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 (WIP)

With the slowdown of Moore's Law, specialized hardware accelerators are gaining tractions as energy-efficient platforms that deliver 100-1000x performance improvement over general-purpose processors. Reconfigurable architectures are particularly promising in providing high-throughput and low-latency computation in streaming and data-intensive analytics applications. Instead of dynamically executing instructions like in processor architectures, reconfigurable architectures have flexible datapath that can be statically configured to parallelize and/or pipeline the program spatially across on-chip resources. The pipelined execution model and explicitly-managed scratchpad in reconfigurable accelerators dramatically reduce the performance, area, and energy overhead in dynamic execution and conventional cache memory hierarchy, respectively. Plasticine is a hierarchical coarse-grained reconfigurable dataflow accelerator introduced at Stanford in 2017. Compared to fine-grained reconfigurable architecture, like FPGAs, Plasticine has shown 76x performance/watt benefit due to the reduction in routing overhead and the improvement in on-chip resource density.

In this work, we focus two aspects of the software-hardware codesign that impact the accessibility and performance of the accelerator. One of the biggest challenges that hinders the adoption of these accelerators is the low-level declarative configuration interface that requires the programmers to have detailed knowledge about the underlying microarchitecture implementation and hardware constraints. To address the challenge, I will introduce a compiler stack that provides a high-level programming interface that efficiently translates

imperative control constructs to streaming dataflow execution with minimum synchronization overhead on an on-chip distributed architecture. The compiler handles the hardware constraints systematically with resource virtualization. Next, I will present a comprehensive study on the on-chip network design for reconfigurable dataflow architectures that sustain performance in a scalable fashion with high energy efficiency.

Acknowledgements

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction (WIP)

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[**?**].

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 [**?**, 3, **?**] and can be rented on the AWS F1 cloud [**?**]. 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 [**?**, **?**, **?**, **?**]. 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

[**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, 4].

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, CGRA data paths now contain increasingly coarse-grained processing blocks such as pipelined, vectorized functional units [4, **?**, **?**]. These data paths typically have a vector width of 8–16x [4], 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 [**?**], 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.

Chapter 2

Background (WIP)

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 pipelined access reduce off-chip access. HBM small capacity scratchpad improves effective bandwidth and capacity

Figure 2.1 shows an example program executed on the

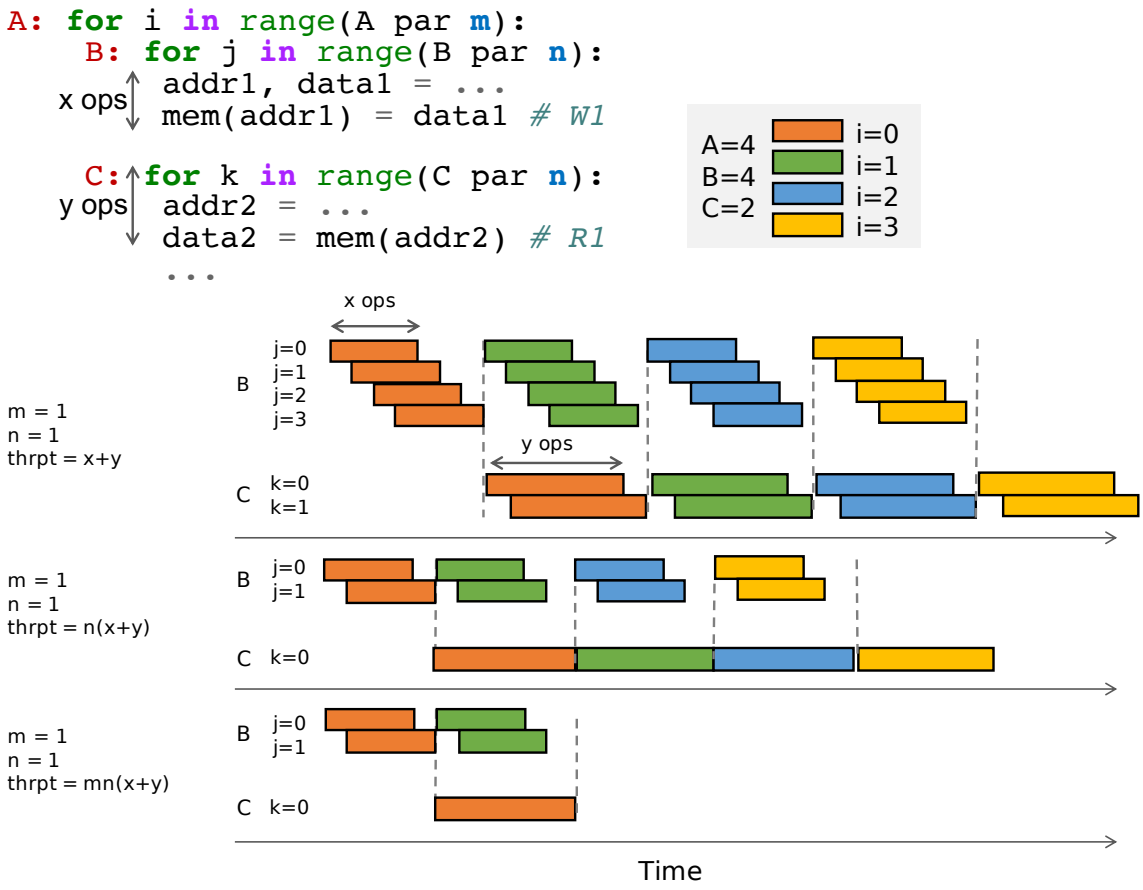


Figure 2.1: Hiearchical pipelining and parallelization in spatial architecture. (a) illustrates the runtime and throughput of a hiearchically pipelined and parallelized program on a reconfigurable spatial architecture. At inner level, instructions within each basic block are fine-grained pipelined across 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 $(x + y)mn$.

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

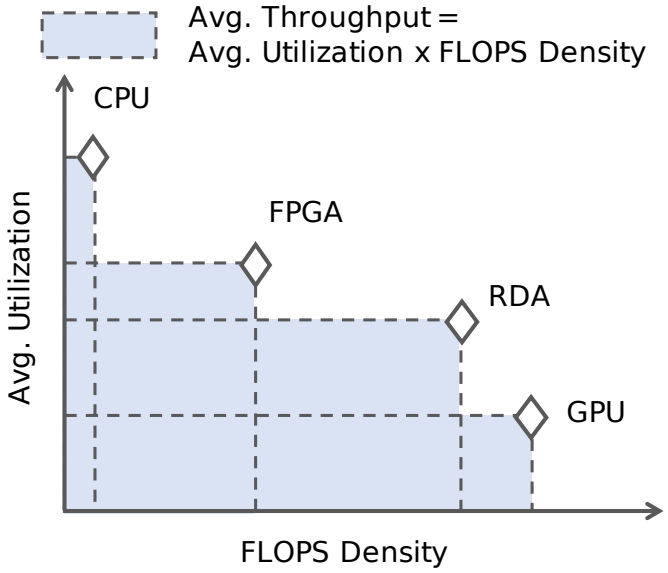


Figure 2.2: Average utilization vs. peak compute density tradeoff among different architectures.

$$\text{thrpt}_{\text{app}} = \min\left(\text{thrpt}_{\text{comp}}, \frac{\text{comp}}{\text{access}_{\text{off}}}\text{BW}_{\text{off}}, \frac{\text{comp}}{\text{access}_{\text{on}}}\text{BW}_{\text{on}}, \frac{\text{comp}}{\text{trans}_{\text{net}}}\text{BW}_{\text{net}}\right)$$

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

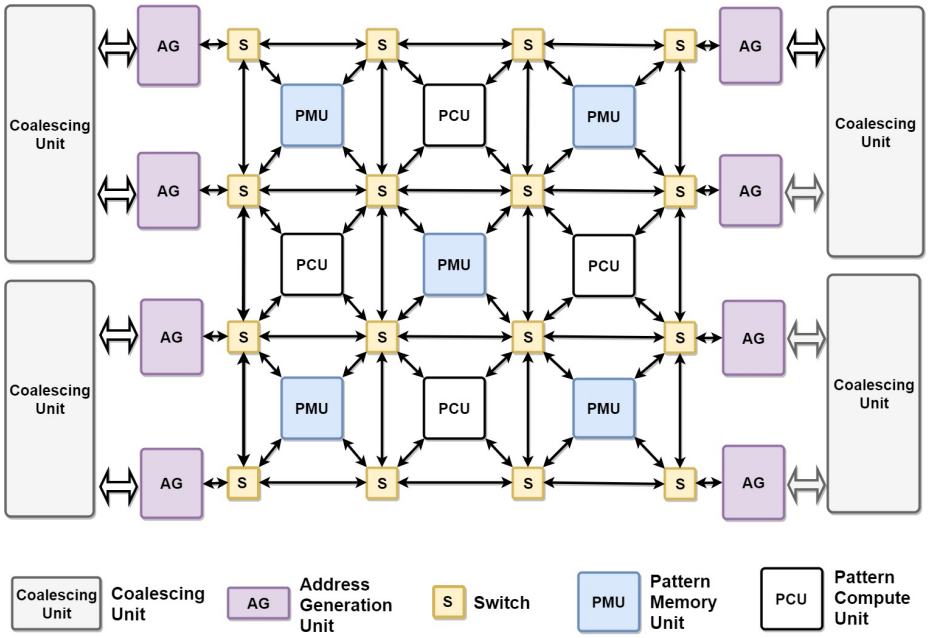


Figure 2.4: Plasticine chip-level architectural diagram

```
// 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 2.5: 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 [5]. 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 2.5. 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 [6].

Chapter 3

Compiler

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

In the following sections, Section 3.4 describes conversion from an imperative paradigm with a nested control hierarchy to the distributed streaming dataflow execution. Section 3.5 details program-partitioning passes that decompose program over distributed resources. Section 3.6 enumerates several optimizations in SARA, and Section 3.7 discuss about PaR and heuristic generation.

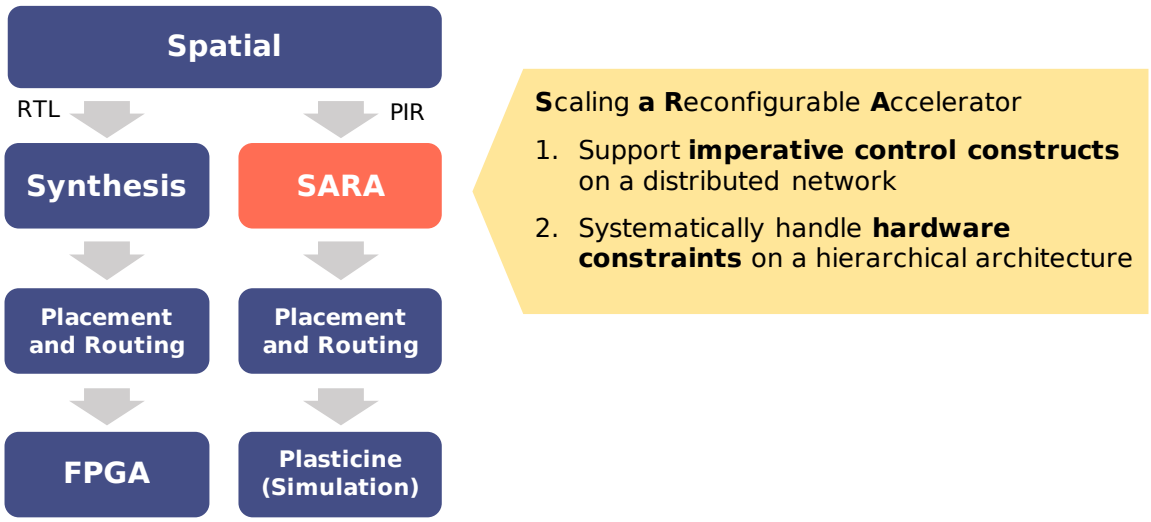


Figure 3.1: Spatial Stack

3.1 High-Level Compiler (WIP)

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.

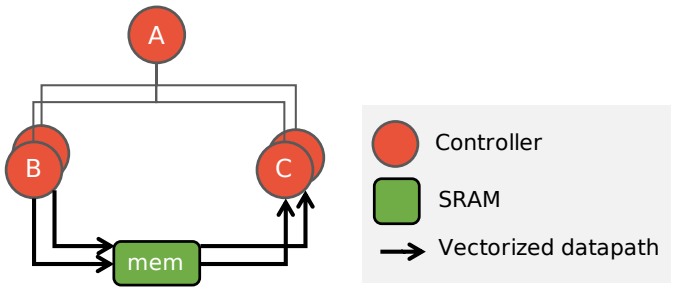
Spatial currently does not capture procedure calls in the IR; all functions are inlined and does not share resources. A future work is to use synchronization mechanism in SARA to implement true procedure call on a spatial architecture.

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

The input to SARA is the backend of the Spatial IR, which is an control hierarchy after loop unrolling. The controller at each level of the hierarchy corresponds to a control primitive, such as a loop, or a branch statement. A basic block is attached to each *inner most* controller including instructions and memory accesses to user declared data-structures. Figure 3.2 shows an example of a program and a schematic Spatial output IR. If the program has instructions outside a outer loop, Spatial automatically inserts a *unit* controller that wraps the floating instructions.

```
mem = OnchipArray(dims=[100])
A: for i in range(A,par=2):
    B: for j in range(B,vec=16):
        addr1, data1 = ...
        mem(addr1) = data1 # W1
    C: for k in range(C,vec=16):
        addr2 = ...
        data2 = mem(addr2) # R1
        ...
```

(a) Pseudo Spatial example



(b) Schematic Spatial IR

Figure 3.2: Pseudo example of SARA's front-end language. (a) shows the an example of SARA's front-end language. The actual front-end Spatial is a Scala-embeded DSL. For simplicity and generality, we use a python style pseudo code to show an language with similar abstraction. The 'par' keyword indicates outer loop unrolling factor, and the 'vec' keyword is followed by a inner-loop vectorization factor. When an iterator is vectorized, instructions using the vectorized iterator is automatically vectorized as well. When unrolling the outer loop `A`, the enclosed loop body and next-level control hierarchy are duplicated, as suggested in (b). Each loop in (a) corresponds to a controller in (b). The inner most controllers `B` and `C` each contain a basic block within instructions within the inner most loops.

3.2 Low-level Native Programming Interface of Plasticine (Ready)

Like high-level synthesis tools for FPGAs, SARA starts with a high-level imperative programming abstraction and synthesizes the program to execute on a reconfigurable accelerator. Targeting an RDA, however, is 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 a collection of memory tiles (PMUs) and compute tiles (PCUs). The global mesh 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 workhorse 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. The 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. 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. Figure 3.3a shows an example of a 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

```
# scalar and vector input streams
bufferA = VecInSteram()
bufferB = ScalInSteram()
bufferC = VecInSteram()
outputD = VecOutSteram()

i = Counter(min=0,max=3,stride=1)
j = Counter(min=0,max=3,stride=1)
chain = CounterChain(i,j)

a = bufferA.deq(when=j.valid())
b = bufferB.deq(when=i.done())
c = bufferC.deq(when=j.done())

forward(stage=0, dst="PR0", src=c)
# b is broadcasted to all lanes
stage(0,"mul",dst="PR1", b, c)
# Operands are PRs from the previous stage
stage(1,"add",dst="PR2",oprd=["PR0","PR1"])

# forwarding PR2 of stage 1 to stage 2
forward(stage=2,dst="PR2",src="PR2")
forward(stage=3,dst="PR2",src="PR2")
forward(stage=4,dst="PR2",src="PR2")
forward(stage=5,dst="PR2",src="PR2")
forward(stage=0,dst="PR3",src=j.valid())
forward(stage=1,dst="PR3",src="PR3")
forward(stage=2,dst="PR3",src="PR3")
forward(stage=3,dst="PR3",src="PR3")
forward(stage=4,dst="PR3",src="PR3")
forward(stage=5,dst="PR3",src="PR3")
outputD.enq(data="PR2", when="PR3")
```

(a) Declarative configuration

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    b = bufferB.deq()
    i = Counter(0,3,1)
    j = Counter(0,3,1)
    ctx.chain(i,j)
    b = bufferB.deq(when=i.done())
    c = bufferC.deq(when=j.done())
    a = bufferA.deq(when=j.valid())

    expr = a * b + c
    outputD.enq(data=expr, when=j.valid())
```

