Compiler-Driven Simulation of Reconfigurable Hardware Accelerators

Zhijing Li*, Yuwei Ye*, Stephen Neuendorffer†, Adrian Sampson*
*Cornell University, †Xilinx Inc.
*{zl679, yy453, asampson}@cornell.edu, †stephenn@xilinx.com

***Abstract*—As customized accelerator design has become increasingly popular to keep up with the demand for high performance computing, it poses challenges for modern simulator design to adapt to such a large variety of accelerators. Existing simulators tend to two extremes: low-level and general approaches, such as RTL simulation, that can model any hardware but require substantial effort and long execution times; and higher-level application-specific models that can be much faster and easier to use but require one-off engineering effort.**

This work proposes a compiler-driven simulation workflow that can model configurable hardware accelerator. The key idea is to separate structure representation from simulation by developing an intermediate language that can flexibly represent a wide variety of hardware constructs. We design the Event Queue (EQueue) dialect of MLIR, a dialect that can model arbitrary hardware accelerators with explicit data movement and distributed event-based control; we also implement a generic simulation engine to model EQueue programs with hybrid MLIR dialects representing different abstraction levels. We demonstrate two case studies of EQueue-implemented accelerators: the systolic array of convolution and SIMD processors in a modern FPGA. In the former we show EQueue simulation is as accurate as a state-of-the-art simulator, while offering higher extensibility and lower iteration cost via compiler passes. In the latter we demonstrate our simulation flow can guide designer efficiently improve their design using visualizable simulation outputs.

***Keywords*-Programming Language; MLIR; Multi-level Abstractions; Simulation; Accelerators; Reconfigurable Hardware**

I. Introduction

Hardware accelerators are a central tool for improving efficiency in the post-Moore era. Successful accelerators cannot be designed in a vacuum: realizing their full potential requires simultaneous advances in algorithms and compilers. Co-design between hardware and its accompanying software stack requires a way to rapidly simulate a proposed hardware accelerator before finalizing its design.

However, standard approaches to hardware simulation, can impede this kind of *rapid iteration*. For instance, although RTL simulation [15], [16], [23], [33] is valuable when the hardware is being finalized at the *end* of the design process, it tends to be too detailed and too slow to be practical for earlier hardware–software co-design phases. Designers often build custom *high-level* simulators [4]–[6], [32] for specific applications and architectures that sacrifice accuracy for greater flexibility and faster simulation times. However, these custom simulators specialize for a specific architecture model—changing the modeled hardware, such as introducing a new level in a memory hierarchy, requires rewriting substantial parts of the simulator. *General-purpose* simulation frameworks, in contrast, tend to focus on processor-centric architectures like CPUs and GPUs [2], [27], [34], meaning that modeling a custom accelerator architecture still requires a custom specialized implementation of the simulation logic. Finally, traditional simulators do not expose an intermediate representation, so it can be challenging to integrate them with a compiler stack to measure performance on real software.

This paper presents a general framework for rapidly implementing high-level simulators for arbitrary hardware accelerators. This framework shares basic discrete-event semantics with many existing simulation systems [28], [37] and focuses on implementation in a multi-level compiler infrastructure, MLIR [20]. This implementation enables rapid iteration and efficient, low-effort simulation of generated architectures and is intended to exist as part of an end-to-end toolchain, rather than as a standalone simulation framework.

The system has two main components: an MLIR *dialect* for representing hardware accelerators, and a generic simulation engine that interprets those representations. Our core contribution is an *event queue* (EQueue) dialect in MLIR: an intermediate language that represents accelerators at many levels of detail, from simple first-order models to detailed, multi-component simulations. The dialect focuses on expressing memory allocation, data movement, and parallel execution with event-based control. The dialect is directly executable by a generic timed discrete-event simulation engine, enabling arbitrary architectures to be executed. The simulation can provide estimates of overall execution time taking into account data dependencies and resource limitations. The simulation engine also produces visualizable operation-level traces for detailed performance analysis.

By building on MLIR, the EQueue dialect leverages a broad ecosystem of transformations, analyses, and other dialects. Designers can quickly prototype compilers from high-level languages to lower-level accelerator configurations.

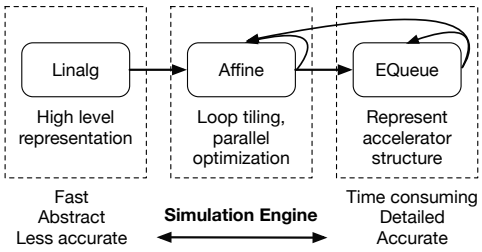


Figure 1: An example of EQueue simulation on different levels of hardware abstraction.

As a software compiler infrastructure, MLIR also enables analysis and transformation tools that are not possible with one-off simulators. For example, Fig. 1 shows a lowering pipeline for progressive optimization of tensor computations using existing MILR dialects: the high-level *Linalg* dialect to represent tensor operations, the *Affine* dialect to express explicit loop tiling, and finally our new EQueue dialect to model explicit data movement among hardware components. Critically, such a lowering pipeline enables simulation at multiple levels of detail: users can get quick-and-dirty performance estimates at the Linalg level on tensor behavior, or they can lower gradually to more detailed EQueue hardware simulations for more accurate but costly estimation.

Compared to traditional one-off accelerator simulators, we see several advantages in the compiler-based approach:

1) We can simulate at different points in a compilation flow, representing a hardware at multiple abstraction levels.
2) Changes to the architecture are decoupled from changes to the simulation logic, so design iteration can be easier.
3) By reusing compiler passes with different parameters to transform the architecture, designers can easily switch among different architectures for the same computation.

This paper presents the EQueue dialect and its open-source[1] implementation using MLIR. We explain the core constructs via a running example (Section II) and then detail the dialect (Section III) and its simulation engine (Section IV). We show the flexibility of programming with two case studies: a systolic array for deep learning computations , and a model of the AI Engine cores in Xilinx's Versal ACAP fabric. In the first case, we compare our EQueue-based simulator against a traditional, special-purpose systolic array simulator, SCALE-Sim [32]. The EQueue approach matches its accuracy while offering better flexibility to rapidly change the modeled data flow (Section VI). In the second case, we demonstrate how EQueue's flexibility can guide designers to improve their designs on a real-world reconfigurable architecture (Section VII).

[1]https://github.com/cucapra/EventQueue

II. Overview by Example

This section summarizes our simulation flow. We write an EQueue program (Fig. 2a) and show how our simulation engine executes it. Fig. 2b depicts the modeled architecture.

A. Structure Specification

The first main part of an EQueue program is a set of *structural* declarations, which define the hardware resources that make up an accelerator. Fig. 2a lists an EQueue program describing a toy accelerator in Fig. 2b. First, `create_*` operations instantiate components like processing elements (PEs) and memories. Then, `launch` operations map work onto this structure to specify the computation.

