guarantee deadlock freedom differs from the aforementioned study as it does not rely on the turn model. Ozturk et al. [42] propose a scheme to increase the reliability of NoCs for chip multiprocessors by sending packets over multiple links. Their approach uses integer linear programming to balance the total number of links activated (an energy-based metric) against the amount of packet duplication (reliability). Ababei et al. [1] use a static placement algorithm and an estimate of reliability to attempt to guide placement decisions for

NoCs. Kapre et al. [26] develop a workflow to map applications to CGRAs using several transformations, including efficient multicast routing and node splitting, but do not consider optimizations such as non-minimal routing.

6 CONCLUSION

In this work, we describe the mapping process from a high-level program to a distributed spatial architecture. We show that target application characteristics, compiler optimizations, and the underlying accelerator architecture must be considered when selecting a network architecture, and that static network bandwidth scales more efficiently. Overall, hybrid networks tend to provide the best energy efficiency by reducing data movement using static place and route, with a 2.3x improvement over the worst configuration. Both hybrid networks and static networks have a performance per area improvement of around 7x for pipelined CGRAs and 2x for time-scheduled CGRAs. Pure dynamic networks are unsuitable for CGRA architectures due to insufficient bandwidth.

Although it is possible to increase interconnect bandwidth, the resources necessary for each node increase simultaneously. When adding nodes, network area increases super-linearly because more network nodes are added, and bandwidth must scale to allow applications to use the larger network. Due to these network-based limitations, a spatially distributed array cannot be scaled simply by increasing the number of compute tiles. Therefore, CGRAs of the future may need to be spatially distributed across several chips, allowing higher-dimensional networks to be used; in the meantime, using static-dynamic hybrid networks can alleviate some of the challenges involved in building larger arrays on-chip.

ACKNOWLEDGMENTS

We thank Gedeon Nyengele and Kartik Prabhu for their assistance in simulation and data collection. This paper is based on research partially supported by a Herbert Kunzel Stanford Graduate Fellowship. This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7865. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This research is also supported in part by affiliate members and other supporters of the Stanford DAWN project: Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, Teradata, SAP and VMware.

REFERENCES

[1] Cristinel Ababei, Hamed Sajjadi Kia, Om Prakash Yadav, and Jingcao Hu. 2011. Energy and Reliability Oriented Mapping for Regular Networks-on-Chip. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*. ACM, 121–128. https://doi.org/10.1145/1999946.1999966

[2] Amazon AWS. [n. d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1.

[3] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. 2004. Network Topology Exploration of Mesh-based Coarse-Grain Reconfigurable Architectures. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1. 474–479. https://doi.org/10.1109/DATE.2004.1268891

[4] Daniel Ulf Becker. 2012. *Efficient Microarchitecture for Network-on-Chip Routers*. Ph.D. Dissertation. Stanford University, Palo Alto.

[5] Ivo Bolsens. 2006. Programming Modern FPGAs, International Forum on Embedded Multiprocessor SoC, Keynote,. http://www.xilinx.com/univ/mpsoc2006keynote.pdf.

[6] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. 2007. Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array. In *Reconfigurable Computing: Architectures, Tools and Applications*, Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13.

[7] Benton H. Calhoun, Joseph F. Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. 2010. Flexible Circuits and Architectures for Ultralow Power. *Proc. IEEE* 98, 2 (Feb 2010), 267–282. https://doi.org/10.1109/JPROC.2009.2037211

[8] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622. https://doi.org/10.1109/MICRO.2014.58

[9] Kiyoung Choi. 2011. Coarse-Grained Reconfigurable Array: Architecture and Application Mapping. *IPSJ Transactions on System LSI Design Methodology* 4 (2011), 31–46. https://doi.org/10.2197/ipsjtsldm.4.31

[10] William James Dally and Brian Patrick Towles. 2004. *Principles and Practices of Interconnection Networks*. Elsevier.