(b) Declarative configuration with automatic register allocation

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    b = bufferB.deq()
    for i in range(0, 3, 1)
        c = bufferC.deq()
        for j in range(0, 3, 1)
            a = bufferA.deq()
            expr = a * b + c
            outputD.enq(expr)
```

(c) Equivalent imperative program

Figure 3.3: Pseudo PCU configuration. (a) shows the simplified declarative-style configuration for the SIMD pipeline context in a PCU. Each PCU context can produce a set of output streams as a function of input streams and counter values. Each stage contains multiple pipeline registers (PRs) that can propogate results across stages. A stage can read PRs from the previous stage and write to PRs in its current stage. Only the first stage can read streams and counter values and the last stage can write streams. Other stages need to propogate the required values through PRs. (b) shows a simplified configuration where registers are implicitly allocated. By configuring when each stream is enqueued and dequeued using signals from the chained counters, these streams are effectively read and written within different loop bodies. (c) shows the effective execution achieved in an imperative program. In (b), the `valid` signal of the inner most counter `j` is high whenever the context is enabled. The context is implicitly triggered whenever its input streams `streamA`, `streamB`, and `streamC` are non-empty and output stream `outputD` is ready. In (b) and (c), we move the definitions of streams outside of the context, so they can be written by other contexts. Each stream can have exactly one writer. If a stream has more than one reader, effectively the writer broadcasts the result to both input buffers of the receiver contexts.

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 dataflow architectures, the dataflow engine in Plasticine is more flexible in that it allows dynamic enqueue and dequeue window for its input and output streams. *This feature enables Plasticine to support complex control hierarchy, such as non-perfectly nested loops and branch statements, across contexts even though individual contexts can only execute instructions that are control-free*.

Figure 3.3c represents the effective execution achieved in an imperative-style program. For simplicity, we will use the imperative-style configuration in the later discussion. However, it is important to realize the instructions in the imperative-style configurations are not executed in time, but rather in space. There is a one-to-one translation from the declarative-style configuration to the imperative-style, but not in a reverse way. Instructions in the contexts must belongs to a single basic block, and the loops need to be perfectly nested (only accesses to streams can occur in the outer loops).

There is no global scheduler to orchestrate the execution order among contexts—the execution is purely streaming and dataflow driven. The only way to order the execution of two independent contexts is to introduce a control **token** between two contexts acting like a dummy data-dependency. This restriction eliminates the need for long-traveling wires and communication hot spots 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 the 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 the total resources consumed by all contexts do not exceed the 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 contexts such that the memory access order is consistent with the program order from a high-level imperative programming language.

Unlike most accelerators at this scale, Plasticine does not have any shared global on-chip memory. This design dramatically improves 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 a 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 unpredictable latency on both control and data path. One of the major contributions of SARA is to hide these synchronization burdens from the programmer and still provide a programming abstraction of logical memories with configurable bandwidth and arbitrary capacity that fits on-chip.

DRAM Interface

The Plasticine architecture provides access to four DDR channels on the two sides of the tiled array, as shown in Figure 2.4. Each side has a column of DRAM address generator (DAG) specialized in generating off-chip requests. Like compute pipeline in PMUs, the pipeline in DAG is also non-vectorized with integer arithmetics. Each DAG can generate 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 responses within each stream and no ordering guaranteed across streams. In streaming pipelined execution, the program can use multiple streams for DRAM accesses appeared in different locations of the program. To provide off-chip memory consistency, SARA allocates synchronizations across these streams to preserve memory order expected by the program.

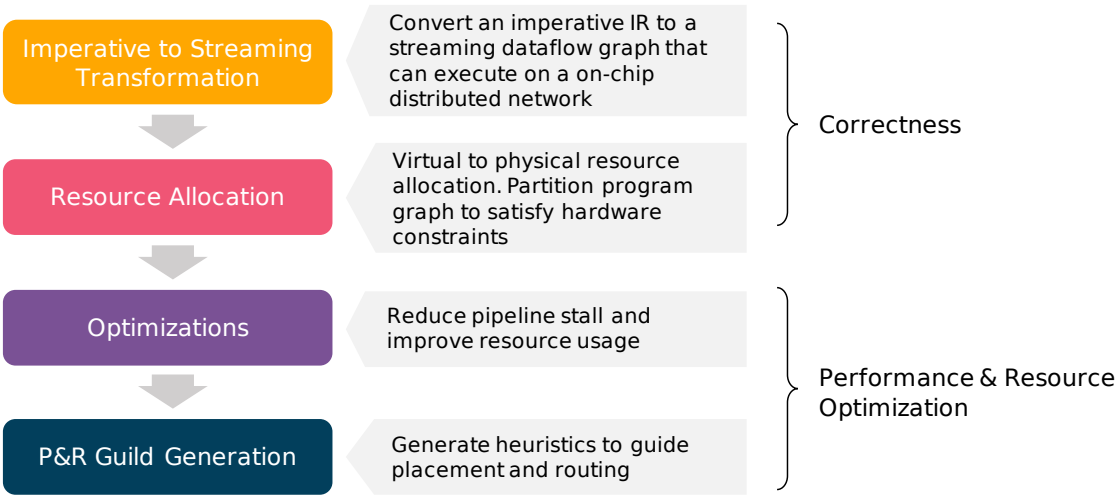


Figure 3.4: SARA Compiler Flow

3.3 Compiler Overview (Ready)

In this section, we describe a systematic approach to compile applications described in an imperative front-end language with a purely declarative and distributed dataflow graph that can run on Plasticine. Figure 3.4 shows the compilation flow to perform this translation between the two paradigms. At high-level, SARA allocates distributed on-chip resources to execute the program in spatially parallelized and pipelined fashion. SARA automatically generates synchronization across distributed units to provide strong memory consistency for an imperative program, on an architecture that does not guarantee memory ordering across multiple access streams. This synchronization is purely point-to-point, introducing minimum performance overhead, which maximizes the scalability of the architecture. SARA further virtualizes resource allocation and hides the underlying resource constrains on this hierarchical architecture from the programmers.

First, SARA takes the input Spatial IR and performs the **imperative to streaming transformation**. A virtual unit (VU) is our intermediate representation that captures computations within the boundary of a physical unit (PU), such as a PCU and PMU. Each VU can contain multiple contexts. However, a VU can be mapped to a PU only if the aggregated resource usage of all contexts in the VU can fit in the PU. The hardware can also

limit the maximum number of contexts a PU can support and has resources that cannot be split across contexts. Additionally, messages across VUs are mapped across the global network, and must tolerate an arbitrary amount of network latencies; messages within a single VU across contexts takes only a single cycle. The transformation phase generates a virtual unit dataflow graph (VUDFG) with appropriate synchronizations, such that the on-chip distributed pipeline produces the same result as a parallelized program executed in time.

At the end of the allocation phase, a virtual unit can consume as much resources as the program requires. The second phase is **resource allocation**, where SARA assign each VU to a PU that processes the required resources. If no PU can execute a VU, SARA tries to partition the VU into multiple VUs to eliminate constraint violations. If there is insufficient PU or the VU cannot be partitioned, the mapping process fails with appropriate hints to the programmer for the limiting resources.

Throughout the first two phases, SARA introduces various **optimizations** that either reduce the resource cost of the VUDFG, or alleviates potential performance bottleneck in streaming pipelined execution. After all VU fits in at least one type of PU, SARA performs a global optimization that merges small VUs into a larger VU to reduce resource fragmentations.

The output of the resource allocation phase is a VUDFG with a tagged PU type for each VU. It is up to the **placement and routing (PaR)** phase to determine where the VU will be finally placed. Additionally, SARA performs static analysis on the traffic pattern and generate heuristics for PaR to reduce runtime congestion.

```
mem = zeros(N)
a = rand(N)
b = rand(N)
for i in range(1, N):
    tmp = a[i] * b[i]
    mem[i] = tmp + i
```

(a) Input program

```
mem = zeros(N)
a = rand(N)
b = rand(N)
tmp = zeros(N)
for i in range(1, N):
    tmp[i] = a[i] * b[i]
for i in range(1, N):
    mem[i] = tmp[i] + i
```

(b) Loop Fission

```
mem = zeros(N)
a = rand(N)
b = rand(N)
tmp = queue()
@concurrent
for i in range(1, N):
    tmp.enq(a[i] * b[i])
@concurrent
for i in range(1, N):
    mem[i] = tmp.deq() + i
```

(c) Loop Division

Figure 3.5: (b) and (c) shows the output of loop fission and loop division of the input program (a), respectively. In (b), the first loop is executed entirely before executing the second loop. The intermediate result `tmp` is materialized into an array with the same size as the loop range. In (b), the two loops can execute concurrently. The intermediate result is materialized into a queue. For each iteration, a loop can execute only if all of its queues are non-empty. The second loop can execute as soon as `tmp` receives the first element.

3.4 Imperative to Streaming Transformation

3.4.1 Loop Division (Ready)

Between the front-end and back-end abstraction of SARA, an obvious gap between the two abstractions is that the front-end imperative language can contain arbitrarily nested control hierarchy, whereas the hardware compute engine can only execute perfectly nested loops. To address this issue, we introduce a new type of loop transformation—loop division—for streaming reconfigurable accelerators. Similar to loop fission, loop division breaks a single loop into multiple loops. The difference is that loop fission generates a sequence of sequentially executed loops, whereas loop division generates loops executing *concurrently*. Additionally, loop fission materializes the intermediate results across fissioned loops into arrays, while loop division use queue to communicate across loops. Each loop generated from loop division can only execute if all of their input queues are not empty. Figure 3.5 gives an example of a loop fusion vs. loop division.

When executing loop division on a single-threaded CPU, the CPU must context switching between the concurrent loops and executing the one with cleared input dependencies.

```
mem = rand(N)
for i in range(1, N):
    tmp = mem[i-1] + i
    mem[i] = tmp
```

(a) Input program

```
mem = rand(N)
tmp = zeros(N)
for i in range(1, N):
    tmp[i] = mem[i-1] + i
for i in range(1, N):
    mem[i] = tmp[i]
```

(b) Invalid Loop Fission

```
mem = rand(N)
tmp = queue()
# Initialize queue with
# 1 dummy element
backward = queue(init=1)
@concurrent
for i in range(1, N):
    tmp.enq(mem[i-1] + i)
    backward.deq()
@concurrent
for i in range(1, N):
    mem[i] = tmp.deq()
    backward.enq(0)
```

(c) Loop Division

Figure 3.6: Example of an illegal loop fission and a legal loop division

Like loop fission, loop division is likely worsening the performance on a processor architecture, as the worst-case memory footprint of the intermediate result `tmp` increases from $O(1)$ to $O(N)$. On RDAs, the divided loops are executing concurrently in a streaming pipelined fashion. The size of the `tmp` can be limit to a small fixed size, efficiently implemented with a hardware FIFO. Although loop transformations are generally optimizations on CPUs, loop division is a required transformation to converts an infeasible program to a feasible one for Plasticine.

Loop fission is not always safe, as it may alter the execution order of the program. Loop division, on the other hand, does not change the underlying data-dependency and is always safe. To achieve this, loop division needs to introduce additional dummy data dependencies across divided loops to enforce the correct execution order. Figure 3.6 gives an example of an invalid loop fission and a correct loop division. **??** gives more detail on how SARA automatically generates the dummy data-dependencies.

3.4.2 Virtual Context Allocation (Ready)

As a start, SARA allocates one virtual memory to hold each on-chip data structure, and one context to execute each basic block within the innermost controllers. A basic block maps naturally to a context, as instructions within a basic block are control-free. Next, SARA makes a copy of all controllers enclosing the basic block in the corresponding context;

```
mem = array(dims=[100])
A: for i in range(A):
    B: for j in range(B):
        addr1, data1 = ...
        mem(addr1) = data1 # W1
    C: for k in range(C):
        addr2 = ...
        data2 = mem(addr2) # R1
        ...
```

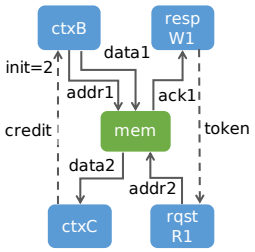
(a) Pseudo input example

```
mem = VirtualMemory(dims=[100])

with Context() as ctxB:
    A: for i in range(A):
        B: for j in range(B):
            addr1, data1 = ...
            mem(addr1) = data1 # W1

with Context() as ctxC:
    A: for i in range(A):
        C: for k in range(C):
            addr2 = ...
            data2 = mem(addr2) # R1
            ...
```

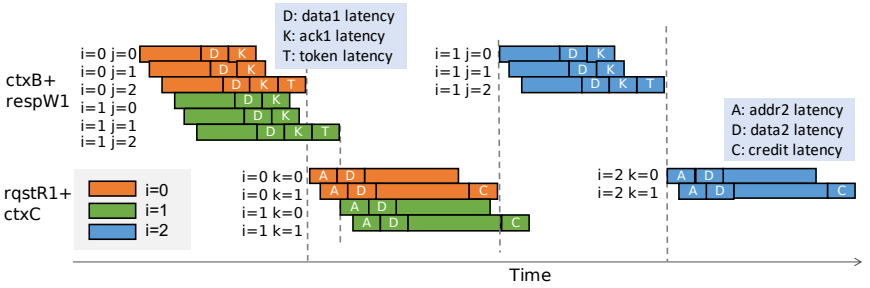
(b) Context allocation

```
mem = VirtualMemory(dims=[100])
token = ControlStream()
credit = ControlStream(init=2)

with Context() as ctxB:
    A: for i in range(A):
        B: for j in range(B):
            addr1, data1 = ...
            mem.addr.enq(addr1)
            mem.data.enq(data1)
        credit.deq()
with Context() as respW1:
    A: for i in range(A):
        B: for j in range(B):
            ack1 = mem.ack.deq()
        token.enq()

with Context() as rqstR1:
    A: for i in range(A):
        C: for k in range(C):
            addr2 = ...
            mem.reqst.enq(addr2)
        token.deq()
with Context() as ctxC:
    A: for i in range(A):
        C: for k in range(C):
            data2 = mem.resp.deq()
            ...
        credit.enq()
```

(c) Request and response division



(d) Context Graph



(e) Timing on Plasticine

Figure 3.7: Lowering example. SARA allocates one context per basic block for `B` and `C`, shown in (b). Outer controller `A` is duplicated in both `ctxB` and `ctxC`. (c) SARA seperates out requestion generator `rqstR1` from `ctxB` for `R1` and response receiver from `ctxC` `respW1` for `W1`. The resulting dataflow graph is shown in (d). To enforce the forward data-dependency between `W1` and `R1`, SARA allocates a forward token between `W1`'s response receiver `respW1` and `R1`'s request generator `rqstR1`; to enforce the loop-carried WAR dependency between `R1` and `W1`, SARA allocates a backward token (`credit`) between `R1`'s response receiver `ctxC` and `W1`'s request generator `ctxB`. The backward credit is initialized with two elements because `mem` is double-buffered. On the writer side, a forward `token` is a produced and a backward `credit` is consumed every `B` iterations; on the reader side, a forward `token` is consumed and a backward credit is produced every `C` iterations. The resulting timing of the execution is shown in (e).

these controllers are later converted to counters and control configurations supported by the hardware. Effectively, SARA performs loop division such that all basic blocks are perfectly nested. Figure 3.7 shows an example of the context allocation. With these controllers, contexts can repeat execution for expected number iterations. However, data-structures written and read by different contexts are accessed in random order. The insight is that *as long as all contexts accessing a shared memory with expected program order, the final result is identical to a sequentially executed program*. Unlike traditional out-of-order execution, where hardware and compiler look for independent instructions to execute concurrently, SARA starts with executing *all* basic blocks in a program in *concurrent* contexts. Next, SARA introduces synchronizations to maintain consistent access order as expected by the program *only* among contexts accessing a shared memory. This way, SARA introduces minimum p2p synchronizations among small groups of contexts; contexts accessing different memories are naturally parallelized without impacting the final output.

3.4.3 Control Allocation (WIP)

Starting with all contexts execute in parallel, SARA introduces **control token**s across contexts to serialize their execution order based on the program order. This control token is no different from a regular data-dependency and can be viewed as an access grant to the shared memory across contexts. By controlling *where*, *how*, and *when* to pass the token, SARA is able to maintain a consistent update ordering between the pipelined and parallelized contexts that access the shared memory.

We refer to a memory access appeared in the input IR as a *declared access*, as supposed to memory accesses executed at runtime. `W1` and `R1` are examples of two declared accesses in Figure 3.7a In the rest of this section, we will walk through how SARA allocates control tokens to maintain sequential consistency on Plasticine.

Where. During control allocation, SARA examines all declared accesses of a memory and checks for dependency across these accesses. SARA only allocate resource to synchronize two contexts if they contain declared accesses that can potentially interfere. Whether two

Data structure	Memory type
array (fit on-chip)	SRAM
array (not fit on-chip)	DRAM
scalar variable	register
queue	FIFO

Table 3.1: Mapping between user declared data-structure to underlying hardware memories. Programmers explicitly specify the desired hardware type inside Spatial. In other languages, this table specifies a mapping between software data-structures and hardware memory types on Plasticine.

Memory type	DRAM	SRAM	FIFO	Register
read-after-read (RAR)	✗	✓	✓	✓
read-after-write (RAW)	✓	✓	✓	✓
write-after-read (WAR)	✓	✓	✓	✓
write-after-write (WAW)	✓	✓	✓	✓

Table 3.2: Interference table for whether two accesses interfere for each memory type.

declared accesses interfere depends on

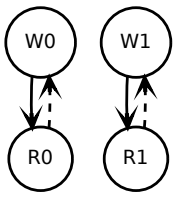
- the type of accesses (read vs. write)
- the type of the memory (e.g. SRAM, DRAM)
- and location of the declared accesses in the control hierarchy.

Table 3.1 lists hardware memories available on reconfigurable architectures and software data-structures providing similar program semantics. The type of the memory matters because they have different programming interface on the hardware. For instance, two DRAM read accesses do not interfere because the DRAM interface permits multiple concurrent access streams through multiple DAGs[1]. Therefore, from the programmer's perspective, users do not need to serialize the two contexts reading the same DRAM address. SRAMs, on the other hand, have a single read and write port. Programmers must guarantee that a PMU receives read requests from a single context at any point in time for

[1] All DAGs can access the full DRAM address space

```
mem = OnchipArray(dims=[100])
A: for i in range(A,par=2):
    ## A0: lane 0 of loop A
    B: for j in range(B):
        mem(addr1) = data1 # W0
    C: for k in range(C):
        data2 = mem(addr2) # R0
    ## A1: lane 1 of loop A
    B: for j in range(B):
        mem(addr1) = data1 # W1
    C: for k in range(C):
        data2 = mem(addr2) # R1
```

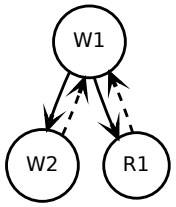
(a) Pseudo example



(b) Dependency Graph

```
mem = OnchipArray(dims=[100])
A: for i in range(A):
    B: mem(addr1) = ... # W1
    C: if cond:
        D: mem(addr2) = ... # W2
       else:
        E: ... mem(add3) # R1
```

(c) Pseudo example



(d) Dependency Graph

Figure 3.8: Dep Graph (WIP)

correctness. Table 3.2 shows the interference relation between different types of memory across accesses.

For every declared access, SARA checks on other accesses appeared earlier in the program order for a possible forward dependency, and later in the program order for a possible loop-carried dependency (LCD). In the example in Figure 3.7a, there is a forward data dependency between `W1` and `R1`, and a write-after-read LCD between `R1` and `W1`. SARA builds a dependency graph between declared accesses for each memory. Figure 3.8 gives more examples of how SARA determines dependencies based on where these accesses are declared in the control hierarchy.

Enforcing all dependencies in the dependency graph may not be necessary, as enforcing a subset of dependencies can be sufficient to enforce the order of the whole graph. 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

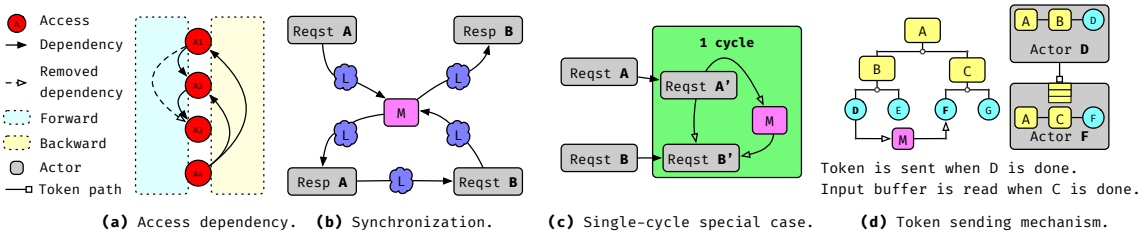


Figure 3.9: (a) Access dependency graph. (b) Synchronization of two accesses on the same memory. (c) Single-cycle special case. (d) Actors uses local states of controller hierarchy to determine when to send a token.

NP-hard, SARA 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.

How. 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 context 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 context 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 context that accumulates the response of access A ($resp_A$) and the context that generates requests for access B ($reqst_B$) (Figure 3.9(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 type of memory (including DRAM and on-chip memories) with unpredictable access latency.

When. SARA configures the contexts 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 contexts are active. For register, SRAM, and DRAM, the program order expects that the producer and consumer write and inspect 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 3.9(d)). The *done* signals of these controllers are used to produce and consume the token in contexts, independently.

Specialization for dense on-chip array 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 3.9(c)). Instead of synchronizing between $resp_A$ and $reqst_B$, we allocate two stateless contexts $reqst'_A$ and $reqst'_B$ within the 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.

Specialization for non-indexable memory 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