We start with structure specification (1). The program uses several `create_*` operations to declare components including processors, memories, and direct memory access (DMA) units. These `create_*` operations select from a range of primitive component types (`ARMr6`, `SRAM`, `Register` , etc.). These tags correspond to performance models in the simulation engine: for example, the simulation model for `SRAM` components has slower warm-up time, slower reads, and higher power usage than the `Register` model. Programs can assemble these components into hierarchies using `create_comp` to create a new component and `add_comp` to add one to an existing component.

B. Control Flow

The second part of a EQueue program is its control flow. The core operation is `launch`. The `launch` operation takes a dependency, a processor component, and a block of code. The simulation engine implements `launch` by issuing code blocks on processors, which execute sequentially.

`launch` or `memcpy` are event operations executed out of order. Every processor in the simulation has an *event queue*; launching a code block enqueues it for the given processor. At simulation, the engine checks the dependency and executes code blocks when their dependencies are ready. Processors communicate by spawning events with `launch` or `memcpy`.

For example, the control flow (2) illustrates how the `Kernel` processor distributes its work to `DMA`, `PE0`, and `PE1`. Fig. 2b indicates the communication using arrows. `DMA`, `PE0`, and `PE1` are all independent processors but can communicate through their event queues. As the `launch` of `PE0` and `PE1` both depend on `mempcy` operation of `DMA`, `PE0` and `PE1` start simultaneously.

Benefits. This example shows how EQueue describes accelerator structure and control flow *separately* from the simulation logic. Designers can change the architecture without needing to modify the simulation engine, which reduces the cost of exploring alternative designs. It also shows how EQueue programs can intermix code from other

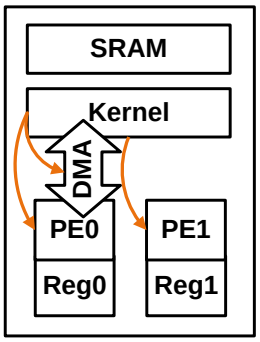
1 **Structure Specification**

```
kernel = equeue.create_proc(ARMr6)
sram = equeue.create_mem(SRAM, [64], 4)
dma = create_dma()
accel = create_comp("Kernel SRAM DMA",
  host, sram, dma)
pe0 = equeue.create_proc(MAC)
reg0 = equeue.create_mem(Register, [4], 4)
pe1 = equeue.create_proc(MAC)
reg1 = equeue.create_mem(Register, [4], 4)
add_comp(accel, "PE0 Reg0 PE1 Reg1", pe0,
reg0, pe1, reg1)
```

2 **Control Flow**

```
start = equeue.control_start()
done = equeue.launch(...) in (start, kernel){
    copy_dep = equeue.control_start()
    launch_dep = equeue.memcpy(copy_dep, DMA, …)
    pe0_dep = equeue.launch(...) in (launch_dep, pe0){
        ...
        ofmap = addi(ifmap, 4)
        ...
    }
    pe1_dep = equeue.launch(...) in (launch_dep, pe1) { … }
    equeue.await(pe0_dep, pe1_dep)
    equeue.return()
}
```

(a) An EQueue program as input to our generic simulator.



(b) Accelerator.

Figure 2: Modeling an accelerator with an EQueue program and the model created by the simulation engine. Code listings omit types and the % prefix for legibility.

MLIR dialects for core accelerator logic: namely, the `addi` operation is from MLIR's standard dialect. The MLIR ecosystem offers operations from many abstraction levels, from linear algebra to machine code. These help to express architectures at different levels, from abstract black boxes to low-level implementation details.

III. EQueue Dialect

We illustrate the EQueue dialect with four parts: modeling hardware, expressing data movement, launching computation and concurrency between controllers.

A. Modeling Structure

The EQueue dialect lets programs declare structural components to model hardware. For example, this code creates the structure from Fig. 3:

```
kernel = equeue.create_proc(ARMr5)
mem = equeue.create_mem([4096], 32, 4, SRAM)
dma = equeue.create_dma()
accel = equeue.create_comp("Memory Kernel DMA",
mem, kernel, dma)
```

There are three kinds of components: processors, memories, and DMA engines. A processor is a component that can execute commands, via the `launch` operation described in Section III-D. A DMA component is a specialized processor that is only used for data movement. A memory stores data; its speed is affected by its type, size, and ports. Each `create_*` operation encodes component properties in its arguments. For instance, `create_mem([4096], 32, 4, SRAM)` declares a memory component of SRAM type with 4 banks and 4096 data elements of 32 bits each.

The `create_comp` composes smaller components into a hierarchy of larger components. The example code declares a component `accel` with three subcomponents with the names "Memory", "Kernel" and "DMA". Later, code can look up components in the hierarchy using a `get_comp` operation:

```
dma = equeue.get_comp(accel, "DMA")
```

Finally, *connections* model bandwidth constraints:

```
connection = equeue.create_connection(Streaming,
    32)
```

The `create_connection` operation has two arguments: their type and their bandwidth in bytes per cycle. The two types are `Streaming`, which allows simultaneous reads and writes, and `Window`, which models a buffer that requires locking for exclusive access. Streaming interfaces typically offer lower latency while windowed interfaces offer higher bandwidth. The simulation engine outputs profiling statistics for each connection's bandwidth utilization over time. The bandwidth limit is optional; the simulation engine can also model infinite-bandwidth connections and still collect statistics, as we show in Section VII-C.

B. Explicit Data Movement

Given the hardware structure, we can specify data movement using allocation, deallocation, and read/write operations on memories. For example, consider two memories:

```
mem0 = equeue.create_mem([4096], 32, 4, SRAM)
mem1 = equeue.create_mem([4096], 32, 4, SRAM)
conn = equeue.create_connection(Streaming, 32)
```

We will model the data movement shown in Fig. 4. To associate a buffer with a memory, we use `alloc`:

```
buffer0 = equeue.alloc(mem0, [64], 32)
buffer1 = equeue.alloc(mem1, [64], 32)
```

The `alloc` operation specifies the memory, buffer size in elements, and element size in bits.

To model data movement, we use `read` and `write` operations to transfer data into a buffer, optionally through a connection:

```
data = equeue.read(buffer0, conn)
equeue.write(data, buffer1, conn)
```

Both operations take a buffer and, optionally, a connection; `write` also takes the value to write. Here, we use connection

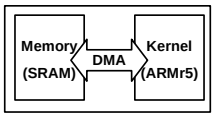
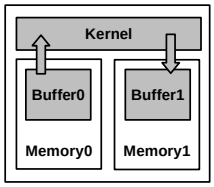
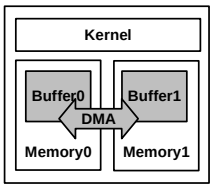


Figure 3: Simple one-core accelerator.