[11] Edsger W Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik* 1, 1 (1959), 269–271. https://doi.org/10.1007/BF01386390

[12] Grigoris Dimitroulakos, Michalis D Galanis, and Constantinos E Goutis. 2006. Exploring the Design Space of an Optimized Compiler Approach for Mesh-like Coarse-Grained Reconfigurable Architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 10–pp. https://doi.org/10.1109/IPDPS.2006.1639349

[13] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[14] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 126–137. https://doi.org/10.1109/HPCA.2016.7446059

[15] Eric Garcia and Maya Gupta. 2009. Lattice Regression. In *Advances in Neural Information Processing Systems*. 594–602.

[16] Christopher J. Glass and Lionel M. Ni. 1992. The Turn Model for Adaptive Routing. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 278–287. https://doi.org/10.1109/ISCA.1992.753324

[17] Seth C. Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: a coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. 28–39. https://doi.org/10.1109/ISCA.1999.765937

[18] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 503–514. https://doi.org/10.1109/HPCA.2011.5749755

[19] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. 2007. On-Chip Interconnection Networks of the TRIPS Chip. *IEEE Micro* 27, 5 (Sept. 2007), 41–50. https://doi.org/10.1109/MM.2007.90

[20] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2016. From Model to FPGA: Software-Hardware Co-design for Efficient Neural Network Acceleration. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. 1–27. https://doi.org/10.1109/HOTCHIPS.2016.7936208

[21] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. 2007. Avoiding Message-Dependent Deadlock in Network-Based Systems on Chip. *VLSI design* 2007 (2007). https://doi.org/10.1155/2007/95859

[22] Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. 2000. KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In *Proceedings 2000. Design Automation Conference (DAC)*. 163–168. https://doi.org/10.1109/ASPDAC.2000.835089

[23] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 171–180. https://doi.org/10.1145/2435264.2435296

[24] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. On-Chip Networks, Second Edition. *Synthesis Lectures on Computer Architecture* 12, 3 (2017), 1–210. https://doi.org/10.2200/S00772ED1V01Y201704CAC040

[25] Nan Jiang, James Balfour, Daniel U Becker, Brian Towles, William J Dally, George Michelogiannakis, and John Kim. 2013. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 86–96. https://doi.org/10.1109/ISPASS.2013.6557149

[26] Nachiket Kapre and André Dehon. 2011. An NoC Traffic Compiler for Efficient FPGA Implementation of Sparse Graph-Oriented Workloads. *International Journal of Reconfigurable Computing* 2011 (2011). https://doi.org/10.1155/2011/745147

[27] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 45, 6 pages. https://doi.org/10.1145/3061639.3062262

[28] Yoonjin Kim, Rabi N. Mahapatra, and Kiyoung Choi. 2010. Design Space Exploration for Efficient Resource Utilization in Coarse-Grained Reconfigurable Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 10 (Oct 2010), 1471–1482. https://doi.org/10.1109/TVLSI.2009.2025280

[29] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srinivas Devadas. 2009. Application-aware Deadlock-free Oblivious Routing. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/1555754.1555782

[30] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[31] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127. https://doi.org/10.1109/ISCA.2016.20

[32] Rainer Kress. 1996. A Fast Reconfigurable ALU for Xputers.

[33] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. FPGA Architecture: Survey and Challenges. *Found. Trends Electron. Des. Autom.* 2, 2 (Feb. 2008), 135–253. https://doi.org/10.1561/1000000005

[34] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application*, Peter Y. K. Cheung and George A. Constantinides (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[35] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/1168857.1168878

[36] Csaba Andras Moritz, Donald Yeung, and Anant Agarwal. 1998. Exploring Optimal Cost-Performance Designs for Raw Microprocessors. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. 12–27. https://doi.org/10.1109/FPGA.1998.707877

[37] Srinivasan Murali and Giovanni De Micheli. 2004. SUNMAP: a Tool for Automatic Topology Selection and Generation for NoCs. In *Proceedings. 41st Design Automation Conference, 2004.* 914–919. https://doi.org/10.1145/996566.996809