```
A: for i in range(N):
    B: for j in range(2*i, N):
        ... = i + j
```

(a) Input program

```
mem = array()
bound = ScalarStream()
with Context() as ctxA:
    A: for i in range(N):
        bound.enq(2*i)
with Context() as ctxB:
    A: for i in range(N):
        start = bound.deq()
        B: for j in range(start, N):
            ... = i + j
```

(b) Context graph

Figure 3.10: (a) shows an example program with dynamic loop range. Expressions to generate the loop bounds belongs to a basic block that gets mapped to `ctx1`. SARA maps the loop bound as a data-dependnecy to context that maps the inner loop context, and configures dequeue signal of `bound` stream to counter `B.done`.

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.

3.4.4 Data-Dependent Control Flow (Ready)

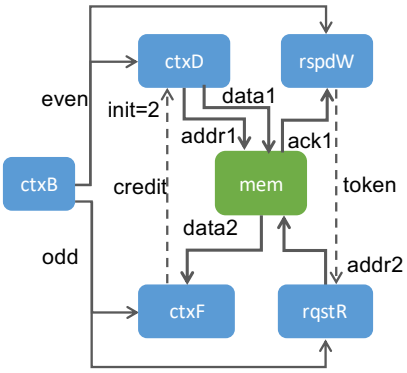
Using the synchronization discussed in Section 3.4.3, SARA can support control constructs that typically are not supported on dataflow accelerators, such as branches and while loops. Most dataflow accelerators do not support control divergence. SIMT architectures, like GPUs, implement branches with predications and pay the latency penalty of both branch cases. To enable these flexible control, the control path of the architecture must permit data dependencies. **??** details the required changes in the control path to support these features.

Dynamic Loop Range Figure 3.10 shows an example of loops with data-dependent ranges. SARA uses a context to compute the loop bounds, which are treated as input dependency to the context that maps the loop body.

Branch Condition SARA supports both predictions and actual branches. Branches within the innermost loops are implemented with predications so that the loop can still be vectorized across SIMD lanes. This is very similar to a SIMT architecture, where all lanes execute the *if* clause followed by the *else* clause, masking off the memory access for the disabled

```
A: for i in range(A):
    B: even = i % 2 == 0
    C: if even:
        D: for j in range(D):
            addr1, data1 = ...
            mem(addr1) = data1 # W
       else:
        F: for k in range(F):
            addr2 = ...
            data2 = mem(addr2) # R
            ...
```

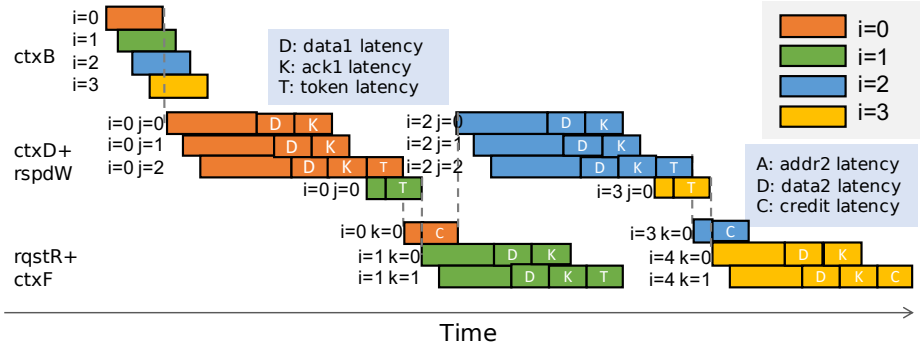
(a) Input program



(b) Dataflow graph

```
even = ScalarStream()
odd = ScalarStream()
token = ControlStream()
credit = ControlStream(init=2)
with Context() as ctxB:
 A: for i in range(A):
     B: tmp = i % 2 == 0
        even.enq(tmp)
        odd.enq(not tmp)
with Context() as ctxD:
 A: for i in range(A):
     C: if even.deq():
         D: for j in range(D):
             addr1, data1 = ...
             mem.waddr.enq(data1)
             mem.wdata.enq(data1)
        credit.deq()
with Context() as rpstW:
 A: for i in range(A):
     C: if even.deq():
         D: for j in range(D):
             pass
        token.deq()
with Context() as rqstR:
 A: for i in range(A):
     C: if odd.deq():
         F: for k in range(F):
             addr2 = ...
             mem.raddr.deq(addr2)
        token.deq()
with Context() as ctxF:
 A: for i in range(A):
     C: if odd.deq():
         F: for k in range(F):
             data2 = mem.rdata.deq()
             ...
        credit.enq()
```

(c) Context configuration



(d) Timing

Figure 3.11: An example program with a branch. (d) shows the timing of execution with $A = 4$, $D = 3$, and $F = 2$.

lanes. The total number of stages required is the total operations in both the *if* and the *else* clauses.

For an actual branch in an outer loop, the branch condition is treated as a data-dependent enable signal for controllers under the branch clauses. If the controller is disabled, its *done* signal raises to high immediately. Output tokens depending on the *done* signal will be immediately sent out. This way, the controller under a branch condition does not need to execute if the outer branch evaluates to false.

Figure 3.11 shows an example with a branch statement. The input program in (a) writes and reads the memory *mem* on even and odd cycles of the outer loop *A*, respectively. SARA maps the three basic blocks in the program in three contexts, *ctxB*, *ctxD*, and *ctxF*, as shown in (b) and (c). For the read and write accesses, SARA allocates a context *respW* to accumulate write acknowledgments and a context *rqstR* to generate read requests. *ctxB* generates the branch conditions for both the *if* and the *else* clauses. The conditions are sent as data dependencies to contexts mapping the basic blocks under the branch conditions.

Figure 3.11 (d) shows the timing of execution. As *ctxB* has no dependencies, *ctxB* computes the conditions for all iterations of *A* in pipelined fashion and broadcasts the conditions to the receiver contexts. The *if* contexts (*ctxD*+*rspdW*) receives a `true` condition for the first iteration of *A* (A_0), executing all iterations of D, and passes a token to the *else* (*rqstR*+*ctxF*) contexts. Because *mem* is double-buffered, the credit is initialized with two elements in the receiver's input buffer, enabling the *if* contexts to execute two iterations of outer loop *A* before waiting for the *else* contexts. For the second iteration of loop *A* (A_1), the *if* contexts receives a `false` condition for branch *C*. Therefore, the enclosing loop controller *D*'s *done* signal is immediately high, sending the token to *rqstR* context right away. On the other side, the A_0 of the *else* contexts is blocked by the token from the *if* contexts. As soon as the token arrives, the *else* contexts send out the credit immediately, Next, the *else* contexts check for the second token, and executes loop *F* for A_1.

Both the *if* and the *else* contexts only execute if their enclosed branch clauses are evaluated to be true. More interestingly, Figure 3.11 (d) shows a overlapping execution of the *if* and *else* clauses across iterations of A. The hardware for both *if* and *else* clauses are active

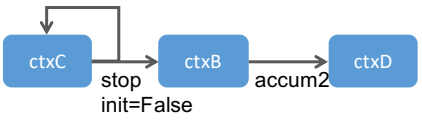
almost all time. If the latency of loop D and F are both L, the total runtime for N iterations of loop A would be on the order of $\frac{N}{2}L + L$ on Plasticine, assuming L and N are large. *This is almost twice as fast as a traditional coarse-grained pipelining with hierarchical FMS schedulers*, like Spatial's FPGA back-end, whose runtime is $NL + L$.

Do While Loops The *do while* construct is very useful to express iterative convergence algorithms or handle external data stream with a last-bit signal that terminates the execution. A *do while* loop works very similarly to a loop with dynamic range, except the loop has a very long initialization interval. The earliest starting time for the second iteration of the loop is when the while condition is resolved. The while condition is a data-dependency to all contexts mapping the basic blocks enclosed by the while loop. There is a loop-carried dependency between the producer of the condition within the while loop body and the loop controller that consumes a value of condition every iteration. Figure 3.12 shows an example with a do-while loop.

Loop *A* reads and block *C* writes a value of stop for every iteration of loop *A*. Figure 3.12 (b) shows the dataflow graph, mapping each basic block to a context. *ctxC* computes the stop condition, which is used by both *ctxC* and *ctxB*. There is a loop-carried dependency between the reader of *stop* from loop *A* and the writer of stop from block *C*. Therefore, the stop streams in (c) are initialized with `False` to enable the execution of the first iteration. The *accum* variable in (a) has two readers in block *B* and block *D*; one has a loop-carried dependency with the writer (line 9) and the other has a forward data-dependency (line 11). Therefore, the variable is mapped to two copies of *accum*, one for each reader. Because the accumulate operation is a single operation ADD, the accumulator gets optimized to the special accumulate pipeline register *accum1*, Without this optimization, *accum1* would be another `ScalarStream` with an initial element 0. The other *accum* becomes a scalar stream *accum2*, written by *ctxB* when loop *A* is done. (d) shows the timing of execution. As we can see, the initialization interval across iterations of A in the steady-state is bounded by the latency to evaluate *stop*, which is the number of operations in stop expression rounded up to a multiple of 6 stages because the data must propagate to the end of each SIMD pipeline.

```python
# This is an approximation of the
# do while loop in python syntax
accum = 0
# A loop running forver until stop
# is True
stop = False
A: for i in range(*) until stop:
    B: for j in range(N):
           accum += i
    C: stop = i > 100
D: ... = accum
```

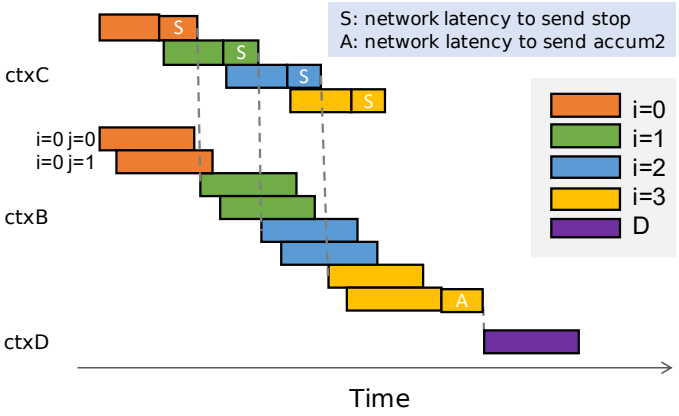
(a) Input program



(b) Dataflow graph

```python
# These streams are initialized with
# one initial element.
stop = ScalarStream(init=False)
accum2 = ScalarStream()
with Context() as ctxB:
    accum1 = AccumPR()
    A: for i in range(*):
        cond = stop.deq()
        if cond: break
        B: for j in range(N):
               accum1.accum(i)
    accum2.enq(accum1.value())
    accum1.reset()

with Context() as ctxC:
    A: for i in range(*):
        cond = stop.deq()
        if cond: break
        C: stop.enq(i > 100)

with Context() as ctxD:
    D: ... = accum2.deq()
```

(c) Context configuration



(d) Timing

Figure 3.12: An example program for do while loop. Loop *A* in (a) is a loop running forever, with a conditional *stop* variable. The break statements in (c) corresponds to configuring the stop signal of the enclosing controller *A* to the output of the stop input buffer. The stop buffer is dequeued every cycle controller *A* is active.

3.4.5 Virtual Unit Allocation (Ready)

After all contexts and shared memory are allocated and synchronized, SARA moves the remaining floating contexts into their own VUs, indicating the boundary of the physical units. Certain contexts mentioned in dense SRAM specialization might already belong to the same VU the memory locates at to ensure the synchronization has a single cycle latency. In the resource allocation phase, SARA might partition the big contexts into multiple VUs, and small contexts into a single VU. When contexts are merged into a single VU, they are still separate contexts triggered independently.

3.5 Resource Allocation (Ready)

The output of the transformation (Section 3.4) is a VUDFG that can execute on a Plasticine with infinite-sized physical units (PUs). The *Resource Allocation* phase enforces and addresses constraint violations given the specification of the Plasticine units. At the end of this phase, SARA assigns each VU in the VUDFG graph to a PU type with required resources; the placer then takes the type assignments and determines the final placement.

Accelerators often have heterogeneity in compute resources to improve efficiency for commonly used special operations. In Plasticine, PMUs and DAGs have specialized compute pipelines for address calculation that are less capable than the compute pipeline in PCUs. However, heterogeneity tends to reduce average utilization because different applications, and even the same application with different data sizes, can vary highly in the desired ratio among different resource [**?**]. A compute-bound application, for example, can heavily underutilize the DAGs and PMUs. To address this problem, SARA models the virtual to physical assignment as a constraint satisfaction problem; each VU consumes a set of resources and can only be assigned to a PU if the PU processes the required resources. Table 3.3 shows the types of resources SARA models in Plasticine's heterogeneous units. For example, special connection to off-chip memory interface is also treated as a type of resource in the DAG, which forces virtual contexts accessing DRAM to map to DAGs. On the other side, regular contexts with non-vectorized fixed-point operations can also be mapped to spare DAGs, which improves utilization.

As shown in Algorithm 1, the *resource allocation* phase contains three steps: *constraint resolution*, *global merging*, and *virtual to physical assignment*. SARA uses a VU-PU bipartite graph (G) to keep track of potential valid assignments between the two. Initially, G is initialized to a complete bipartite graph, i.e., all VUs can be assigned to all PUs. We refer to all PUs connected to a VU v as the domain of v in G, i.e. $dom(v)$.

Constraint Resolution A list of constraint pruners, each considering a set of on-chip resources, incrementally remove the VU-PU edges that violate the resource constraints. If a

```
Function alloc(V, P, pruners): /* Allocation Algorithm                    */
    Data: V: a set of VUs from the VUDFG
    Data: P: a set of all PUs on the hardware
    Data: pruners: a list of constraint pruners to check and fixes constraint
          violations
    /* Initialize a complete bipartite graph                              */
    G = new BipartiteGraph();
    G[V] = P;
    /* Constraint resolution                                              */
    prune(G, pruners);
    /* Global merging                                                     */
    merge(G);
    /* Heuristic check on whether assignment is feasible                  */
    check(G);
    /* Virtual to physical assignment                                     */
    backtracking_assign(G);