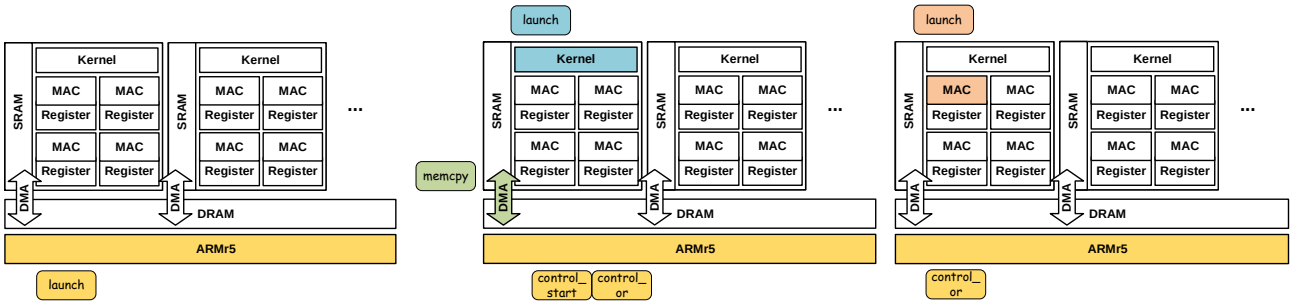


(a) Data movement controlled by kernel.



(b) Data movement controlled by DMA.

Figure 4: Expressing explicit data movement using EQueue dialect.



(a) Event queue of ARMr5 processor when `launch` waits on its dependency.

(b) Event queues of Kernel and DMA when `launch` and `memcpy` wait on their dependency.

(c) Event queue of MAC when `launch` wait on its dependency.

Figure 5: Three stages of the timeline of execution of the accelerator in Fig. 6.

whose bandwidth is 32 bytes per cycle. Finally, programs use `dealloc` to free buffers:

```
equeue.dealloc(buffer0)
equeue.dealloc(buffer1)
```

So far, these operations specify how data moves, but not the processors executing the operations. The next subsection shows how to assign this code to processors.

C. Launching Computations

The `launch` operation schedules blocks of code onto processors. This code runs a block on the `kernel` processor:

```
equeue.launch(buffer0, buffer1 = b0, b1)
  in (kernel){
    data = equeue.read(buffer0)
    equeue.write(data, buffer1)
    equeue.return
}
```

The arguments to `launch` pass resources that the code block, represented by an MLIR region, may access. The code in the region will be dispatched to the particular processor for execution. When the region is executed, the resources will be available, enabling the region to run to atomically run to completion. Although most The code runs sequentially. Fig. 3a illustrates the above data movement.

The `memcpy` operation is syntactic sugar for a `launch` that reads and then writes data. `memcpy` is mostly used with DMA units. Fig. 3b shows the data movement in this code:

```
equeue.memcpy (mem0, mem1, dma)
```

Launching a code block enqueues it for later execution; the next section describes how this queueing work.

D. Concurrent Event Scheduling

Concurrency in the EQueue dialect occurs at the granularity of *events*. While the code within a `launch` block executes sequentially, it can use *event operations* to spawn asynchronous, concurrent work. Event operations include `launch` and `memcpy` described above, and also logical operations on events: `control_start`, `control_and` and `control_or`.

Event operations can have *dependencies*, indicating other events they depend on that must finish before the event can start. `launch` and `memcpy` each have one dependency, and `control_start` has none: it is a special operation for beginning a chain of events. `control_or` and `control_and` are ready when any or all of their dependencies finish, respectively.

During simulation, launching an event pushes it onto a given processor's *event queue*. Different processors can execute events from their queues in parallel, but each processor only executes one event at a time. Events can launch other events, so simulations can nest `launch` operations in arbitrary ways to reflect their control hierarchy.

The EQueue dialect includes an `await` operation that blocks execution until a different event completes. Finally, a `launch` block can pass values out with the `return` operation.

Example. Fig. 6 shows an example accelerator that uses concurrent tasks, and Fig. 5 illustrates the timeline of its execution. One ARMr5 processor uses a DMA unit for data transfer and a small MAC kernel using `launch` operations. The `control_*` operations encode the execution order while allowing parallel execution on the DMA and MAC kernel.

Fig. 4a highlights the event queue of `ARMr5` when it waits

```
start = equeue.control_start()
// Event in Fig. 5(a).
done, ret = equeue.launch (buf0, buf1, dma,
    kernel, mac = b0, b1, d, k, m)  in (start,
    ARMr5){
    // Event in Fig. 5(b).
    start_event = equeue.control_start()
    done_dma = equeue.memcpy(start_event,
        buf0, buf1, dma)
    done_kernel = equeue.launch(MAC=mac) in
        (start_event, kernel){
        // Event in Fig. 5(c).
        start_mac = ...
        done_mac = equeue.launch(...) in
            (start_mac, MAC){ ... }
        ...
    }
    done_compute = equeue.control_and(
        done_kernel, done_dma)
    equeue.await(done_kernel)
    equeue.return(done_compute)
}
```

Figure 6: Example showing concurrent execution of an ARMr5 control processor, a DMA engine, and a MAC unit.

on `start` event to issue `launch`. Fig. 4b shows when `start` is generated, the `launch` is removed from the event queue of `ARMr5`. Then, `control_start` and `control_or` are pushed to the queue of `ARMr5`, `launch` is pushed to the queue of `kernel` , and `memcpy` is pushed to the queue of `dma`. This way, the event operations run concurrently since they do not block the execution of other operations in `ARMr5`'s `launch` block. Finally, since `memcpy` of `dma` and `launch` of `kernel` both depend on `start_event`, once it finishes, both `memcpy` and `launch` can be issued from their corresponding event queues, as indicated by Fig. 4c. Fig. 4c also shows that `launch` of `MAC`, after its dependency finishes, is pushed to event queue of `MAC`. The `return` operation passes `done_compute` back to the top level as the result value, `ret`. Notice that `done`, the first return value of `launch`, is the dependency generated by `launch`.

E. Introducing External Operations

Sometimes there are special cases where existing MLIR dialects cannot express a specific hardware operation. We introduce `op` to address this situation:

```
res0, ... = equeue.op("mac", {arg0, arg1, ...})
```

`op` takes in a *signature* specifying the operation name and an arbitrary number of inputs and outputs. Here the signature is `"mac"`, which can be modeled as multiplication and addition in the one cycle in the simulator library. The simulation engine checks the signature to jump to the operation's implementation specifying cycle counts and the simulated behavior.

IV. Simulation

This section introduces the EQueue simulation engine. Fig. 7 shows an overview of the simulation workflow.

A. Inputs

The simulation engine takes in an EQueue program. As Fig. 7 shows, an EQueue program is composed of a structure definition and a control flow. Designers can produce EQueue programs by writing simple *generators* in C++, as we demonstrate in Section VI-A. Alternatively, compilers can translate to EQueue from high-level dialects such as Linalg, as we show in Section VI-D. The infrastructure includes many reusable passes (Section V) to enable these lowering pipelines.