[38] Chris Nicol. [n. d.]. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. https://wavecomp.ai/wp-content/uploads/2018/12/WP_CGRA.pdf

[39] Juanjo Noguera, Chris Dick, Vinod Kathail, Gaurav Singh, Kees Vissers, and Ralph Wittig. 2018. Xilinx Project Everest: 'HW/SW Programmable Engine' *(Hot Chips 30)*. http://www.hotchips.org/hc30/2conf/2.03_Xilinx_Juanjo_XilinxSWPEHotChips20180819.pdf

[40] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 416–429. https://doi.org/10.1145/3079856.3080255

[41] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-Defined Accelerator for LargeScale DNN Systems *(Hot Chips 26)*. https://doi.org/10.1109/HOTCHIPS.2014.7478821

[42] Ozcan Ozturk, Mahmut Kandemir, Mary J Irwin, and Sri HK Narayanan. 2010. Compiler Directed Network-on-Chip Reliability Enhancement for Chip Multiprocessors. *ACM Sigplan Notices* 45, 4 (2010), 85–94. https://doi.org/10.1145/1755951.1755902

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

[44] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 370–380. https://doi.org/10.1145/1669112.1669160

[45] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. 2005. A Detailed Power Model for Field-programmable Gate Arrays. *ACM Trans. Des. Autom. Electron. Syst.* 10, 2 (April 2005), 279–302. https://doi.org/10.1145/1059876.1059881

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

[47] Robert Clay Prim. 1957. Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal* 36, 6 (1957), 1389–1401.

[48] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. https://doi.org/10.1145/2678373.2665678

[49] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. 2010. An Analysis of On-chip Interconnection Networks for Large-scale Chip Multiprocessors. *ACM Trans. Archit. Code Optim.* 7, 1, Article 4 (May 2010), 28 pages. https://doi.org/10.1145/1736065.1736069

[50] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 422–433. https://doi.org/10.1109/ISCA.2003.1207019

[51] Keun Sup Shim, Myong Hyon Cho, Michel Kinsy, Tina Wen, Mieszko Lis, G. Edward Suh, and Srinivas Devadas. 2009. Static Virtual Channel Allocation in Oblivious Routing. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. 38–43. https://doi.org/10.1109/NOCS.2009.5071443

[52] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar Architecture. *ACM Trans. Comput. Syst.* 25, 2, Article 4 (May 2007), 54 pages. https://doi.org/10.1145/1233307.1233308

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

[54] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. 2003. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 341–353. https://doi.org/10.1109/HPCA.2003.1183551

[55] Russel Tessier, Kenneth Pocek, and André DeHon. 2015. Reconfigurable Computing Architectures. *Proc. IEEE* 103, 3 (March 2015), 332–354. https://doi.org/10.1109/JPROC.2014.2386883

[56] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. Static Versus Scheduled Interconnect in Coarse-Grained Reconfigurable Arrays. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 268–275. https://doi.org/10.1109/FPL.2009.5272293

[57] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A Memory System Simulator. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 100–107. https://doi.org/10.1145/1105734.1105748

[58] Xiaohang Wang, Peng Liu, Mei Yang, and Yingtao Jiang. 2013. Avoiding Request–Request Type Message-Dependent Deadlocks in Networks-on-Chips. *Parallel Comput.* 39, 9 (2013), 408–423. https://doi.org/10.1016/j.parco.2013.05.002

[59] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27, 5 (Sept. 2007), 15–31. https://doi.org/10.1109/MM.2007.89

[60] Andy Ye and Jonathan Rose. 2006. Using Bus-based Connections to Improve Field-Programmable Gate-array Density for Implementing Datapath Circuits. *IEEE Trans. Very Large Scale Integr. Syst.* 14, 5 (May 2006), 462–473. https://doi.org/10.1109/TVLSI.2006.876095