Function prune(G, pruners): /* A recursive pruning function               */
    Data: G: bipartite graph between VUs and PUs
    Data: pruners: a list of constraint pruners to check and fixes constraint
          violations
    Result: The function update G by removing VU-PU edges that violates
            constraints guarded by pruners. The function may fail and raise an
            exception.
    for pruner in pruners:
        for v in G.keys():
            for p in G[v]:
                if pruner.cost(v) > pruner.cost(p):
                    G[v] -= p;
            if G[v].empty():
                /* Partition VU v based on resource constraints
                   registered in pruner.  Not all resources can be
                   partitioned and this step may fail.  If
                   succeeded, the function returns a new set of
                   VUs.                                                   */
                V' = pruner.partition(v);
                G' = new BipartiteGraph();
                G'[V'] = G.values();
                prune(G',pruners);
                G -= v;
                G[V'] = G'[V'];
```

Algorithm 1: Resource allocation. The bipartite graph `G` contains a bi-directional many-to-many map. `G[key]` returns the set of values connecting to key (dom(key)), and `G[value]` returns the set of keys connecting to the value. `G[KeySet] = ValueSet` creates all-to-all connection between `KeySet` and `ValueSet`.

Feature	PCU	PMU	DAG	Host Unit
Vector lane width	16	16	1	1
Fixed-point op	✓	✓	✓	✗
Float-point op	✓	✗	✗	✗
Number of stages	6	10	5	0
Number of pipeline registers	8	8	4	0
Reduction tree	✓	✗	✗	✗
# Vector FIFO	6	6	4	0
# Scalar FIFO	6	6	4	16
# Control FIFO	16	16	4	16
Scratchpad banks	0	16	0	0
Scratchpad capacity	0	256kB	0	0
MergeBuffer	✓	✗	✗	✗
Splitter	✓	✗	✗	✗
Scanner	✓	✗	✗	✗
Access to DRAM Interface	✗	✗	✓	✗
Access to Host IO	✗	✗	✗	✓

Table 3.3: MergeBuffer, Splitter, and Scanner are new hardware introduced in [**?**] and [**?**] to support database and sparsity in Plasticine.

Function *check*(*G*): `/* Assignment feasibility check */`
  Data: G: bipartite graph
  Result: whether assignment is possible
  `/* For every value set in G */`
  for *V* **in** *G.values*().*toSet*()**:**
    K = ∅;
    for *v* **in** *V***:**
      for *k* **in** *G*[*v*]**:**
        if *G*[*k*] ⊂ *V***:**
          K += k;
    if |*K*| > |*V*|**:**
      return *failure*();
  return *success*();

Algorithm 2: Heuristic check on whether it is possible to assign all key with an value in a bipartite graph. Given there are only a few types of hardware tiles, $G.values().toSet()$ is relatively small. This algorithm roughly runs in $O(|G.keys()| \times |G.values()|)$, which is still much faster than the backtracking assignment, which has exponential runtime.

VU v has an empty domain after pruning, the pruner attempts to fix the violation by decomposing the VU into multiple VUs. Not all resources are composable, and the partitioning transformation may fail. If succeeded, the partitioner generates a new set of VUs V'. SARA starts a new complete bipartite graph between V' and all resources P, and recursively prune on V'. If succeeded, the original graph G is updated with V' and their pruned resources.

Global Merging After all VUs have at least one PU in the bipartite graph, SARA triggers a global optimization that merges small VUs into a larger VU to reduce fragmentation in allocation. Each type of resource has an aggregation rule to compute how the resource cost changes if two VUs are merged together. Most aggregation rules are simple, such as addition, logical or, max, or union. The in- and out-degree costs are tricker and will be detailed in Section 3.5.1.

Virtual to Physical Assignment Next, SARA performs a quick heuristic check on the bipartite graph to see if there exists a possible assignment for all VUs with sufficient PUs (Algorithm 2), and provide feedback on the limiting resources, otherwise. Finally, SARA assigns each VU to a PU type with a backtracking search on the pruned bipartite graph.

This approach can be easily extended to handle new heterogeneous tiles in the architecture by registering new types of resources with aggregation and partitioning rules. The rest of this section will focus on two types of partitioning transformations–compute partitioning in Section 3.5.1 and memory partitioning in Section 3.5.2.

3.5.1 Compute Partitioning

The *compute-partitioning* phase addresses VUs using more compute resources than any PU can provide. If a VU contains multiple contexts, SARA first moves the contexts into separate VUs. If a single context exceeds the resource limit, SARA breaks down the dataflow graph in the context into multiple contexts and puts them in separate VUs. During partitioning, SARA maps each subgraph of the large dataflow graph into a new context, mirrors

Problem	Partition the dataflow graph into subgraphs such that all subgraphs satisfy the constraints of a hardware unit.
Objective	Minimize the number of partitions and connectivity across partitions.
Constraints	Each partition must not exceeds the limit on the number of • live in/out variables (I/O ports) • operations (pipeline stages), • and live variables across operations (pipeline registers), etc. No *new* cycles can be formed across partitions other than the cycles in the original dataflow graph.

Table 3.4: Formulation of the compute partitioning problem

the control states of the original context, and streams live variables in between. We can formulate the problem of how to partition in the dataflow graph as an optimization problem, shown in Table 3.4. The partitioner "fixes" the VU v based on a single PU specification, albeit there are many potential PUs the decomposed VU can be mapped to. Currently, we use a heuristic to select a PU type from $dom(v)$ right before the compute pruning as a guiding constraint for partitioning.

Because the global network is specialized to handle efficient broadcasts, the in/out-degree of a partition counts the number of unique live-in/out variables, as supposed to the number of edges across partitions. In addition, the partitioned subgraphs cannot form *new* cycles; contexts waits for all input dependencies and therefore cycles across contexts cause deadlock. Nonetheless, the original graph might contain cycles representing loop-carried dependencies, such as accumulation. For these cycles, SARA initializes the back edge of the cycle with dummy data to enable execution. Figure 3.13 shows examples of valid and invalid partitioning solutions. Figure 3.14 shows another partitioning example of a dataflow graph with cycles.

Community Detection The formulation of compute partitioning is similar to the community detection problem[**?**], which has a similar objective. The major difference is that the latter often takes the number of output partitions as an input to the algorithm, whereas

our problem partitions until all subgraphs satisfy all constraints. Moreover, community detection algorithms do not enforce the cycle constraints. Finally, the edge connectivity in community detection counts the number of edges across partitions, as supposed to broadcast edges.

Retiming Imbalanced data paths across partitions can cause pipeline stalls at runtime. To ensure full-throughput pipelining, SARA needs to insert retiming buffers along the imbalanced data path across partitions. TODO: add an example. Retiming introduces new VUs in addition to the partitioned VUs, which attributes to the cost in Table 3.4's objective.

In the following sections, we present two algorithms to solve this problem: a traversal-based algorithm providing a decent solution with fast compile time, and a convex optimization-based algorithm with an optimum solution but long compile time.

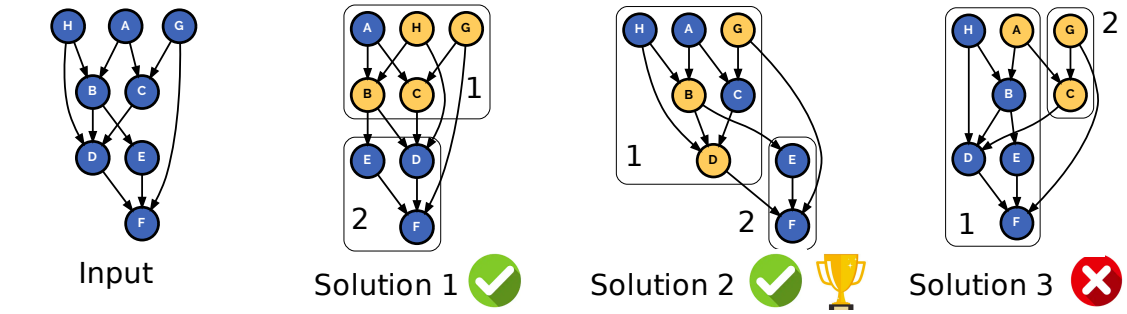
Traversal-based Solution

To address the cycle constraint, the traversal-based algorithm performs a topological sort of the dataflow graph. The topological traversal ignores the back edges in the graph. Staring from the beginning of the sorted list, the algorithm iteratively adds nodes into a partition until it fits no more nodes. The algorithm then 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 in the dataflow graph.

The partitioning result is a function of the traversal order. We experienced with depth-first search (DFS) and breadth-first search (BFS) with forward and backward dataflow traversal orders. For DFS, we re-sort the remaining list each time we start with a new partition.

Solver-based Solution

The convex optimization solution models the problem as a node-to-partition assignment problem. Table 3.6 gives our formulation. At a high-level, we use a boolean matrix B to keep track of the assignment. B has dimension equals to the number of nodes in the

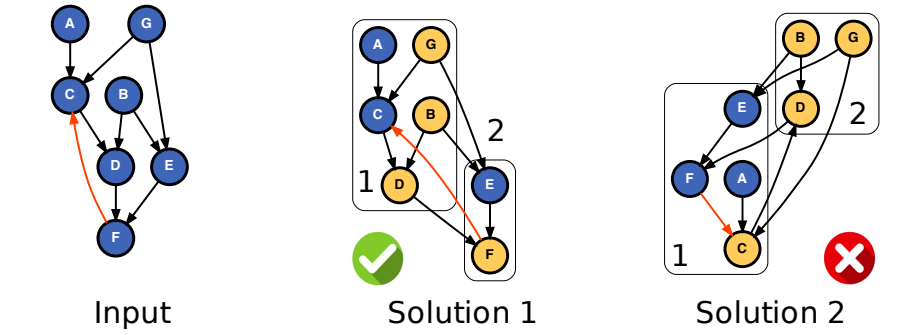


Partition	1	2
# op	5	3
# input	0	4
# output	4	0

Partition	1	2
# op	6	2
# input	0	3
# output	3	0

Partition	1	2
# op	6	2
# input	2	1
# output	1	2

Figure 3.13: Compute partitioning examples. Solution 1 and 2 are both valid partitioning. Solution 2 is better because it has less number of broadcast edges (3 as supposed to 4 in Solution 1) across partitions. Solution 3 is an illegal partition result due to the cycle between partition 1 and 2.



Partition	1	2
# op	5	2
# input	1	3
# output	3	1

Partition	1	2
# op	4	3
# input	3	1
# output	1	3

Figure 3.14: Compute partitioning examples with cycle in the dataflow graph. Solution 1 is valid because there is no cycle between partitions after removing the back edge in the original graph. Solution 2 is invalid because there is still cycle between partition 1 and 2 after removing the back edge.

dataflow graph by the number of partitions, where$B[i, j] = 1$ indicates node i is assigned to partition j. In Table 3.6, *partition assignment* restricts each node to have a single partition assignment. The *input and output arity constraints* show the formulations that limit the number of input and output for a subgraph. 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_n with a size equivalent to the number of nodes. The delay vector encodes a schedule to execute each node, and the values are selected by the solver. The *dependency constraint* enforces scheduling a node no earlier than its input dependencies, and no later than its output dependents. Since the operations within a partition have to be triggered atomically, there is another delay vector d_p for the partitions. The *delay consistency* enforces the schedule of a node equals to the schedule of its assigned partition. Finally, *constant validity* limits the range of values the delay vectors can be chosen from. In addition to enforcing the cycle constraint, these delay variables are also used to calculate where retiming is required and project the amount of introduced retiming VUs. Finally, to reduce the solving time, we use the traversal-based solution to warm start the assignment matrix B.

Comparison

TODO: Benchmark Table Figure 3.15 shows the comparison between the traversal-based and the solver-based solutions for both compute partitioning and global merging. Global merging is a global optimization merging small VUs into a large VU that can still fit in a PU. The merging algorithm is very similar to the compute partitioning algorithm, where nodes in the dataflow graph corresponds to the VUs in a VUDFG. The merging problem also has a traversal-based and solver-based solution. Section 3.6 will discuss merging in more detail.

We used a commercial solver, Gurobi [7], for the solver-based solutions. The evaluation is performed on an Intel Xeon E7-8890 CPU at 2.5GHz with 1TB DDR4 RAM. Gurobi is parallelized across ten threads for each application. To speed up convergence, we configured Gurobi with a 15% optimality gap, i.e., the solver is allowed to early stop after the solution

Name	Type	Description	Definition / Default
$\mathcal{N}$	Constant	Enumeration of nodes to partition, numbered $\{n_i\}_i$	-
N	Constant, $\mathbb{Z}_{\geq 0}$	Number of operations to partition	$N = \lvert\mathcal{N}\rvert$
P	Constant, $\mathbb{Z}_{\geq 0}$	Number of partitions to consider	N, or from heuristic
$\mathcal{E}$	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	Supplemental Materials
$\text{and}(\cdot,\cdot)$	$\{0,1\} \times \{0,1\} \to \{0,1\}$	Boolean and of binary variables	Supplemental Materials
d_p	Variable, $\mathbb{Z}_{\geq 0}^P$	Vector of partition delays	-
d_n	Variable, $\mathbb{Z}_{\geq 0}^N$	Vector of node delays	-
$\text{dest}(n)$	$\mathcal{N} \to \mathcal{P}(\mathcal{N})$	The set of nodes which depend on n	$\{n' \lvert n' \in \mathcal{N}\ s.t.\ (n \to n') \in \mathcal{E}\}$
c_o	Constant, $\mathbb{Z}_{\geq 0}$	Maximum output arity of a partition	HW Spec
c_i	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input arity of a partition	HW Spec
b_d	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input buffer depth	HW Spec
K	Constant, $\mathbb{R}_+$	Very Large Constant, used for constraint activation	$P \times N$
α_d	Hyperparameter, $\mathbb{R}_+$	Retime merging probability multiplier	$\frac{1}{\max\{c_o,c_i\}}$
$\mathcal{C}_r$	$[\mathcal{N} \to \mathbb{R}_+, \mathbb{R}_+, [\mathbb{R}_+] \to \mathbb{R}_+]$	List of per-node values, limits, and reduction functions for reducible constraints	Supplemental Materials
F	$\{0,1\}^{N\times P}$	Feasibility matrix, whether a partition can support a node	HW Spec

Table 3.5: Names and definitions used in the solver-based algorithms.

Type	Description	Expression
Cost Function	Allocated Partitions	$\Sigma_i \text{proj}_{\mathbf{B}}(\Sigma_j B_{i,j})$
	Retiming Partitions	$\alpha_d \Sigma_{n_i \to n_j \in \mathcal{E}} \text{proj}_{\mathbf{B}}(\max\{d_n(j) - d_n(i) - b_d, 0\})$
Partition Constraint	Partition Assignment	$\forall n_i \in \mathcal{N} : \ \Sigma_j B_{i,j} = 1$
	Input Arity Constraint (vectorized)	$\Sigma_{n_i \in \mathcal{N}} \max\{\text{proj}_{\mathbf{B}}(\Sigma_{n_j \in \text{dest}(n_i)} B_{j,:}) - B_{i,:}, 0\} \leq c_i \times \vec{1}$
	Output Arity Constraint	$\forall p \in [0, P) :$ $\Sigma_{n_s \in \mathcal{N}} \text{and}(B_{s,p}, \text{proj}_{\mathbf{B}}(\max\{(\Sigma_{n_d \in \text{dest}(n_s)} B_{d,p}) - K \times B_{s,p}, 0\})) \leq c_o$
	Dependency Constraint	$\forall n_i \to n_j \in \mathcal{E} : \ d_n(i) + 1[p_i \neq p_j] \leq d_n(j)$
	Delay Consistency	$\forall n_i \in \mathcal{N} : \ d_n(i) \leq \min_j(d_p(j) + K - B_{i,j} \times K)$ $\forall n_i \in \mathcal{N} : \ d_n(i) \geq \max_j(d_p(j) + B_{i,j} \times K - K)$
	Constant Validity	$\forall n_i \in \mathcal{N} : \ d_n(i) \leq K$ $\forall i \in [0, P) : \ d_p(i) \leq K$
Merge 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 3.6: Solver formulation for partitioning. Expressions are presented using the Disciplined Convex Programming ruleset [1, 2]. Explanations for selected expressions can be found in the supplemental material.

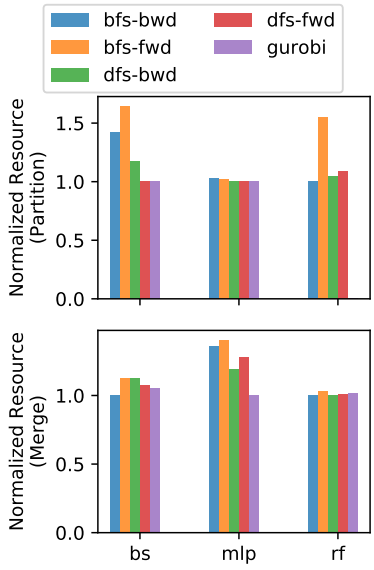
is more than 85% close to the optimum. The solving time increases dramatically as getting close to 100% optimum.

Figure 3.15 (a) shows the normalized resource in a number of VUs after partitioning and merging. We can see that Gurobi provides almost the best solution for all applications when it can derive an answer in a reasonable amount of time. The missing solver bar in random forest (*rf*) partitioning is due to timeout after a few days. The traversal-based algorithms can sometimes match or even outperform the solver slightly. However, because the partitioning result is a function of the traversal order, each traversal order has adversarial cases, where they can be up to 1.7x worse in resource than the best possible solution. The forward (*fwd*) traversal order schedules nodes as earlier as possible, which reduces the number of external live variables; the backward traversal minimizes the number of internal live variables across partitions. The depth-first-search (DFS) traversal order minimizes the number of live variables between partitions, albeit producing more imbalanced paths between partitions. On the other hand, breath-first-search (BFS) produces more balanced partitioning with more live variables and partitions.

There are two common graph patterns in applications that require partitioning. The first is a dataflow graph from a large basic block, which contains a small set of external live-in and live-out variables and abundant intermediate temporary variables. Such graphs typically end up with long-live variables across partitions that require retiming. TODO: show example and discuss the other pattern.

Figure 3.15 (b) and (c) shows the compile time for these algorithms. The single-threaded the traversal-based algorithm runs in minutes, which is significantly faster than the parallelized solver that takes hours to days. In general, the solver runtime becomes quickly unbounded with a large amount of VUs. TODO: show solver time with increasing number of VBs.

In summary, the solver solution provides a guaranteed close-to-optimum solution at an expensive compile time. Nonetheless, the solver-solution treats the retiming and partitioning as a joint optimization, whereas the traversal-based solution solves these two problems in two separate passes, which is less optimum. Moreover, the solver-based solution tends



(a) Resource Comparison

Apps	bfs-bwd	bfs-fwd	dfs-bwd	dfs-fwd	gurobi
bs	3s	3s	2s	2s	2h4m
mlp	<1s	<1s	<1s	<1s	4s
rf	1m37s	1m7s	29s	27s	-

(b) Compile time for splting

Apps	bfs-bwd	bfs-fwd	dfs-bwd	dfs-fwd	gurobi
bs	1s	2s	5s	3s	1m4s
mlp	5s	5s	6s	11s	31m16s
rf	11s	10s	2m7s	1m0s	14h27m

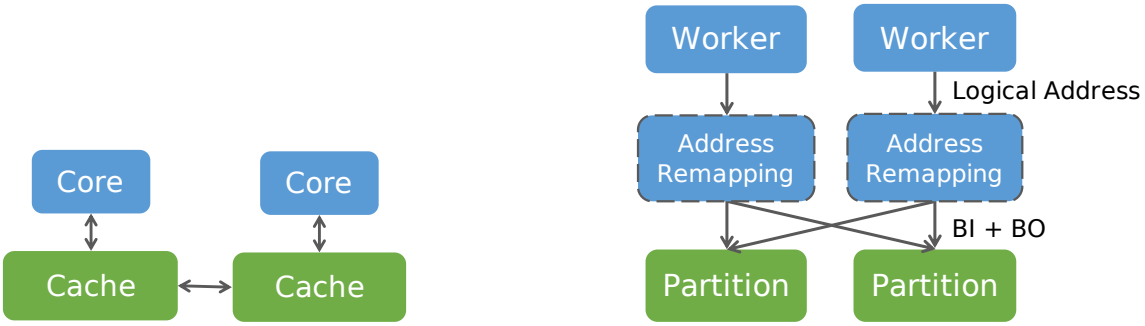
(c) Compile time for merging

Figure 3.15: Partitioning and merging algorithm comparisons. (a) shows the normalized resource usage between different algorithms (the lower the better). (b) and (c) shows the compile time of each algorithm.

to produce a better result for architectures with tight I/O bound and relax stage constraint. The traversal-based solutions, on the other hand, can produce a decent solution in a short amount of time. However, the solution is prune to adversarial cases and potentially perform badly for unseen graph structures. In practice, we can combine the two approaches and invoke the expensive solver only when the traversal-based solution is insufficient. The quality of the traversal-based solution can be easily estimated by the resource utilization in each partition.

3.5.2 Memory Partitioning

The memory pruner addresses virtual on-chip memory exceeding the capacity and bandwidth limit of a single PMU. As we parallelize the computation, the on-chip memory must



(a) Processor Architecture (b) Reconfigurable Spatial Architecture

Figure 3.16: Memory model of different architectures

provide higher address bandwidth to sustain the compute throughput. On a processor-based architecture, this is often achieved by associating a separate first-level cache for each processor core, as shown in Figure 3.16 (a). The cache implements a hardware coherence protocol that synchronizes the different copies of data behind the scenes, providing the abstraction of a shared memory.

The coherence protocol is both expensive in hardware complexity and hard to scale in bandwidth for streaming pipelined execution. Instead of making redundant copies of the data, we can partition the data across different memory partitions to get additional address port on a reconfigurable accelerator, as shown in Figure 3.16 (b). Each parallel worker broadcasts the requests to all memory partitions. If the data accessed by two parallel workers live in the same bank, the bandwidth will be halved. To avoid bank conflicts, an important static analysis often used on reconfigurable accelerators is called static partitioning, or static banking [8]. For most address patterns, the compiler can derive a partitioning scheme such that every partition is accessed by a single worker at any time, which guarantees bandwidth at runtime. The output of the analysis is an expression for bank ID (BI) and bank offset (BO), both are functions of the requested logical memory address. BI determines which partition each request is going to, and BO is the offset within the partition.

Figure 3.17 shows how static banking is achieved on the Plasticine architecture. Banking analysis from Spatial specifies the number of banks required to sustain bandwidth for the current parallelization factors, and expressions for BIs and BOs. SARA groups the banks

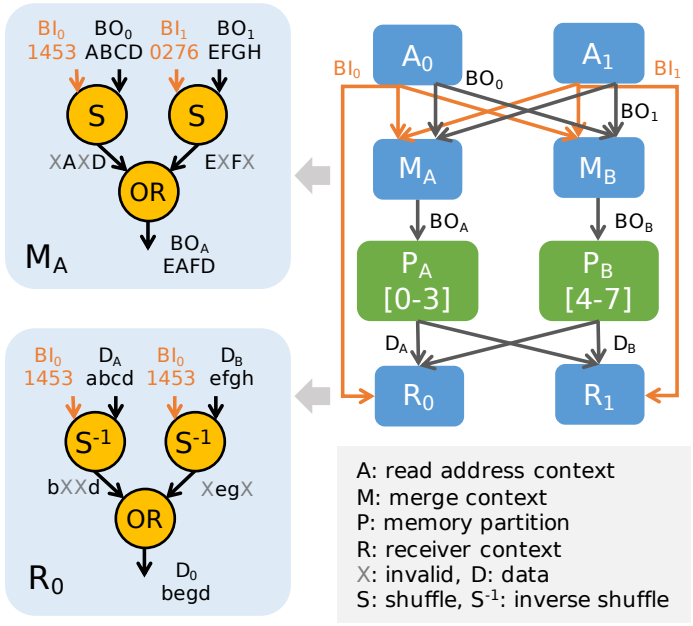


Figure 3.17: Example of memory partitioning across physical units. In this example, the program parallelizes the outer loop by two and vectorizes the inner loop by 4, which generates two vectorized access lanes. We need eight scratchpad banks to sustain the read bandwidth. SARA groups the 8 banks in two virtual memory partitions, P_A (bank 0-3) and P_B (bank 4-7). The address generation contexts A_0 and A_1 contains the computation for logical address and address remapping, which output BIs and BOs. SARA allocates one merge context per memory partition (M_A and M_B), merging requests from all access lanes. Inside the merge context, the shuffle operator converts the BO from access-aligned to bank-aligned. A_0, for example, requests offsets A, B, C, and D (BO_0) from banks 1,4,5,and 3 (BI_0). The shuffle operator picks the requests belonging to partition A and outputs a BO aligned with bank 0-3. If the bank i in the partition has no request, the ith element in the output is marked as invalid. The merge context contains one shuffle operator per request lane, and uses a tree of OR operators to combine all bank-aligned BOs into the final BO_A. On the receiver side, the memory broadcasts its response to all receiver lanes. The receiver context uses an inverse shuffle operator to convert the response from bank-aligned back to access-aligned, using the BI forwarded from the address context. The response is then merged with another OR tree.

into virtual memories, with group size limited by the number of banks in a PMU. For each access lane, SARA allocates a context to compute the logical and remapped address, which outputs a vectorized BI and BO for each access lane. BIs and BOs are broadcasted to all memory partitions. For each memory partition, SARA allocates a merge context, merging BOs from all requesting lanes. The merge context uses a shuffle operator to transform the vectorized BO from bank order specified by BI (access-aligned) to bank order assigned to the current partition (bank-aligned). Because the static banking analysis guarantees no bank conflicts for all banks, SARA can use a OR tree to merge the bank-aligned BOs into the final BO that gets send to banks in the partition. On the receiver side, SARA uses an inverse shuffle to convert responses from partitions from bank-aligned back to access-aligned. SARA uses another OR tree to merge the access-aligned responses, which produces the final vector data requested by the access lane. With large parallelization, both the merge OR tree and the respond OR tree can be partitioned across VUs if running out of stages in a VU, and to scale in the network bandwidth.

TODO: Talk about what if banking fails.

3.5.3 Register Allocation