B. Outputs

The simulation engine outputs a profiling summary and a visualizable tracing file. The profiling summary includes the simulation execution time, the simulated runtime in cycles, read and write bandwidth for each connection, maximum bandwidth, and the total bytes read or written for each memory. We also report a *max bandwidth portion* for both read and write bandwidth, which is the fraction of the total simulated runtime spent at a channel's maximum bandwidth. The designer can use this statistic to adjust bandwidth accordingly to avoid waste or increase computation utilization.

The trace is a JSON file with operation-wise records in event trace format [9]. The Chrome browser can visualize this event trace format [10]. We show in Section VII on how to use this visualization to address a performance bottleneck.

C. Simulation Engine

The simulation engine loops over four stages: set up entry, check event queue, schedule operation, and finish operation. The first stage sets up an *operation entry* for each processor's current and next operation. The second stage checks the head of each processor's event queue to decide whether to issue it. The third stage models operations' execution time by updating the time logs in the operation entry. To estimate execution time, the simulation engine uses a state object for each component. For instance, a memory component uses banks, cycles per access, and read/write ports to calculate the time for a read or write operation. Each component uses a *schedule queue* to track operations and to model delay when contention happens, such as when two concurrent writes contend for the same memory. The final stage models the effect of each operation by resetting its operation entry when it finishes.

D. Extending the Simulator Library

The simulation engine implements the primitives for EQueue programs using an extensible set of *operation functions* and a *component library*. The interface for operation functions consists of a cycle count and a stall signal. In the

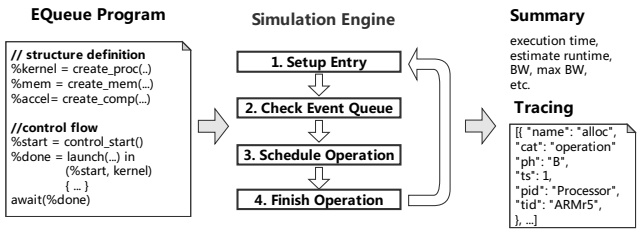


Figure 7: The simulation workflow.

simulator's third stage, where it schedules operations (see previous section), it queries each operation function to obtain timing information. At this point, the operation function may invoke a component object.

The EQueue infrastructure provides a standard library of components, such as SRAM memories and processors. Designers can extend the library with custom components to introduce custom simulation logic. To introduce a cache component, for example, a user would add a new `Cache` class to the component library and define an operation function to support `create_mem("Cache", ...)` in EQueue programs. The operation function simply instantiates the cache component object. The `Cache` class can inherit from a base `Memory` component class; the user only needs to override a method called `getReadOrWriteCycles` to determine whether the access is a hit or a miss and report a latency accordingly. The `Memory` class inherits from a more general `Device` class that manages one or more scheduling queues to avoid conflicts. In the case when there is conflict, the operation function returns a stall signal instead of a cycle count. By extending these base classes, users can specify arbitrary behavior for components in EQueue programs.

V. Lowering Passes

The EQueue dialect provides a set of reusable compiler passes. Designers can combine these passes to build accelerators for simulation. We will show how to use these passes with the case study of systolic array in Section VI-D.

1) EQueue Read Write Pass: This pass translates `load` and `store` in MLIR's Affine dialect to EQueue's `read` and `write`.

2) Allocate Memory Pass: This pass allocates buffers on a specified memory component.

3) Launch Pass: This pass adds `launch` operations by taking in a specified processor component and a code block.

4) Memcpy Pass: This pass adds `memcpy` operations given specified source and destination buffer and a DMA component.

5) Memcpy to Launch Pass: This pass changes a `memcpy` operation to `launch` with a block containing `read`s and `write`s.

6) Split Launch Pass: This pass splits the specified launch block at the specified place.

7) Merge Memcpy Launch Pass: This pass merges `memcpy` to the specified `launch` operation. It avoids repetition if the `launch` block accesses the same buffer as the `memcpy`.

8) Reassign Buffer Pass: This pass replaces the uses of a buffer to another buffer. For instance, a SRAM read can be replaced with a register read.

9) Parallel to EQueue Pass: This pass converts Affine dialect's `parallel` to EQueue's `launch` with event dependencies.

10) Lower Extraction Pass: This pass unrolls components denotation in vector form.

VI. Case Study: Systolic array

Systolic array is a widely-used mapping strategy to implement efficient multiplications and additions among matrices [17]. As the communication is limited to neighbor processing elements (PEs), there is no cycle wasted on global communication and address matching. Because of their extremely broad design space of application-specific mapping strategies, memory systems, and PE designs, rapid simulation is critical to effectively exploiting systolic array designs.

In this section, we build and study an EQueue model of a systolic array. We aim to answer these questions about the EQueue dialect for this case study:

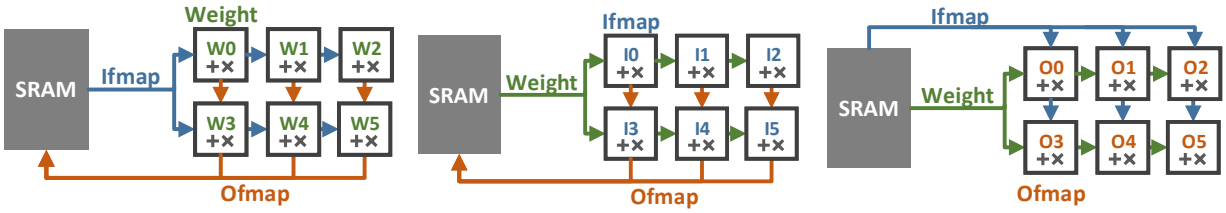
1) Does embedding a simulator into a compiler framework help facilitate exploration of algorithmic mapping options? (Sections VI-B and VI-D)
2) Can the simulation accurately estimate performance? (Section VI-C)
3) Is the simulation useful to help designers find the best design and does it scale? (Section VI-E)

To answer question 1, we first show how to model a systolic array accelerator for convolutions using a *generator* that emits a variety of configurations as EQueue programs. We then also demonstrate a *lowering pipeline* that translates from a high-level MLIR dialect into an EQueue model via a series of reusable compiler passes. For question 2, we compare the EQueue simulation to a state-of-the-art custom simulator. To address question 3, we measure our model to explore a design space of convolution accelerators.

A. Background: Dataflows

A key design decision in a systolic accelerator implementation is the *dataflow*, which determines how loops in the algorithm are mapped spatially onto processing elements (PEs) [3], [36]. In this case study, we consider three widely-used dataflows: Weight Stationary (WS), Input Stationary (IS), and Output Stationary (OS) [36]. The difference is which tensor remains in each PE's register file: the weights, input feature map (ifmap), or output feature map (ofmap).

Fig. 8 illustrates the data movement for each dataflow. On each cycle, each PE computes a part of final result and passes the partial result to its neighbor [22]. We use E_h, E_w for



(a) Weight stationary. (b) Input stationary. (c) Output stationary.