```
mem = array(dims=[16])
A: for i in range(16,vec=16):
       data1 = ...
       mem(i) = data1 # W1
B: for j in range(M):
    C: for i in range(16,vec=16):
           data2 = mem(15-i) # R1
           ...
```

(a) Input program

```
mem = VectorStream()
with Context() as ctxA:
    # Loop A is optimized away
    data1 = ...
    mem.enq(data1)
with Context() as ctxC:
    ## vector constants
    fromAddr = xrange(0,16)
    toAddr = xrange(start=15,end=-1,step=-1)
    B: for j in range(M):
           # Loop C is optimized away
           tmp = mem.deq()
           data2 = shuffle(from=fromAddr,
                   to=toAddr,base=tmp)
           ...
```

(b) Context graph

Figure 3.18: (a) shows an example where read and write address of `mem` can be constant folded because loop A and loop C are fully parallelized. Instead of mapping `mem` into a scratchpad, which heavily underutilize the capacity of the scratchpad, SARA maps the memory into a non-indexable vector stream in (b). The vector stream stores address [0-15] of the original memory, whereas the reader $R1$ requests address [15-0] instead. SARA uses a `shuffle` operator to swap the vector elements based on reader's expected order. This is the same `shuffle` operator we introduced in the memory partitioning section Section 3.5.2.

3.6 Optimizations (Ready)

SARA performs many of the standard compiler optimizations, such as Dead Code Elimination and Constant Propagation. Some of them, however, plays a much more important rule for reconfigurable accelerator because they have a direct impact on resource usage. Other optimizations can be counter-intuitive, as they introduce redundant computation that reduces resources without necessarily impacting performance. In this discussion, we focus our primary objective on performance. Saving resource is an indirect objective as resource reduction enable larger parallelization factors, which in turn improves performance. Other objectives, such as power saving, is left as future work.

3.6.1 Description (Ready)

```
1  # Input program
2  accum = 0 # M1
3  out = 0 # M2
4  for i in range(N):
5      for j in range(M):
6          accum += i * j # W1
7      out = accum # R1 and W2
8  ...
9  ... = out # R2
10
11 # Optimized program
12 out = 0 # M2
13 for i in range(N):
14     for j in range(M):
15         out += i * j # W1
16 ...
17 ... = out # R2
```

(a) Example with variable

```
1  # Input program
2  tmp = queue() # M1
3  out = queue() # M2
4  for i in range(N):
5      for j in range(M):
6          tmp.enq(i * j) # W1
7  for i in range(N*M):
8      out.enq(tmp.deq()) # R1 and W2
9  ...
10 ... = out.deq() # R2
11
12 # Optimized program
13 out = queue() # M2
14 for i in range(N):
15     for j in range(M):
16         out.enq(i * j) # W1
17 ...
18 ... = out.deq() # R2
```

(b) Example with queue

Figure 3.19: (a) and (b) shows route through opportunities in the program. These opportunities are often outcome of lowering and other optimizations. To proof the route through is safe, SARA needs to have access to all readers and writers of a memory, and analyze the control structure around the memory accesses. In addition to saving on the memory, the optimization eliminates the basic block in line 7 of (a) and line 8 of (b), which saves a context.

Memory strength reduction (msr). Like traditional strength reduction on arithmetics, SARA replaces expensive on-chip memories with cheaper memory types whenever possible. Scratchpad, input buffer, and pipeline register are different types of on-chip memory on Plasticine from the most expensive to the cheapest. For example, SARA maps on-chip array to a scratchpad by default. However, if all of the readers and writers of the array index into memory with constant addresses, often as a result of constant propagation of full parallelized loops, SARA maps the memory to the input buffer of a context, and setup datapath to connect the reader and writer directly. Figure 3.18 shows an example of strength reduce a scratchpad into an input buffer with additional operations. Fully parallelized loops are a common outcome of an automatic design space exploration of the parallelization factors. We also use it as a way for the user to explicitly program vectorized streaming operation for Plasticine.

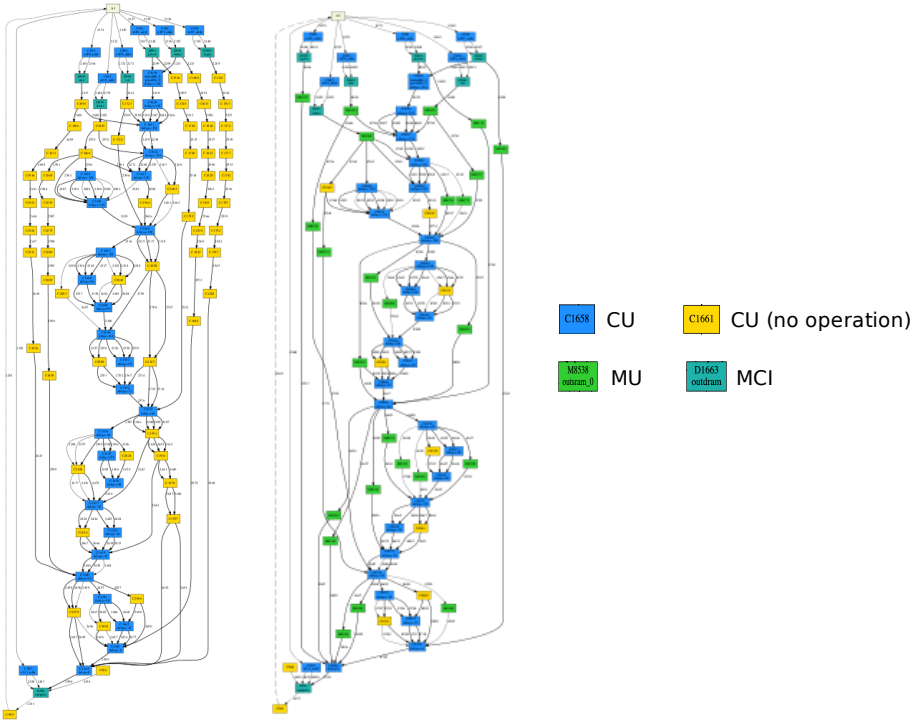
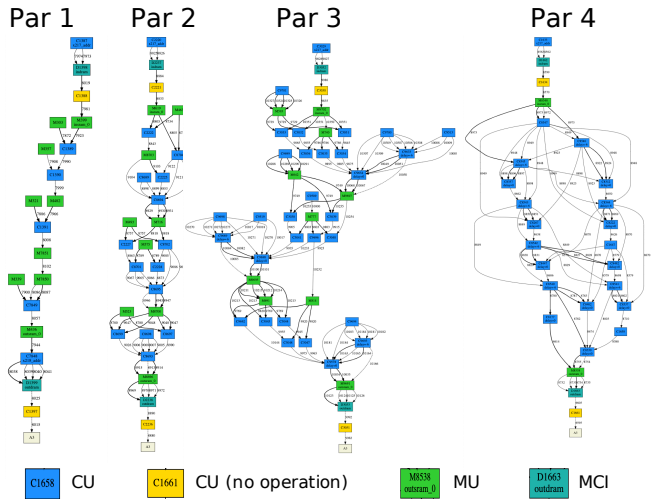


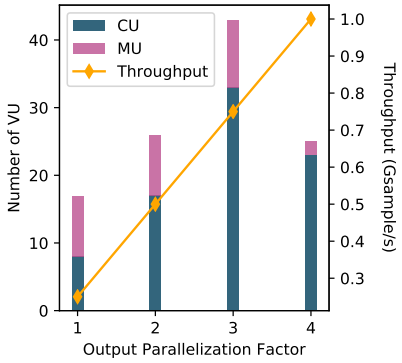
Figure 3.20: Virtual unit dataflow graph (VUDFG) after retiming. CU: virtual compute unit. MU: virtual memory unit. MCI: memory controller interface. Left: retiming with input buffer only. Right: retiming with either input buffer or scratchpad. The yellow CUs are CUs used for retiming. The original program does not have MU and the yellow MU on the right are MUs used for retiming.

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 and feeds the output of M1's writer (W1) directly to M2's reader (R2) if it can prove the propagation is safe. Figure 3.19 shows examples of route-through opportunities. These patterns are often the outcome of multiple levels of compiler lowering and the *msr* optimization. Eliminating route-through patterns can simplify control hierarchy and reduce the number of basic blocks, which eliminates contexts.

Retiming with scratchpad (retime-mem). By default, SARA uses input buffers in PU for retiming purposes. This option enables SARA to use scratchpad memory to retime paths requiring deep buffers. Figure 3.20 shows VUDFGs after retiming with and without



(a) VUDFGs with different parallelization factors



(b) Throughput and resource as a function of parallelization factor

Figure 3.21: A 4x4x4 MLP case study. Light-weight models like this can be useful in ultra high-bandwidth and low-latency inference in packet processing [**?**]. (a) shows changes in VUDFG as we parallelize the output dimension. The input dimension of MLP is vectorized across SIMD lanes. As reaching par factor at 4, the intermediate memory in MU gets optimized away due to constant folding and rtelm on fully unrolled loops. (b) shows the throughput and resource increase as the output dimension is parallelized.

scratchpad retiming. As we can see, allowing scratchpad and input buffer for retiming significantly reduces the total number of retiming units.

Crossbar datapath elimination (xbar-elm). Although in the general cases described in Section 3.5.2, crossbar data paths between accessors and memory partitions are very expensive, the BI can often be statically resolved to a constant vector for certain parallelization factors and address patterns. When BI is a constant, SARA can use this information to intelligently group banks into memory partitions that reduce the crossbar to a partial or a point-to-point connection. This is an important optimization that prevents resource scaling quadratically with increasing parallelization in the common cases.

Combined with *msr* and *rtelm* mentioned above, SARA is able to dramatically reduce the resource usage for certain parallelization factors. Figure 3.21 shows an interesting case study on an MLP presenting performance scaling and resource increase while scaling up

parallelism. Figure 3.21 (b) demonstrates that Plasticine can have resource non-linearly increasing with parallelization factors, while having perfect performance scaling, due to compiler optimizations.

Read address duplication (dupra). Between the address context and receiver contexts in memory partitioning shown in Section 3.5.2 Figure 3.17, the imbalance datapaths between BI forwarding and request datapath can cause significant pipeline stalls, limiting the overall application performance to a small fraction of the ideal throughput. When parallelizing the memory access, the OR tree in the merge context gets partitioned into a tree of contexts, which further increases the mismatch in latency between the imbalanced datapaths. Instead of retiming BI forwarding path, this compiler flag forces SARA to duplicate the BI computation using local states within the receiver context, which could use fewer resources than retiming.

Global Merging (merge). After all VUs satisfy the hardware constraint, we perform a global optimization to compact small VUs into larger VUs. Merging has a very similar problem statement as compute partitioning (Section 3.5.1) except with more constraints. For merging, the VUDFG is the graph, the VUs are the nodes, and the merged VUs are partitions. Unlike the partitioning problem, which uses a single set of cost metrics to determine if a partition is valid, the merged partition can be mapped to any PU available on-chip, each having a different set of cost metrics and a quantity limit. When adding a new VU m to a partition P in the traversal-based algorithm, we first take the union of the domains of VUs within the current partition, and intersect with the domain of m. The domain of the merged partition M is updated to all PUs within this intersection that M can fit.

$$M = P \cup \{m\} \tag{3.1}$$

$$dom(M) = \{p | p \in \{\cup\, dom(n) \mid \forall n \in P\} \cap dom(m), cost(M) \leq cost(p)\} \tag{3.2}$$

```
mem1 = rand(N)
mem2 = rand(N)
# Block 1
c = a + b
for i in range(N):
    # Block 2
    mem2[i] += mem[i] * c
```

(a) Input program