Figure 8: Dataflows mapping on systolic array.

the ofmap height and width, F_h, F_w for the filter height and width, N for number of weights, and C for channels. Fig. 8a shows that for WS, on each cycle, ifmaps and ofmaps are passed to the neighbor PEs, while each weight is stationary until $E_h \times E_w$ ifmaps convolve with it:

```
pe[i+1][j].ofmap = pe[i][j].ofmap + pe[i][j].
    ifmap * pe[i][j].weight
pe[i][j+1].ifmap = pe[i][j].ifmap
```

Fig. 8b shows IS. On each cycle, weights and ofmaps are passed to the neighbor PEs, while every ifmap is stationary until N weights convolve with it:

```
pe[i+1][j].ofmap = pe[i][j].ofmap + pe[i][j].
    ifmap * pe[i][j].weight
pe[i][j+1].weight = pe[i][j].weight
```

Fig. 8c shows OS. On each cycle, ifmaps and weights are passed to the neighbor PEs, while every ofmap is stationary until $F_h \times F_w \times C$ ifmaps and are convolved:

```
pe[i][j].ofmap += pe[i][j].ifmap * pe[i][j].
    weight
pe[i+1][j].ifmap = pe[i][j].ifmap
pe[i][j+1].weight = pe[i][j].weight
```

B. Systolic Array Generator

This section demonstrates a *generator* that emits EQueue code to model systolic array architectures. We start with simple parallelism and build up to the full generator to illustrate the simplicity relative to a traditional, custom simulator.

1) Parallelization: We first show how to construct parallelism using EQueue dialect. We use MLIR's *builder* API, which lets C++ code construct MLIR programs. This pseudo code shows a generator for a simple parallel architecture:

```
start = control_start()
for h in arr_height:
  for w in arr_width:
    done = equeue.launch (...)
      in (start, pe[h][w]){...}
      // assume there is a PE array
    if w ==0 && h==0:
      prev_done = done
    else:
      prev_done = equeue.control_and(done,
          prev_done)
equeue.await(prev_done)
```

The `for` loop iterates over the dimension of the processing element (PE) array (`arr_height` by `arr_width`). Each PE runs in parallel since they are all "launched" by the same `control_start` event. On each loop, `control_and` collects the `launch` events of the current and previous PE. An `await` barrier ensures that the current processor waits for all `launch` events to finish. We later denote this pattern as `par_for`.

2) Systolic passing: We next extend the generator to pass values systolically between PEs. We use two stages: one reads values from a buffer and compute results, and a second stage passes values to neighboring PEs. This generator code shows WS dataflow and omits boundary conditions for simplicity:

```
//reading stage
par_for (h, w) in arr_height-1, arr_width-1:
  done, weight_value[h][w], ofmap_value[h][w]=
    equeue.launch(
    weight_buffer = pe[h][w].weight_buffer,
    ifmap_buffer = pe[h][w].ifmap_buffer,
    ofmap_buffer = pe[h][w].ofmap_buffer)
    in (start, pe[h][w].kernel) {
    ifmap = equeue.read(ifmap_buffer)
    weight = equeue.read(weight_buffer)
    ofmap_old = equeue.read(ofmap_buffer)
    ofmap = ifmap * weight + ofmap_old
    equeue.return weight, ofmap
  }
//writing stage
par_for (h, w) in 1 to arr_height, 1 to arr_width
    :
  done = equeue.launch(
    weight = weight_value[h][w],
    ofmap = ofmap_value[h][w],
    weight_buffer = pe[h][w+1].weight_buffer,
    ofmap_buffer = pe[h+1][w].ofmap_buffer)
    in (start, pe[h][w].kernel) {
    equeue.write(ofmap, ofmap_buffer)
    equeue.write(weight, weight_buffer)
  }
```

In the read stage, each PE reads ifmap, weight and ofmap values from corresponding buffers and computes an ofmap. In the write stage, the PEs in each column (`pe[h][w]`) pass weights to the next column (`pe[h][w+1]`). PEs in a given row write ofmaps to buffers in the next row (`pe[h+1][w]`).

3) Model SRAM Bandwidth: So far, we have a complete and cycle-accurate model of the core PE array logic. The next step is to model the PE array's interaction with associated SRAMs to measure read and write bandwidth.

Extending our EQueue generator, we can change the `launch` input in our read stage to make the first column of PEs read from an SRAM:

```
par_for (h, w) in arr_height, arr_width:
  if w == 0: buffer = sram.ifmap_buffer
  else: buffer = pe[h][w].ifmap_buffer
  done, ... = equeue.launch( ifmap_buffer =
      buffer,
    ...){ ... } // other code same as before
```

Similarly, we can modify the write stage to store ofmaps from the last row of PEs to an SRAM:

```
par_for (h, w) in arr_height, arr_width:
  if w == arr_height-1: obuffer=sram.ofmap_buffer
  else: obuffer=pe[h+1][w].ofmap_buffer
  done = equeue.launch( ofmap_buffer = obuffer,
    ...) {...} // other code same as before
```

With these small changes, the simulation engine can model communication between SRAMs and the PE array.

Benefits. EQueue programs can modularize hardware components (e.g., SRAM interfaces and processors) and thereby study the individual effect of a component. Separating representation from simulation allows a programmer to concentrate on architecture design; no changes are necessary to the simulation engine to evolve the modeled hardware.

C. Comparison with SCALE-Sim

To check the accuracy of our systolic array EQueue model, we compare to a validated simulator SCALE-Sim [32] specific to WS, IS, and OS convolutions on systolic array.

Fig. 9 compares the simulated cycles and average bandwidth for our model and SCALE-Sim, both modeling a 4×4 WS systolic array with various ifmap and weight sizes. Our EQueue-based simulation matches SCALE-Sim's results.

Benefits and Costs. When exploring design alternatives, an EQueue-based simulator has a lower programming cost than a custom one-off simulator. SCALE-Sim [32]'s WS and IS implementation have little code overlap: WS is implemented in Python in 569 lines of code (LOC), but switching from WS to IS requires changing 410 LOC. In contrast, our EQueue program for WS is implemented in C++ in 281 LOC only needs 11 LOC to switch from WS to IS.

In exchange, the one-off simulator has a performance advantage: for experiments in Fig. 9, SCALE-Sim takes at most 1.1 second, while the EQueue simulator takes at most 7.2 seconds. The speed comes at the cost of complex modifications while exploring the architectures and algorithm mappings.

D. Lowering Pipeline

Rationale. The benefit of a compiler-driven approach is not limited to lowering the bar of programming: more importantly, it makes it possible to program the simulator using compiler passes. Integrating with a compiler stack's shared passes avoids the need for tedious, manual modification to explore different program mappings.

This section constructs a *lowering pipeline* that compiles from high-level algorithmic specifications to EQueue hardware models. Critically, the pipeline can produce different dataflows for the same input program by applying different sequences of compiler passes.

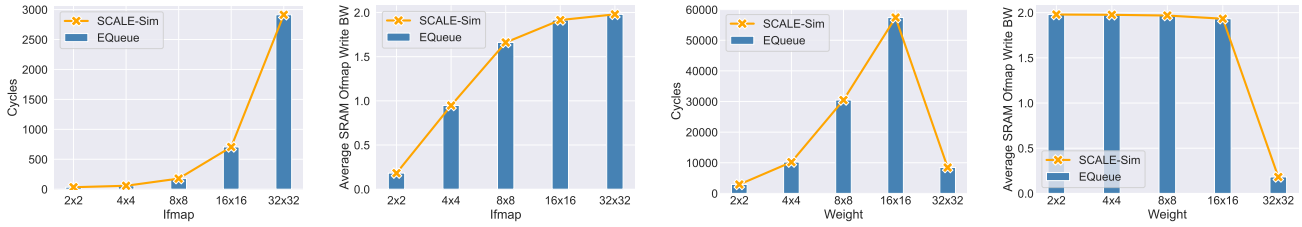
Implementation. IS, WS, and OS all share a core systolic design: on each cycle, each PE reads a value, modify it, and writes to a neighbor PE. Fig. 10 shows how the systolic dataflows share stages along a lowering pipeline. The first 3 stages (Linalg, Affine, and Reassign) are the same. The final stage (Systolic) diverges, but lowering from Reassign stage allows different dataflows to share lowering passes with different orders and parameters. This way, hardware designers can only implement the highest level abstraction and then explore design spaces with no programming overhead.

1) Linalg to Affine: We start with a convolution in Linalg dialect, an MLIR dialect that can express arbitrary linear algebra. The Linalg dialect can be first lowered to the Affine dialect with the standard `--convert-linalg-to-affine-loops`, which lowers the convolution to explicit nested loops. We then apply `--equeue-read-write` to change `load` and `store` operations in Affine dialect to `read` and `write` in EQueue dialect to model data movement.

2) Affine to Buffer Reassign: Next, we apply the `--allocate-buffer` and `--reassign-buffer` passes to replace direct SRAM reads and writes with PE local register accesses. At this stage, we also flatten the 6 convolutional dimensions (E_h, E_w, N, F_h, F_w, C) into 3: $E_h \times E_w$, N, $F_h \times F_w * C$. This flattening reflects the stationary dimension on PEs for each dataflow: for WS, each weight is stationary on a PE until computed with $E_h \times E_w$ ifmaps; for IS, each ifmap is stationary for N weights; for OS, each ofmap is stationary until accumulated with $F_h \times F_w \times C$ ifmaps and weights.

3) Buffer Reassign to Systolic Array: After flattening, for WS and IS, we first need to copy weights or ifmaps from the SRAM into the PE array registers. We generate the necessary `memcpy` operations with a `--mem-copy` pass and merge them with `launch` operations using `--merge-memcpy-launch` . Then, we implement systolic communication. For WS, we need to pass the ifmaps and ofmaps to the right and down on every cycle. Similarly, for IS, N weights and ofmaps are passed, while for OS, $F_h \times F_w * C$ ifmaps and weights are passed. The `--split-launch` and `--reassign-buffer` passes implement this systolic communication. Finally, we apply `--parallel-to-equeue` and `--lower-extraction` passes to complete lower operations to EQueue dialect.

The key advantage of the lowering pipeline approach is the reduced effort for implementing different dataflows. In a traditional simulator, changing the mapping strategy requires extensive rewriting of the simulation engine. In the compiler-driven approach, designers can apply different combinations



(a) Cycles. (b) SRAM write BW. (c) Cycles. (d) SRAM write BW.

Figure 9: Comparing our simulation with SCALE-Sim on 4×4 systolic array, by convolving various ifmaps with fixed $2 \times 2 \times 3$ weights (a–b) and by convolving various weights convolved with fixed 32×32 weights (c–d).

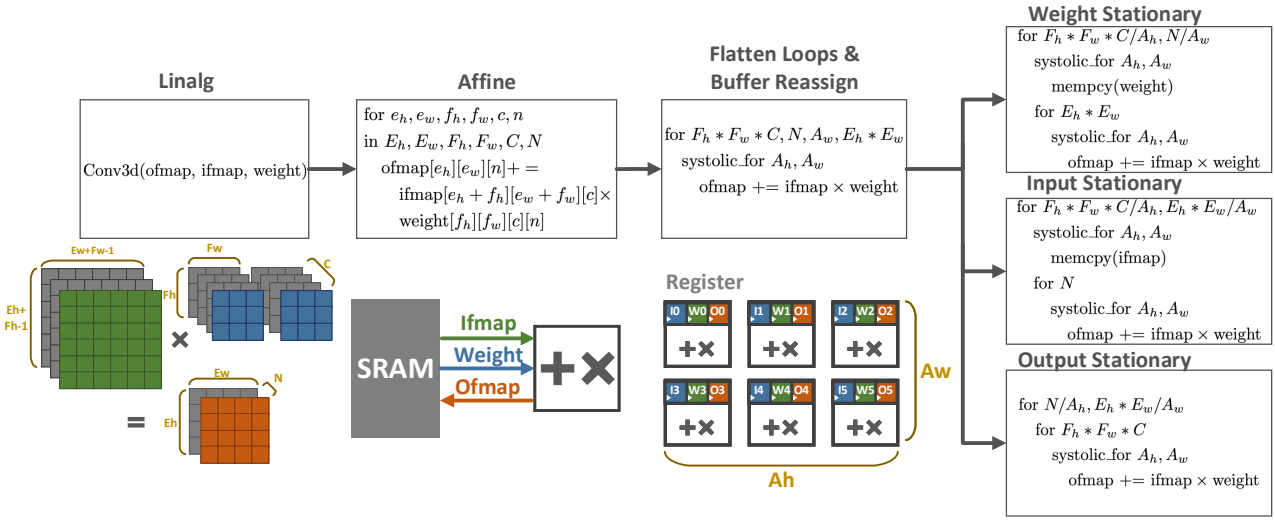


Figure 10: Lowering pipeline for WS, IS, OS dataflow. They share the same lowering stages except the last one.

of reusable passes to try out different dataflows.

Results. Fig. 11 plots the simulator execution time, simulated runtime, and read and write bandwidth on the four convolution settings at the four stages (Linalg, Affine, Reassign, and Systolic). This compiler pipeline does not take a significant amount of time (it typically finishes in microseconds).

The first three lowering stages are identical for different dataflows, so they have the same bandwidth and runtime. This sharing reflects the framework's reusability for common parts of different accelerator implementations. At the final stage, the runtime differs from the simpler generator-based approach from Section VI-B by 1.2% on average, up to 2%. The difference lies in warm-up and cool-down phases that the passes do not model. Register and SRAM bandwidth differs for the same reason.