```
mem1 = rand(N)
mem2 = rand(N)
for i in range(N):
    # Block 2
    c = a + b
    mem2[i] += mem[i] * c
```

(b) Reversed loop invariant hoisting

Figure 3.22: The original program requires at least two contexts to execute Block 1 and Block 2. By moving the invariant instruction `c = a + b` into the loop body, (b) only needs a single context instead. Because instructions within `Block 2` are pipelined across loop iterations, adding instructions in the loop body introduce minimum performance impact. This transformation is beneficial until `Block 2` exceeds six operations, at which point both version consume the same amount of resources.

The caveat is that even if M has non-empty domain, the bipartite graph might not have a possible assignment after merging, as M might fit in a larger PU with insufficient quantity. Therefore, we perform the feasibility checking of the bipartite assignment at each step of merging, shown in Algorithm 2, and only merge if the assignment is feasible.

The solver-based solution combines the VU merging with VU to PU assignment as a joint problem. The output of merging gives both partition assignment as well as a PU type assignment, which eliminates the risk of infeasible assignment due to merging in the traversal-based solution.

Reversed Loop Invariant Hoisting A common loop optimization on traditional CPU is to move loop-invariant expressions outside of the loop body to reduce computation. This transformation, however, can introduce more basic blocks in the program, that translates to more contexts on Plasticine. Therefore, the reversed loop invariant hoisting, which moves loop-invariant expressions into the loop body, can sometimes save resources by simplifying the control structure, as shown in Figure 3.22. Because instructions within the basic blocks are pipelined, moving instructions into the loop body increases computation without hurting throughput, which dominant the performance for spatial architecture. Currently, we rely on the user to perform this optimization manually.

Figure 3.23: Optimization effectiveness. Percentage resource reduction or performance improvement from each optimization. The heat map shows the maximum differences when turning on/off the optimization, while fixing other optimizations at the most optimum combination for the application. Each application has a large design space with various parallelization factors in different loops, we report a design point that has the maximum impact, either positively or negatively.

3.6.2 Evaluation (WIP)

Figure 3.23 presents the evaluation of performance improvement and resource reduction from optimizations discussed in Section 3.6.1.

merge and *rtelm* are optimizations that gives a consistent resource improvement for most applications. *dupra* and *msr* helps **mlp** and **lstm**, which are heavily partitioned during the memory-partitioning stage.

dupra significantly reduces unbalancing in the data path caused due to the large merge tree generated by memory partitioning. *retime* can have a significant impact on the performance of the application at the cost of an increase in resource usage. Using scratchpad as a retiming buffer with the *retime-mem* flag can significantly reduce resource usage.

3.7 Placement and Routing (Ready)

The input to the *placement and routing* (*PaR*) phase is a VUDFG, with each VU tagged to a type of PU. Before PaR, SARA also performs a runtime analysis of the program, annotating each edge in VUDFG with a link priority. The link priority is used to determine the routing order during PaR. PaR then iteratively places VUs and route edges in VUDFG onto the global network.

In addition to the original static network in Plasticine, we introduce a dynamic network in parallel to the static network, forming a dynamic-static hybrid network. Both purely static and purely dynamic networks are instances of the parameterized hybrid network. Section 4.3.2 will discuss the details about the hybrid network. The PaR algorithm needs to handle both network types and multiple network granularity (vector, scalar, control) at the same time.

The major difference between the static and the dynamic network is that each physical link in the static network is dedicated to a logical edge in the VUDFG for the entire duration of the execution (circuit-switching[2]), whereas the physical link in the dynamic network can be time-shared by multiple logical edges (packet-switching). Both static and dynamic networks are statically routed by the PaR algorithm. For a dynamic network, PaR also needs to assign virtual channels (VCs) to prevent network deadlock. The PaR algorithm configures the lookup table in each router that maps the packet header to the destination port and VC.

In the rest of this section, Section 3.7.1 discusses the placement and Section 3.7.2 reviews the routing algorithm. Section 3.7.3 explains the need for VC allocation. Lastly, Section 3.7.4 describes how SARA generates link priority.

[2]Technically, our static network is more restrictive than circuit-switching because circuit-switching allows deallocation after the connection is terminated. Our static network, on the other hand, cannot be reconfigured until the application terminates.

3.7.1 Iterative Placement

At high-level, the PaR algorithm works very similarly to the FPGA PaR algorithm using simulated annealing [**?**]. For the initial placement, the algorithm places VUs in VUDFG in topological order. Each VU is placed to the next available PU with minimum Manhattan distance to the placed neighbors of that VU. Next, the algorithm routes all edges in the VUDFG, starting from edges with the highest link priority. If no routes are available on the static network, the route will be moved onto the dynamic network. For purely static networks, there is an "imaginary" dynamic network for this step. At the end of the PaR, if there are still routes on the fake dynamic network, PaR is considered failed for the static network. After all routes are routed, either on the static or dynamic network, the PaR evaluates the congestion cost of the current placement. Then, a genetic algorithm shuffles the VUs whose edges contribute most to congestion, and keeps the new position if it improves the route assignment. By iteratively re-placing and re-routing, the mapping process eventually converges to a good placement.

The PaR uses a heuristic cost model 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. SARA provides a static estimate of the number of packets sent on each logical link. The PaR algorithm estimates 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.

3.7.2 Congestion-Aware Routing

To achieve optimal performance, we use a routing algorithm that projects congestion and routes around it. Routing starts with the highest-priority routes, as determined by fanout of the broadcast edge and estimated packet count; broadcast edge with higher fanout is

harder to route and hence are routed first. Using the packet count as a priority 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 [**?**] and a hop weighting function. To ensure maximum link reuse in broadcast edges, we can augment the hop weight in Dijkstra's algorithm by a multiplication factor between zero and one. When finding the shortest path between each source-destination pair, the weight of the hop is multiplied by the factor each time the hop is reused for the same broadcast edge. In other words, the reused path has a lower hop cost compared to other paths. This trick provides a balance between the shortest path and link reuse: a smaller factor encourages link reuse, even though individual source-destination pairs are not the shortest path; a factor equals to 1 ensure all source-destination pairs are routed with the shortest path. The other objective for routing broadcast links is balancing the hop counts between all destinations. This is because the shorter path will back pressure the sender before packets reaching to the longer path, causing pipeline stalls. To achieve this, we start with the source-destination pair with the longest Manhattan distance, ensuring the most far apart source-destination pair is routed on the shortest path. The consecutive source-destination pairs will try sharing the link on this path and slightly detour from their shortest path, which balances the hop count.

3.7.3 VC Allocation for Deadlock Avoidance

Deadlock is a system pathology in dynamic routing where multiple flits form a cyclic holds on/waits for dependency on each others' buffers and prevent forward progress. Most dataflow 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 back-pressure 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 [**?**].

Figure 3.24(a) shows a sample VUDFG graph, which is statically placed and routed on a 2×3 network in (b). Logical edges B and C share a physical link in the network. If C fills the buffer shared with B, VU-3 will never receive any packets from B and will not make

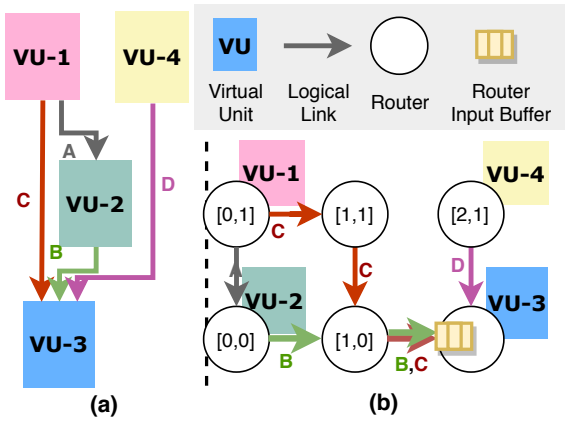


Figure 3.24: An example of deadlock in a streaming accelerator, showing the (a) VU dataflow graph and (b) physical placement and routes on a 2×3 network. There are input buffers at all router inputs, but only the buffer of interest is shown.

forward progress. With streaming computation, the program 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 VU 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.

3.7.4 Runtime Analysis for 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 each basic block will execute. The execution count on the basic block can future used to derive the packet count produced by these basic blocks. If the control is data-dependent, the user can annotate the data-dependent loop range, or the fraction for how frequently the branches evaluating to true. These heuristic guides can help the placer to evenly spread out the 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.

Using the estimated packet 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: packets generated by the innermost loops are the more likely to be frequent. This heuristic provides a reasonable estimate of links' priorities for routing purposes.

TODO: Elaborate on how to perform the analysis especially for streaming program

3.8 Debugging and Instrumentation Support (WIP)

Deadlock in Streaming Reconfigurable Architecture

Debugging Support and Performance Instrumentation

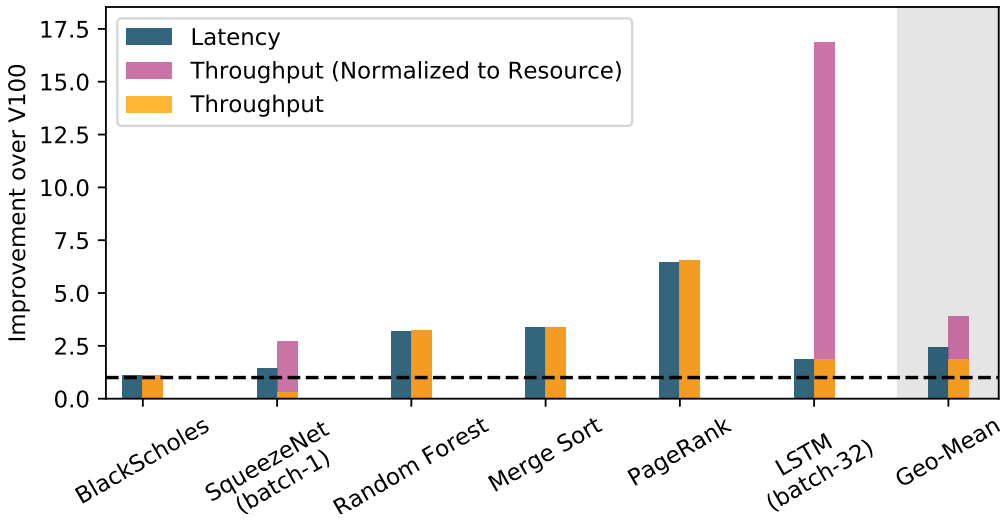


Figure 3.25: Plasticine's latency and throughout improvement over V100 GPU. The evaluated Plasticine architecture has area footprint of 352mm^2 at 28nm. V100 GPU has area footprint of 815mm^2 at 12nm. Both platforms have the same off-chip bandwidth at 1TB/s with HBM technology. Yellow and blue bars show 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.

3.9 Evaluation (WIP)

Chapter 4

Architecture

In this section, we discuss the architectural advancement on top of the original Plasticine architecture introduced in [4]. 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 4.1 lay outs the datapath changes in order to support more flexible banking schemes required by general access patterns supported in Spatial; Section 4.2 discusses the hardware specialization and architectural sizing for machine learning applications; **??** provides an extensive study on on-chip network selection reconfigurable spatial architectures.

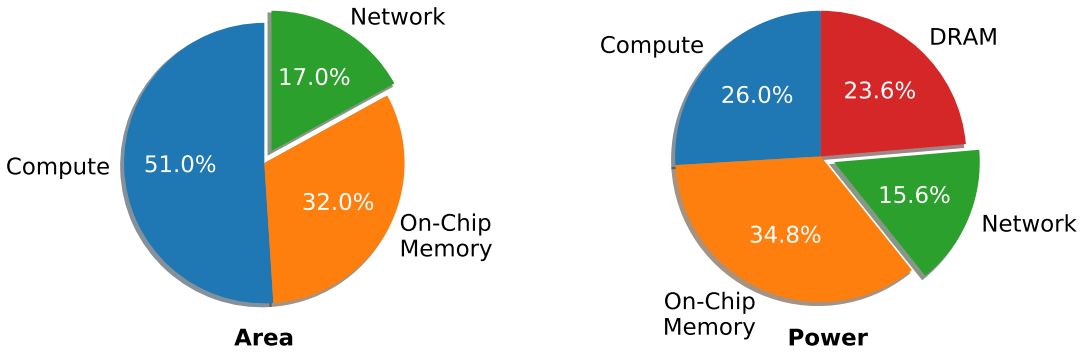


Figure 4.1: Area and power breakdown of Plasticine

4.1 Generic Banknig Support (WIP)

4.2 Plasticine Specialization for RNN Serving (Ready)

Low-precision inference are commonly used to reduce the memory footprint of deep learning models and increase the compute density of hardware accelerators. Plasticine, however, only supports 32-bit operations and datapath. Using RNN serving as a motivating example, this section discusses the necessary architecture augmentation and specialization needed to efficiently map real-time inference on Plasticine.

Recurrent Neural Networks (RNNs) are a class of sequence models that play a key role in low-latency, AI-powered services in datacenters [9, 10]. These applications have stringent tail latency requirements, within the window of milliseconds, for real-time human-computer interactions. An example of such workloads is Google Translate; inference runs concurrently as the user types. Efficient acceleration of RNN requires flexible datapath to support global optimizations beyond BLAS kernels, which is where dataflow accelerators like Plasticine would shine. To meet this low-latency requirement, the other prerequisite is that everything has to stay on-chip. To achieve this, we introduce limited mixed-precision support with changes localized to PCUs. The enhancement supports the commonly used precision in machine learning without introducing massive overhead from fine-grained reconfigurability.

In the rest of this section, Section 4.2.1 proposes the necessary micro-architectural changes to support low-precision arithmetics on Plasticine. Section 4.2.2 introduces a folded reduction structure in PCU's SIMD that improves the function unit (FU) utilization. Section 4.2.3 discusses architectural parameter selection for Plasticine to serve RNN applications efficiently.

4.2.1 Mixed-Precision Support

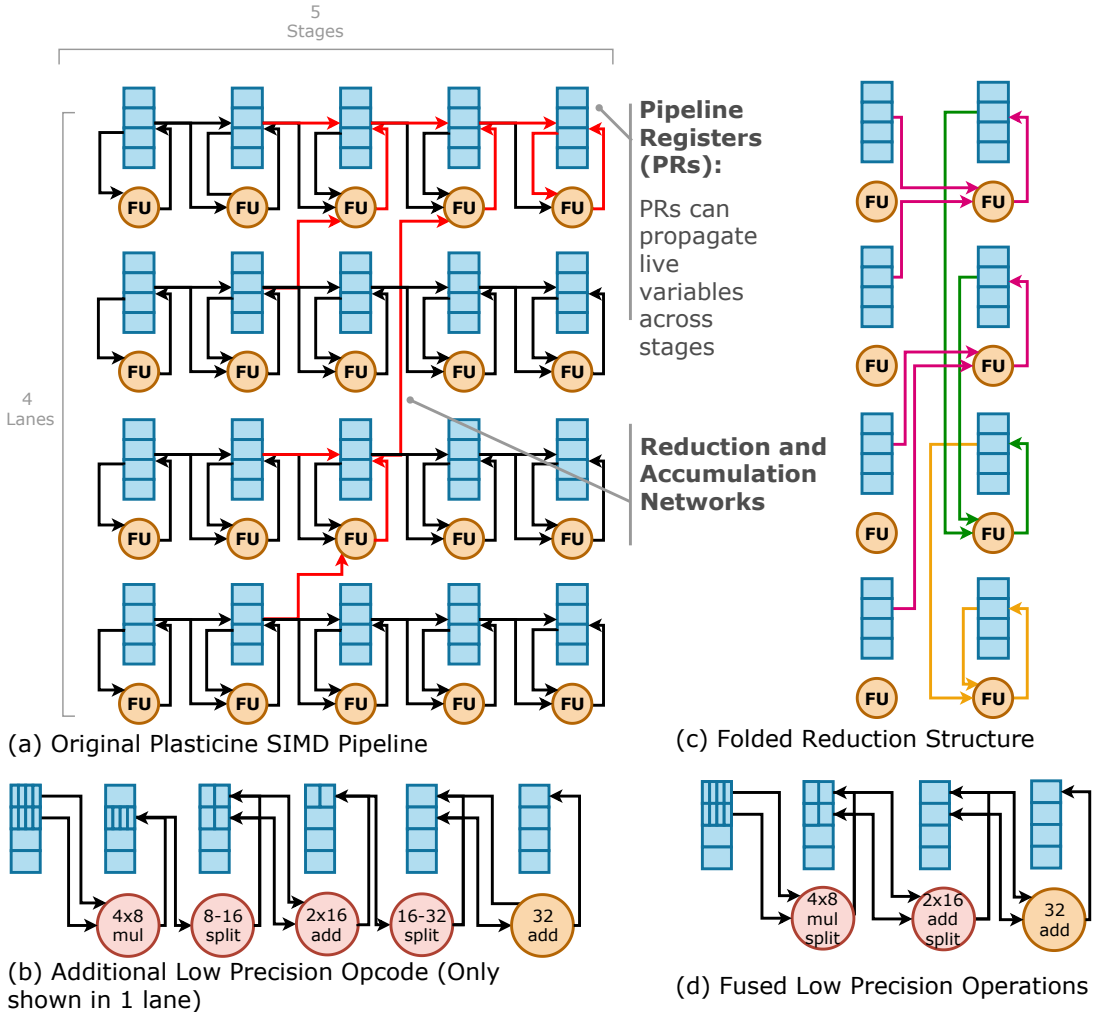
Previous works [9, 10] have shown that low-precision inference can deliver promising performance improvements without sacrificing accuracy. In the context of reconfigurable architectures such as FPGAs, low-precision inference not only increases compute density, but also reduces the required on-chip capacity 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 pack multiple low-precision values into single-precision storage. We support only 8 and 16-bit precisions, which are commonly seen in deep learning inference hardware. Users can only access values that are 32-bit aligned. This constraint guarantees that the microarchitectural change is only local to the PCU. PMU and DRAM access granularity remains intact from the original design.

Figure 4.2 (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 innermost 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 4.2 enable reduction operations. To support 8-bit element-wise multiplication and 16-bit reduction, we add 4 opcodes to the FU, shown in Figure 4.2 (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 4.2 (a) to complete the remaining reduction and accumulates in a 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 a 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 rearrange into the



(a) Original Plasticine SIMD Pipeline

(b) Additional Low Precision Opcode (Only shown in 1 lane)

(c) Folded Reduction Structure

(d) Fused Low Precision Operations

Figure 4.2: 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).

same stage. We also fuse the first low-precision reduction with the next rearrange shown in Figure 4.2 (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.

4.2.2 Folded Reduction Tree

The original reduction tree shown in Figure 4.2 (a) requires $\log_2(\#_{LANE}) + 1$ number of stages for $\#_{LANE}$ operations, leading to a low utilization of the FUs in the SIMD pipeline. The reduction tree also restricts the SIMD pipeline to have at least $\log_2(\#_{LANE})+1$ stages to compute a full reduction within a PCU. For a SIMD pipeline with 16 lanes, the FU utilization is only 53.33% in reduction stages.

To improve FU utilization, we introduce a folded reduction structure that performs $\#_{LANE}$ operations entirely within a single-stage pipelined over multiple cycles. Figure 4.2 (c) shows the folded reduce-accumulate structure. Instead of feeding the output of the reduction operation to the next stage, this structure folds the output back to the PR of the next unused FU in the same stage. The entire reduction plus accumulation is still fully pipelined in $\log_2(\#_{LANE}) + 1$ cycles with no structural hazard. In the original design, only a single register among the PRs have the special reduction tree connection. The downside of the folded structure is that no other live variables can be propagated after this reduction stage, unless adding multiple folded trees. In applications, we rarely see the need for multiple live variables after the reduction operation other than the accumulator itself. Therefore, it makes the most sense to put the folded reduction tree only in the last stage of a SIMD pipeline.

With the fused reduced-precision operations and the folded reduction tree, a PCU is able to perform 64 8-bit map-reduce in 4 stages, and 32 16-bit map-reduce in 3 stages. All the operations are still completed in $2 + \log_2(\#_{LANE})$ cycles, i.e. 8, 7, and 6 cycles for 8-, 16-, and 32-bit operations, respectively.

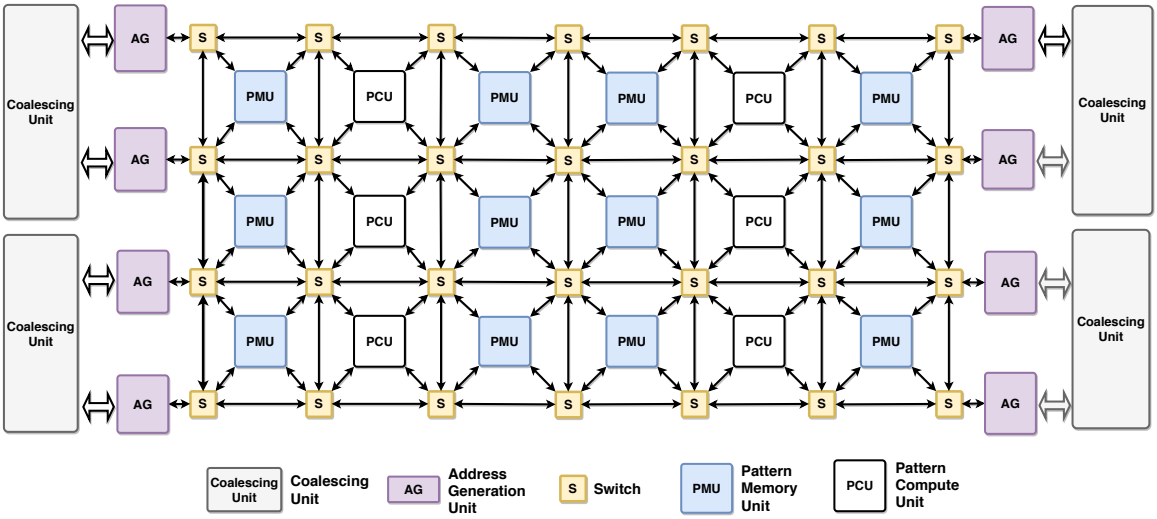


Figure 4.3: Variant Plasticine configuration for RNN serving with 2:1 ratio for PMU and PCU

4.2.3 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 a 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 six stages and 16 lanes, and a PMU has 16 banks, which gives 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. Figure 4.3 shows a specialized Plasticine configuration for RNN serving and general machine learning. Specifically, we choose a 2 to 1 PMU-PCU ratio with 4 stages in each PCU.

4.3 On-chip Network (WIP)

Achieving scalable 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.

On-chip 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 [**?**, **?**]. 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 section, 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 in Section 4.3.1. Using these insights, we describe a hybrid network with both static and dynamic capabilities to enable both high bandwidth traffic and high resource sharing. Section 4.3.2 identifies a space of interconnection networks with static and dynamic capabilities, at multiple granularities. Next, We explain our methodology on performance, area, and power modeling in Section 4.3.3. Finally, Section 4.3.4 perform a detailed evaluation across the identified design space for a variety of benchmarks.

Because CGRAs encompass a broad range of architectures, we narrow our study on tiled-based CGRAs with a streaming dataflow execution model, like Plasticine. At high-level, the architecture may contain a pool of heterogeneous compute and memory tiles (we

refer as physical units (PUs)) with a global network. The network guarantees exactly-once, in-order delivery with variable latency, and communication between PUs can have varying granularities (e.g., 512-bit vector or 32-bit scalar).

To generalize the study, we introduce a variant processing engine (PE) style than Plasticine's pipeline SIMD unit. The alternative style uses time-scheduled execution, where each PU contains a vector function unit (FU) that can executes a small loop of instructions repeatedly in time. The scheduling window is small enough that instructions are stored as part of the configuration fabric, without dynamic instruction fetch overhead. Compared to the pipelined architecture, this execution model creates more interleaved pipelining across PUs with communication that is tolerant of lower network throughput, which provides an opportunity to share links.

4.3.1 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 [4, **?**], 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 to achieve good performance on spatial reconfigurable accelerators is to explore multiple levels of parallelization and pipelining, within and across PUs. By default, pipeline parallelism across PUs introduces

high-bandwidth one-to-one communication between dependent stages. To balance the pipeline throughput, we often need to parallize the pipeline stage with the most computation, resulting in one-to-many communication when the receiver stage is parallelized, and many-to-one communication when the producer stage is parallelized. When both the producer and the consumer are parallelized, the worst case is many-to-many communication, as illustrated in Section 3.5.2. These broadcast and incast patterns introduce challenges in high-performance on-chip network, as their bandwidth demands scale quadratically with chip size in the worst case.

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 [4, **?**, **?**]. 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 [**?**]. 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 PU resources. As the compiler performs optimization passes, such as loop unrolling and memory partitioning, it has static knowledge about communication generated by these transformations. This knowledge enables compiler optimizations to improve network congestion, such as explained in Section 3.7.4.

To study the communication patterns, 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 4.1 lists the applications and their data size. Figure 4.4 shows, for each design, which resource limits

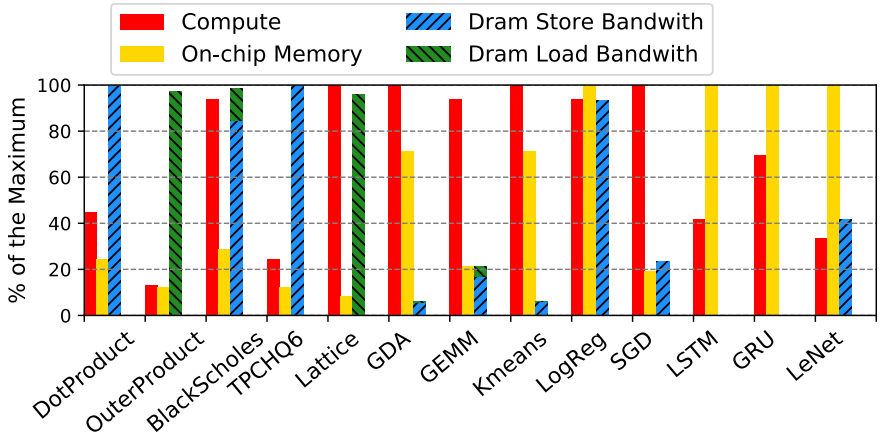


Figure 4.4: Physical resource and bandwidth utilization for various applications.

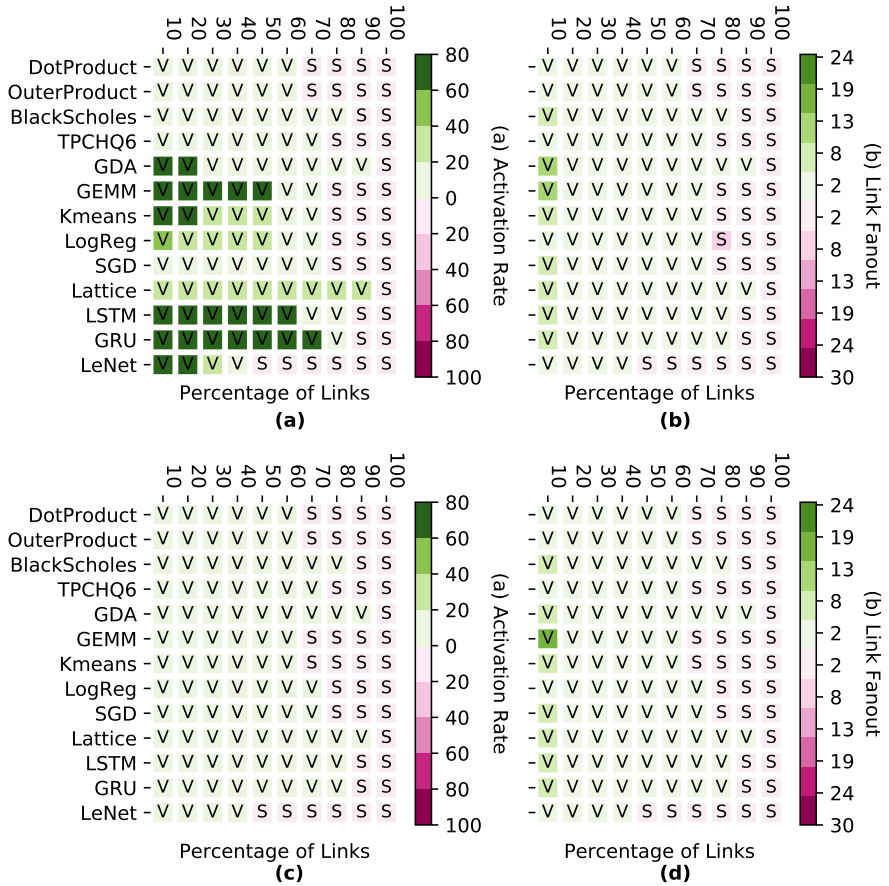


Figure 4.5: Application communication patterns on pipelined (a,b) and scheduled (c,d) CGRA architectures. (a) and (c) show the activation rate distribution of logical links at runtime. Links sorted by granularity, then rate; darker boxes indicate higher rates. The split between green and pink shows the ratio of logical vector to scalar links. (b) and (d) show the distribution of broadcast link fanouts.

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 4.1: Benchmark summary

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 4.5(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 4.5(c,d), exhibit lower bandwidth requirements due to the lower throughput of individual compute PUs. 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 4.6 shows statistics describing the VU dataflow graph (VGDFG) before and after the partitioning described in Section 3.5. The blue bars show the number of VUs, number of logical links, and maximum VU input/output degrees in the original parallelized program; the yellow and green bars show the same statistics after partitioning. Fewer VUs are partitioned for hybrid networks and dynamic networks with the time-scheduled architecture. When a VU in a pipelined CGRA consumes too many inputs or produces too many *distinct* outputs, SARA partitions 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 PU. We do not partition broadcasts with high output degrees because they are handled natively within the network. Finally, the output degree does not change with partitioning because most outputs with a large degree are from broadcast links.

4.3.2 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.

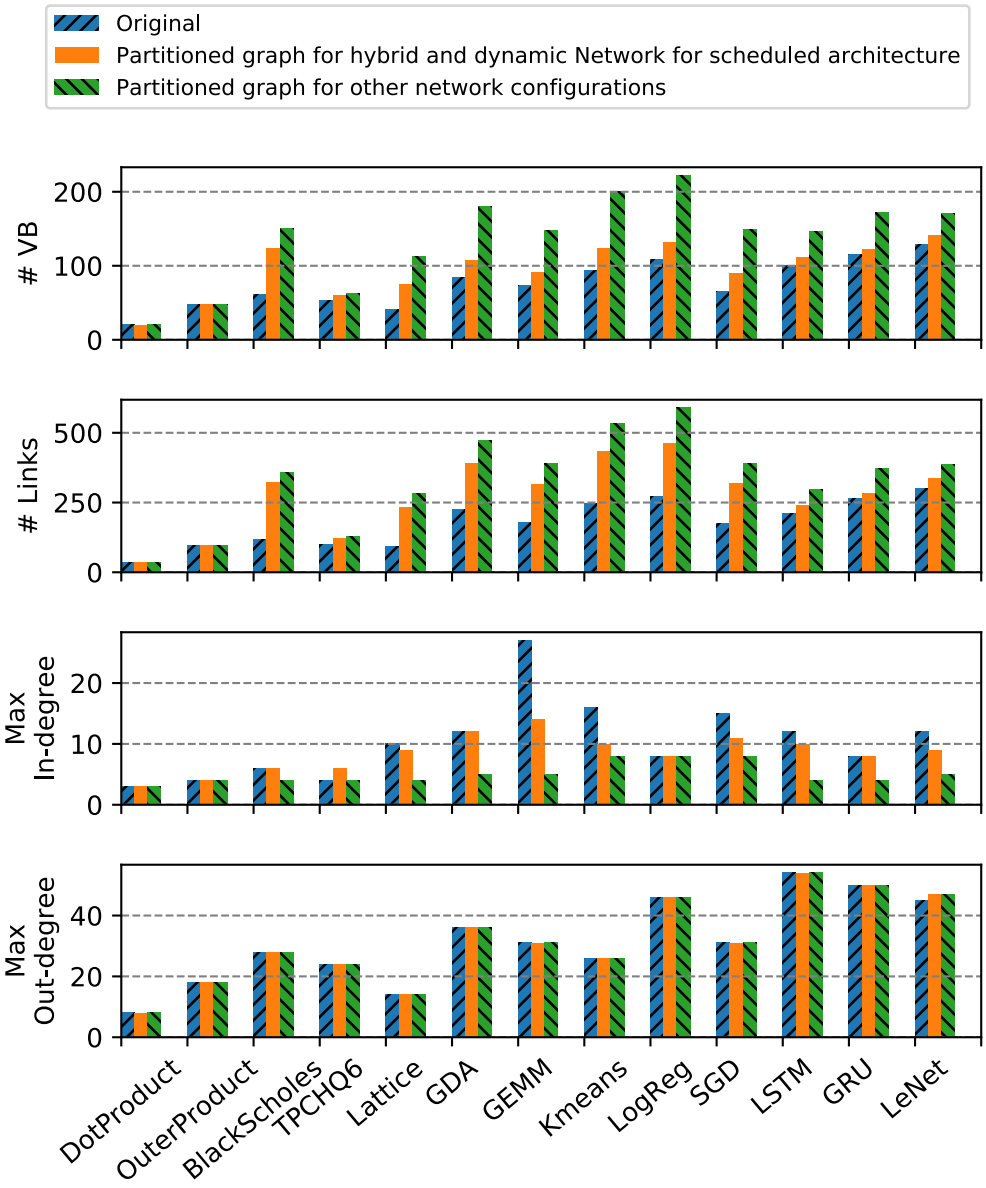


Figure 4.6: Characteristics of program graphs.

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 PUs 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 PU pushes the result to the next PU(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. PUs 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 PU 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.

4.3.3 Performance, Area, and Energy Modeling

Simulation

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 multicasting 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.

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 $28\,\mathrm{nm}$ 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 4.3.3, 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, E_{flit}) by dividing active energy by the number flits transmitted in the testbench:

$$E_{\text{flit}} = \frac{(P - P_{\text{inactive}})\, T_{\text{testbench}}}{\#\text{flit}} \tag{4.1}$$

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

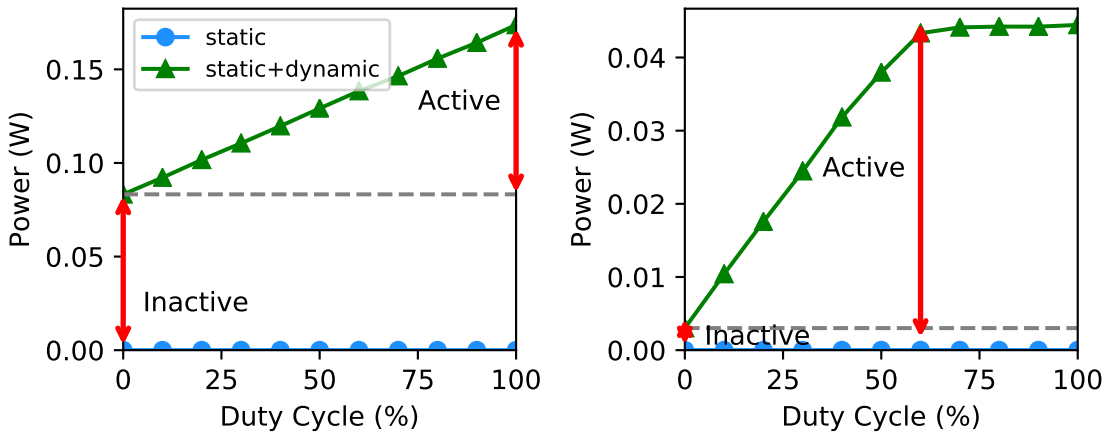


Figure 4.7: Switch and router power with varying duty cycle.

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:

$$E_{\text{net}} = \sum_{\text{allocated}} P_{\text{inactive}} T_{\text{sim}} + E_{\text{flit}} \#\text{flit}, \tag{4.2}$$

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

Figure 4.7 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 4.8. 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

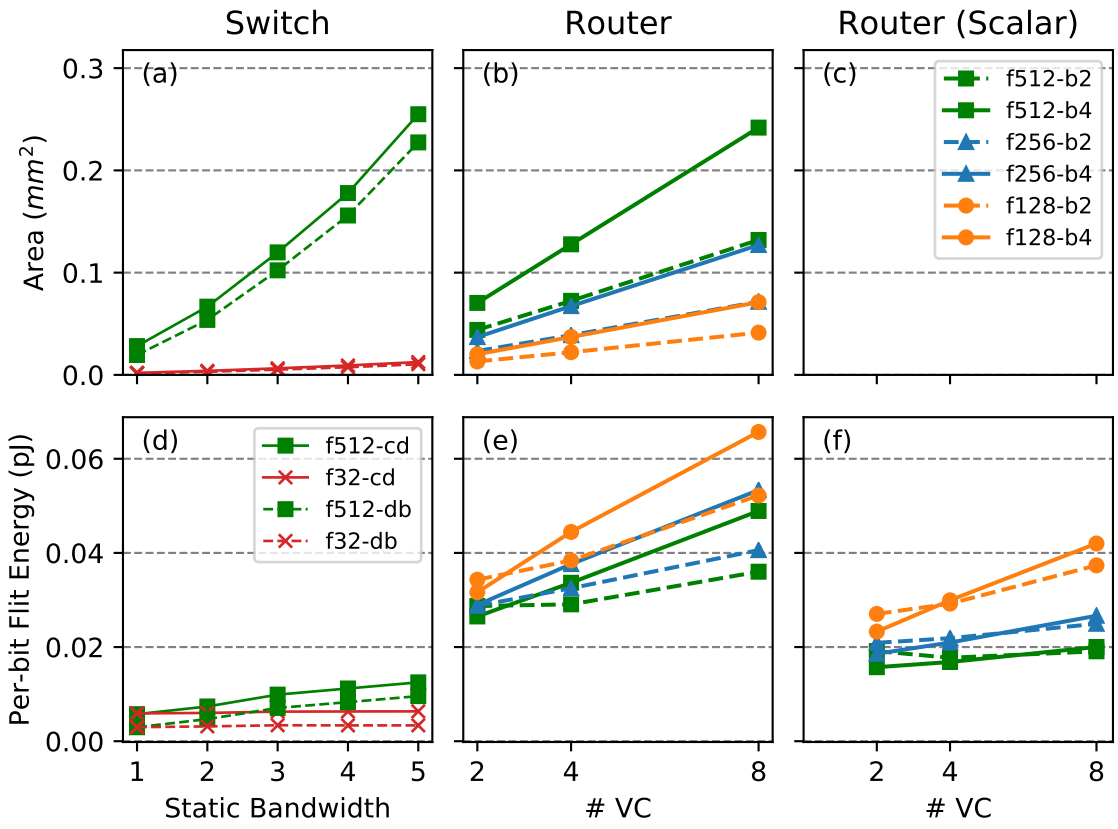


Figure 4.8: Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit).

overall router consumes less power (as shown in Figure 4.7); 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,

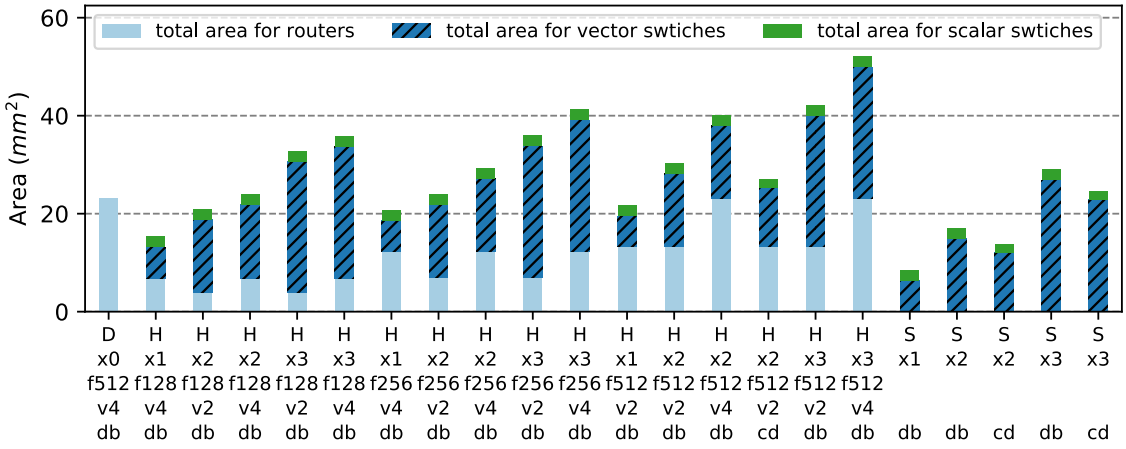


Figure 4.9: Area breakdown for all network configurations.

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 4.2: Network design parameter summary.

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

4.3.4 Network Architecture Evaluation

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 4.3.4, we use a network size of 14×14 end-point PUs. All vector networks use a vectorization factor of 16 ($512\,\mathrm{bit}$ messages).

Bandwidth scaling with network size

Figure 4.10 shows how different networks allow several applications to scale to different numbers of PUs. 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 PUs with a small network—which would be underutilized for non-vectorized problems—and smaller compute PUs with limited performance due to network overhead. For more finely grained compute PUs, 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

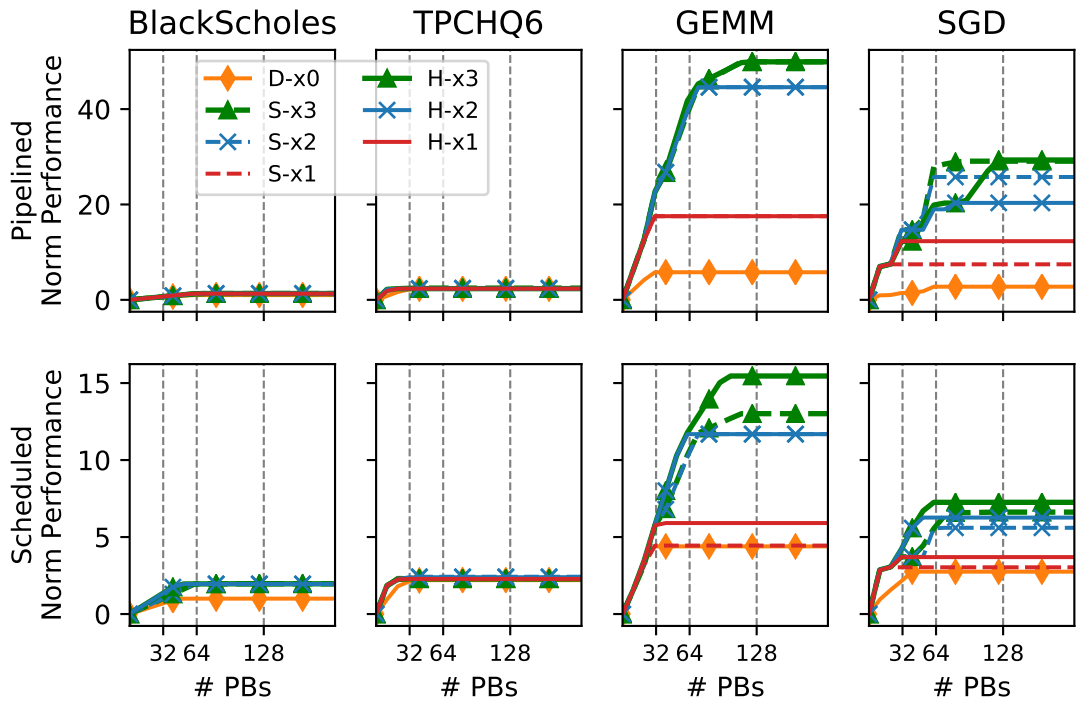


Figure 4.10: Performance scaling with increased CGRA grid size for different networks.

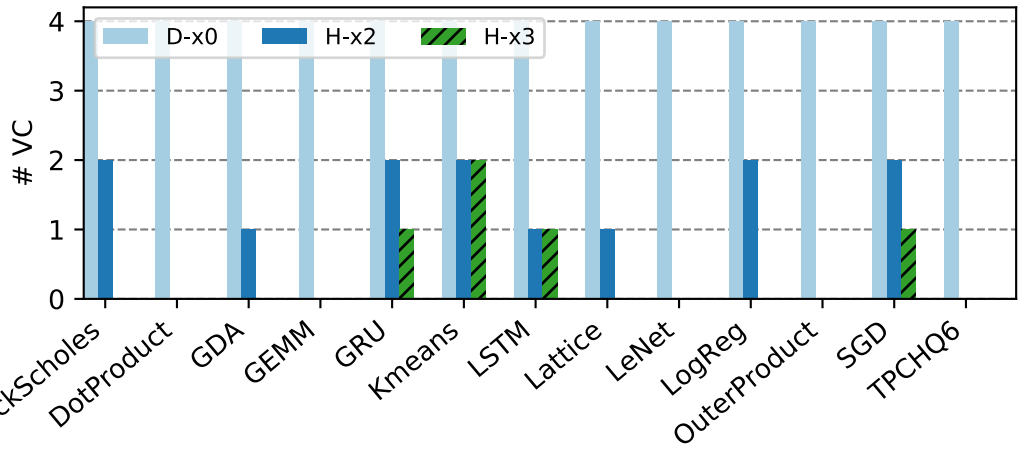


Figure 4.11: Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.)

that of deeply pipelined architectures (top row). Although communication between PUs 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 PU and its router, which prevents the PU 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 4.12, we show that increased static bandwidth results in a linear performance increase and a superlinear increase in area and power. As shown in Section 4.3.4, any increase in accelerator size must be

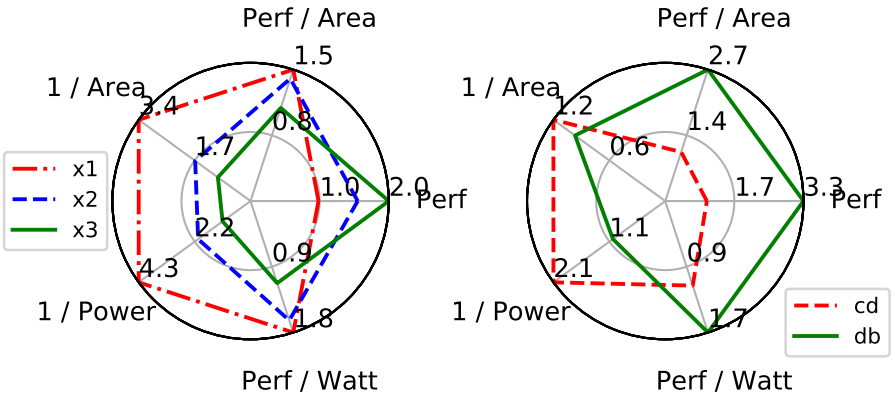


Figure 4.12: Impact of bandwidth and flow control strategies in switches.

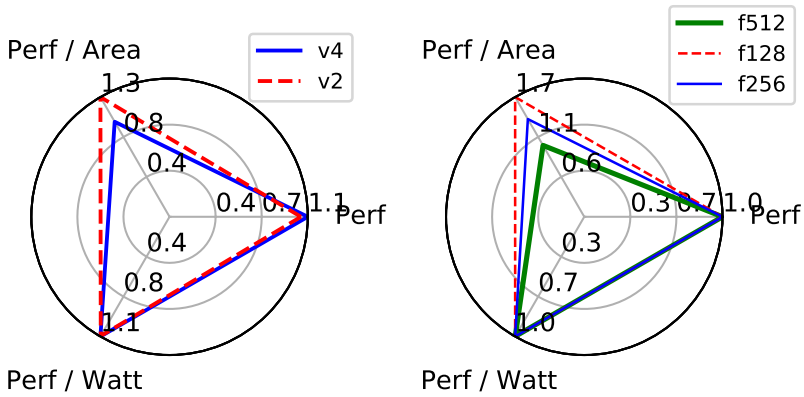


Figure 4.13: Impact of VC count and flit widths in routers.

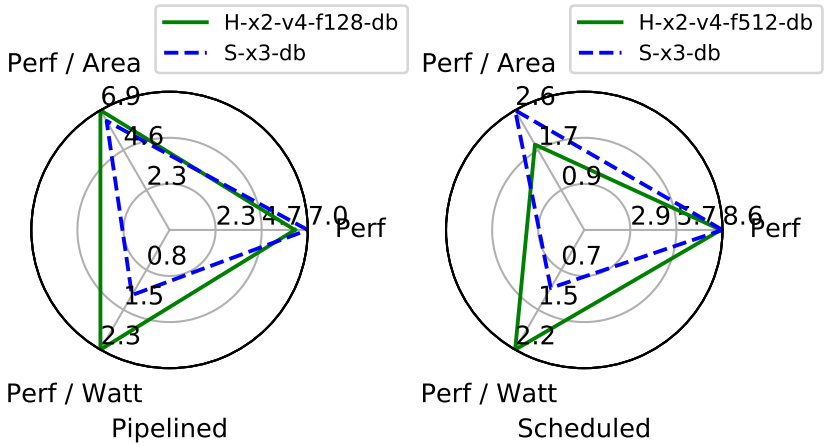


Figure 4.14: Geometric mean improvement for the best network configurations, relative to the worst configuration.

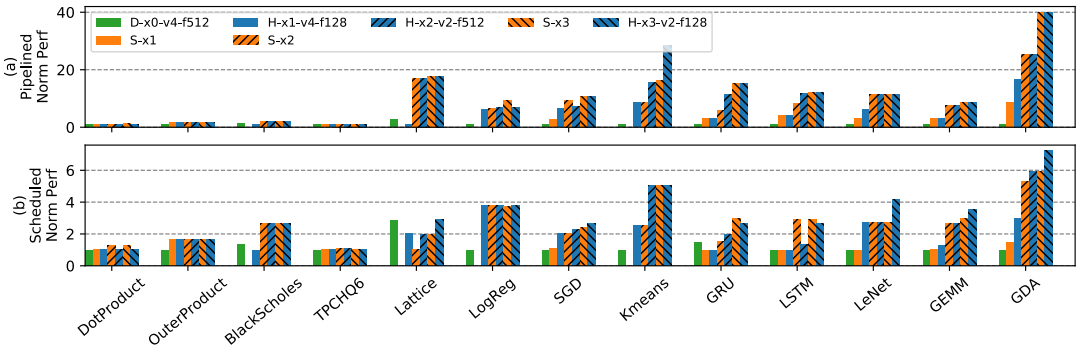


Figure 4.15: Normalized performance for different network configurations.

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 4.12 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.

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 4.3.3, 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 4.13 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 4.11 shows that the best hybrid network configurations with 2x and 3x static bandwidth require at most 2 VCs, whereas the

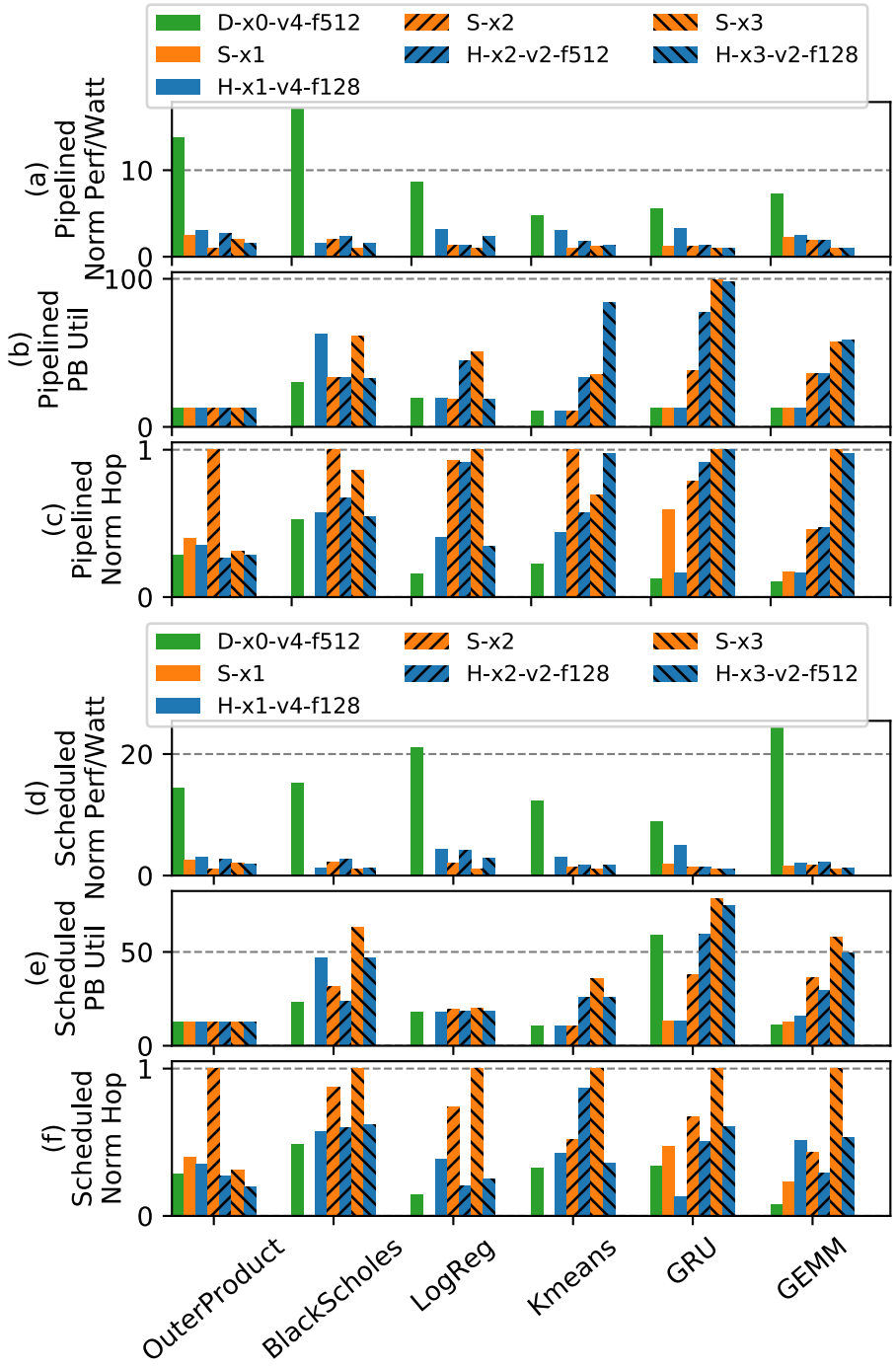


Figure 4.16: (a,d): Normalized performance/watt. (b,e): Percentage of compute and memory PUs utilized for each network configuration. (c,f): Total data movement (hop count).

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 4.3.3, 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 4.13 shows that, for a hybrid network, reducing flit width improves area efficiency with minimal performance loss.

Static vs. hybrid vs. dynamic networks

Figure 4.15 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 PUs, as shown in Figures 4.16(b,e), because it permits more parallelization. It also has more data movement, as shown in (c,f), because PUs 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 4.16(b,e), where networks D-x0-v4-f512 and S-x2 have small PU 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 4.16(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 PUs are utilized with static networks than hybrid networks. This is because the compiler imposes less stringent IO constraints on PUs when partitioning for the hybrid network (as explained in Section **??**), which results in fewer PUs, less data movement, and greater energy efficiency for hybrid networks.

In Figure 4.14, 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 5

Related Work

5.1 Network (Ready)

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

5.1.1 Tiled Processor Interconnects

Architectures such as Raw [**?**] and Tile [**?**] use scalar operand networks [**?**], 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 **??**, 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 [?] is a tiled dataflow architecture with dynamic execution. TRIPS does not have a static interconnect, but contains two dynamic networks [?]: an operand network to route operands between tiles, and an on-chip network to communicate with cache banks. Wavescalar [?] 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 [?] 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.

5.1.2 CGRA Interconnects

Many previously proposed CGRAs use a word-level static interconnect, which has better compute density than bit-based routing [?]. CGRAs such as HRL [?], DySER [?], and Elastic CGRAs [?] 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 [?, ?, ?] 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 [?] 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.

5.1.3 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 [?] 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 [?] focuses

on area and energy tradeoffs with different network topologies with the ADRES [**?**] architecture, where all topologies use a fully static interconnect. KressArray Xplorer [**?**] similarly explores topology tradeoffs with the KressArray [**?**] architecture. Other studies explore topologies for mesh-based CGRAs [**?**] and more general CGRAs supporting resource sharing [**?**]. Other tools like Sunmap [**?**] allow end users to construct and explore various topologies.

5.1.4 Compiler Driven NoCs (WIP)

Other prior works have used compiler techniques to optimize various facets of NoCs. Some studies have explored statically allocating virtual channels [**?**, **?**] 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 [**?**]. 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. [**?**] 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. [**?**] use a static placement algorithm and an estimate of reliability to attempt to guide placement decisions for NoCs. Kapre et al. [**?**] 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.

5.2 Compiler

Streaming Dataflow IRs Although many works claim to emit efficient and information-rich dataflow IRs for the downstream compilers, very few of them can capture the high-level parallel patterns and implementation details that are critical to RDA mappings. For example, TensorFlow [**?**] emits dataflow IR composed of tensor operations. However, its IR lacks information on the parallel patterns within these operations. In contrast, most

of the streaming languages [**?**, **?**, **?**] are not able to extract nested loop-level parallelism from modern data-intensive applications. For example, StreamIt [**?**], a language tailored for streaming computing, also adopts distributed control as in SARA. However, it lacks the necessary language features to describe deeply and irregularly nested loops that are common in modern data-intensive applications.

Hardware Architectures Spatial reconfigurable accelerators (*e.g.*, Dyser [**?**] and Tartan [**?**]) have only one-level of hierarchy. Hence, such accelerators' performance can be bottlenecked by their limited interconnect bandwidth and power budget. Sparse Processing Unit (SPU) [**?**] can sustain higher interconnect bandwidth by introducing on-chip hierarchy; however, it lacks support for polyhedral memory banking [8], a pivotal optimization to achieve massive parallel accesses to on-chip memory. Plasticine [4] provides us with the desired architecture features; however, its compiler lacks the necessary components to support efficient streaming execution. Given that Plasticine resembles many key features of the RDA model, we target Plasticine with SARA.

Spatial Compilers Most previous works [**?**, **?**] only consider allocating resources at the same level. SARA takes a more general assumption by co-allocating resources at multiple levels of an accelerator's hierarchy.

The Plasticine compiler [4] is similar to SARA that it also uses a token-based control protocol. However, it performs worse than SARA due to the following reasons. First, the Plasticine compiler allocates VBs for every level of Spatial's (a high-level language) control hierarchy. The communication between parent and child controllers lead to both communication hotspots around the parent, and bubbles before entering a steady-state of the loop iterations. Second, the Plasticine compiler assigns a single memory PB for each logical memory in the Spatial program. Hence, it could not handle the case where a logical memory exceeds the capacity or bank limits of the physical PBs. Third, the Plasticine compiler only supports polyhedral memory partitioning at the first dimension of the on-chip

memory. Hence, its applicability to data-intensive applications with high-dimension tensor algebra is questionable. Last, compared to SARA's separate allocation and assignment phases described in Section 3.4 and Section 3.5, the Plasticine compiler allocates one VB for a specific type of PB and underutilizes resources within PBs.

Chapter 6

Future Work (WIP)

Chapter 7

Conclusions (WIP)

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.

Appendix A

Appendix

A.1 Context Programming Restrictino

We use the imperative context configuration for ease of understanding, but it is important to realize not all program expressible at this abstraction can be executed by the PCU.