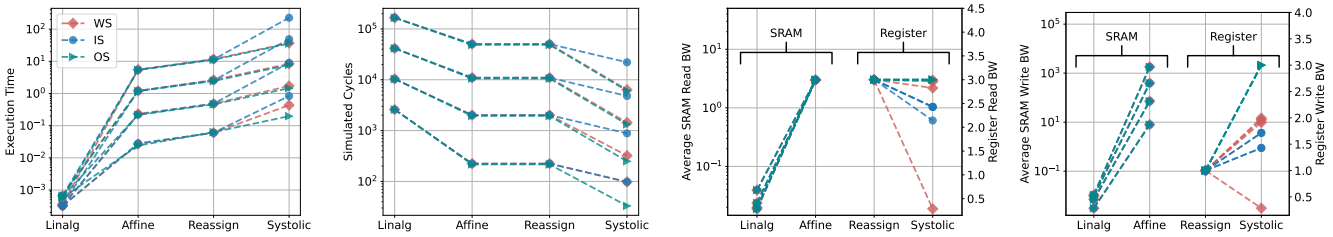
Fig. 11 also reflects the transition of hardware at each stage. From Linalg stage to Affine stage, the execution time grows, the runtime reduces, and the SRAM bandwidths grow, since affine stage models explicit nested loops and data movements. At the Reassign stage, we model reads and writes on registers rather than SRAM, so the register bandwidth changes from 0 to 1 byte per cycle and the execution time grows. At systolic stage, we introduce a grid of PEs running concurrently, resulting in higher execution time, lower runtime in cycles and differentiated bandwidth.

Benefits. The availability of reusable lowering passes lets designers rapidly switch between program–accelerator mappings and enables efficient design space exploration. In contrast, one-off simulators would require custom modifications to support these transformations.

E. Scalability Evaluation

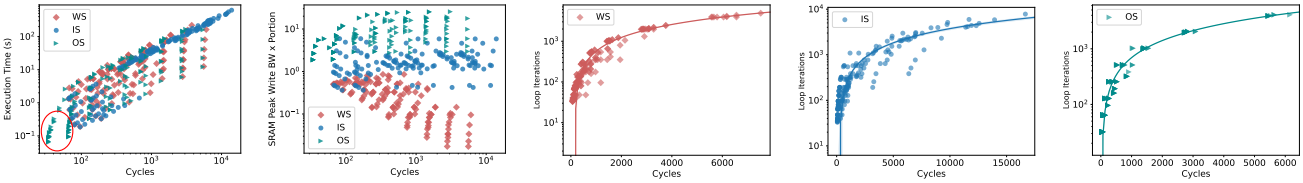
For hardware designers, information on different dataflow performance patterns is essential when designing new mapping strategies. To test our simulator's generality and scalability, we measure runtime and bandwidth for 4,050 combinations of array configuration ($A_h = 2, 4, 8, 16, 32, A_w = 64/A_h$) and convolutions ($H/W = 2, 4, 8, 16, 32$, $F_h/F_w/C = 1, 2, 4$, $N = 1, 2, 4, 8, 16, 32$) on the three dataflows.

Simulator scalability. Fig. 12a plots the simulator execution time versus the cycle counts for each simulation. The execution time is roughly proportional to the cycle count, since



(a) Execution time for simulation. (b) Runtime in cycles. (c) Read bandwidth. (d) Write bandwidth.

Figure 11: Figure showing various metrics along the four stages of lowering pipeline on an accelerator with a 4×4 PE array and convolution settings $H = W = 4, 8, 16, 32, F_h = F_w = 3, C = 3, N = 4$. Compilation time is negligible and omitted.



(a) Execution time. (b) Peak read bandwidth. (c) Loop iteration for WS. (d) Loop iteration for IS. (e) Loop iteration for OS.

Figure 12: Given various convolution and array configuration, plotting different parameters versus cycles for three dataflows.

our simulator faithfully reflects behavior of each processor. At most, our simulator may require over 10 minutes for simulation. Future work could reduce execution time by building a lookup table to skip duplicated behavior or by adding parallelism to the simulation engine.

Dataflows. Fig. 12b plots the SRAM read bandwidth at peak (the maximum bandwidth times the duration) versus cycle time. OS has the highest read bandwidth overhead while WS requires the least. Though Fig. 12a highlights OS can achieve the shortest runtime in cycles, designers can choose the dataflow according to hardware requirements.

Array configuration. Our simulator can help designers observe general "rules" about performance. Figures 12c to 12e plot the relationship between cycles and array structures. The loop iteration count is proportional to cycle count. We can calculate loop iterations as $\lceil D1/A_h \rceil \times \lceil D_2/A_w \rceil$, where $D_1 = F_H \cdot F_w \cdot C, D2 = N$ for WS, $D_1 = F_H \cdot F_w \cdot C, D_2 = E_h * E_w$ for IS and $D1 = N, D_2 = F_H \cdot F_w \cdot C$ for OS. With this general rule, we can always get the minimal execution time by choosing the array structure that minimizes loop iterations.

Benefits. The evaluation on 4050 data points shows that our simulator scales to various convolutions. Algorithm designers can use it to choose the best dataflows and array configuration for a convolution.

VII. Case Study: ACAP AI Engine

A common approach to hardware–software co-design is to start simple and, guided by bottlenecks, build up a more sophisticated architecture. This section uses the EQueue dialect to simulate a real-world architecture: Xilinx's AI Engine in Versal ACAP [35]. We show our simulation result matches the AI Engine simulator [38], while the high-level simulator allows architectures ignore real-word constraints like bandwidth and gradually introduce them with low programming cost. During this process, the EQueue visualized tracing can guide designers to improve their designs.

A. Versal ACAP

Xilinx's Versal adaptive compute acceleration platform (ACAP) is a reconfigurable platform that includes programmable logic, ARM cores, and *AI Engines*, which are specialized vector units [8], [35], [40]. The AI Engine is a fixed array of interconnected VLIW SIMD processors optimized for signal processing and machine learning.

B. FIR

A finite impulse response (FIR) filter is a common signal processing primitive that responds to inputs of finite duration. An FIR operation filters and accumulates on a sliding-window. Given a series of discrete input samples x and N coefficients c, the output samples y are calculated as:

$$y_n = \sum_{k=0}^{N-1} c_k \cdot x_{n+k}$$

Xilinx's AI Engine programming tutorial [38] uses a FIR filter as an example to demonstrate the hardware's flexibility and capabilities. In this case study, we implement the same FIR example using the EQueue dialect dialect to demonstrate how the language and simulation engine can easily model an existing programmable architecture. We compare our

simulator's reports to those from Xilinx's own, hand-written, closed-source simulator to ground the results.

The Xilinx FIR tutorial uses a filter with 32 complex, asymmetric coefficients and a digital series of length 512. Each value occupies 32 bits.

C. Case 1: Unlimited Resources

We start with a basic 1-processor implementation and use empirical measurements to improve the design. We can use the AI Engine's intrinsics: `mul4` and `mac4`. On each cycle, `mul4` computes on 4 parallel lanes to perform 8 multiplications where each lane performs 2 [39]. `mac4` works in the same way. Analytically, therefore, it should take 16 cycles to compute 4 outputs for a filter length of 32.