```
with Context() as ctx:
    bufferA = VecInSteram()
    bufferB = VecInStream()
    bufferN = ScalInStream()
    bufferAcc = ScalOutStream()

    for i in range(0, N, 16)
        a = bufferA.deq()
        b = bufferA.deq()
        # vectorized multiply
        prod = a * b
        # produce a scalar ouptut
        r = reduce(prod, lambda a,b: a+b)
        # scalar accumlation in PR
        accum += r
    # sends bufferAcc every N/16 cycles
    bufferAcc.enq(accum)
```

(a) Declarative configuration

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    b = bufferB.deq()
    for i in range(0, 3, 1)
        c = bufferC.deq()
        for j in range(0, 3, 1)
            a = bufferA.deq()
            expr = a * b + c
            outputD.enq(expr)
```

(b) Imperative configuration

Bibliography

[1] M. Grant, *Disciplined Convex Programming*. Phd. thesis, Stanford University, 2014.

[2] Y. Y. Michael Grant, Steven Boyd, "Disciplined convex programming," 2019.

[3] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "Sda: Software-defined accelerator for largescale dnn systems," Hot Chips 26, 2014.

[4] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 389–402, ACM, 2017.

[5] 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.

[6] 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.

[7] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2019.

[8] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, (New York, NY, USA), pp. 199–208, ACM, 2014.

[9] 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.

[10] 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.