We follow Section III-E to self-define `mul4` with `equeue.op("mul4", {ofmap, ifmap, filter})`. In the simulator library, an operation with the "mul4" signature reads from a buffer, computes 4 lanes with 2 computation at each lane per cycle, and writes to the buffer. We define the `mac4` operation the same way. This pseudocode shows the MLIR generator for a single-core implementation, where `ifmap`, `ofmap` and `filter` are buffers:

```
start = equeue.control_start()
equeue.launch(...) in (start, ai_engine){
    equeue.op("mul4", {ofmap, ifmap, filter})
    for 0 to 11:
        equeue.op("mac4", {ofmap, ifmap, filter})
    ifmap_tensor = equeue.read(sin)
    equeue.write(ifmap_tensor, ifmap)
    for 0 to 4:
        equeue.op("mac4", {ofmap, ifmap, filter})
    ofmap_tensor = equeue.read(ofmap);
    equeue.write(ofmap_tensor, sout)
}
```

Our EQueue simulation reports 2048 cycles to generate 512 outputs, close to Xilinx AI Engine simulator's result of 2276 cycles [38]. The Xilinx simulator also models other factors in performance, including loop control costs, synchronization overhead, etc. The EQueue simulation engine's throughput is slightly higher because it does not model these overheads.

D. Case 2: Optimizing Case 1

The next step for a hardware designer is to incrementally increase the design's complexity to attain higher throughput. In an ideal world, since `mul4/mac4` computes 4 lanes, each with 2 operation per cycle, we could pipeline $32/2 = 16$ processors to maximize throughput. Due to bandwidth constraints, Xilinx's FIR tutorial simulates 4 processors rather than 16. Using the EQueue model, we can first model the full 16-processor pipelined system and then introduce more realistic constraints to measure their effect on performance.

The modification to our EQueue program is straightforward. Instead of one processor executing 16 sequential operations, we now create 16 processors, where each processor completes one `mul4`/`mac4` operation. We show the simplified control flow:

Figure 13: Visualizing operation-wise tracing of FIR implemented with 16 processors with limited bandwidth.

```
start = equeue.control_start()
for k in 0..16:
    equeue.launch(...) in (start, ai_engine[k]){
        ifmap_tensor = equeue.read(sin)
        equeue.write(ifmap_tensor, ifmap)
        equeue.op("mac4", {ofmap, ifmap, filter})
        ofmap_tensor = equeue.read(ofmap);
        equeue.write(ofmap_tensor, sout)
    }
equeue.await()
```

The simulation engine reports 143 cycles to produce outputs for 512 inputs. This matches the expected performance because pipelining 16 processors requires 15 cycles to warm up.

E. Case 3: Limited Bandwidth

The AI Engine is constrained by the 32-bit bandwidth of its AXI4-Stream I/O interfaces [39]. To add bandwidth constraints, we need only add a connection (Section III-A) and update the reads and writes accordingly:

```
conn_in = connection("Streaming", 32);
conn_out = connection("Streaming", 32);
...
ifmap_tensor = equeue.read(sin, conn_out)
equeue.write(ofmap_tensor, sout, conn_out)
```

Adding this bandwidth constraint entails only simple, local changes to the EQueue program; extending a custom simulator, in contrast, could require invasive modifications. According to our simulation engine, it takes 588 cycles to generate 512 outputs, including 79 cycles to warm up.

To understand the reason for reduced throughput, Fig. 13 shows the operation-wise tracing with visualized via the Chrome browser, where green slots are `mul` operations, red slots are `mac` operations, blue slots indicate installing and the x-axis shows cycle counts where $1\mu s$ stands for one cycle. For every 4 cycles, each processor operation stalls for 3 cycles. The stalls are the result of the 32-bit bandwidth constraint: it takes 4 cycles to transmit 4 inputs, but computation only takes 1 cycle to consume these values. For each AI Engine's attempt to start computation, it waits for its preceding core compute (1 cycle) and pass values to it (4 cycles), so the warm-up stage takes $5 \times 16 - 1 = 79$ cycles.

F. Case 4: Optimizing Case 3

Our bandwidth-constrained model shows that 75% of the hardware's computation power is wasted, i.e., we stall on 3

Figure 14: Visualizing operation-wise tracing of FIR implemented with 4 processors with limited bandwidth.

of every 4 cycles. To balance the system and avoid wasting area and power, a designer can reduce the 16 processors to 4:

```
start = equeue.control_start()
for k in 0..4: // 16 -> 4 cores
    // 1 -> 4 sequential computations
    equeue.launch(...) in (start, ai_engine[k]){
        ifmap_tensor = equeue.read(sin,
            connection_in[k])
        equeue.read(ifmap_tensor, ifmap)
        equeue.op("mac4", {ofmap, ifmap, filter})
        equeue.op("mac4", {ofmap, ifmap, filter})
        ofmap_tensor = equeue.read(ofmap);
        equeue.write(ofmap_tensor, sout,
            connection_out[k])
        equeue.op("mac4", {ofmap, ifmap, filter})
        equeue.op("mac4", {ofmap, ifmap, filter})
    }
equeue.await()
```

Our EQueue simulation engine reports that generating 512 outputs requires 538 cycles, which matches Xilinx's simulator result of 539 cycles. Warm-up takes 26 cycles, which is much faster than the previous case. Fig. 14 visualizes the operation trace for the balanced 4-processor system: there is no stalling once the processors have warmed up.

Benefits. Typical simulation tools can make it challenging for software designers to identify hardware bottlenecks. This case study advocates an opposite approach: designers can start with a simple design and gradually add real-world constraints to examine their effects on performance. Our EQueue approach requires modest modification at each step, but effectively guides users to improve their design.

Our EQueue-based approach matches the results of Xilinx's existing AI Engine simulator tool. Thanks to its high-level abstraction, the EQueue-based simulator is much faster: the 4 processor implementation takes 0.07 seconds, while the AIE simulator first requires 5 minutes for compilation and then 3 minutes for simulation. Also, due to its focus on low-level details, the AI Engine implementation is spread across six separate files. Any updates to the interface or mapping strategy requires substantial work to implement and recompile.

VIII. Related Work

Because simulation is a critical part of a hardware design workflow, it is an old and well-studied research topic. Space precludes a complete census of all approaches to simulation, but we discuss the most closely related techniques here.

The EQueue simulation flow takes inspiration from hardware modeling languages [1], [11], [12], [19], [21], [29], [31]. It differs from RTL simulation with its higher-level representation and focus on an intermediate representation that can be transformed by compiler passes. We designed the EQueue dialect because existing languages and MLIR dialects cannot represent the core concepts required for flexible, high-level simulation: fine-grained concurrency, contention for shared heterogeneous hardware resources, and data movement constraints.

RTL simulators: Most RTL development tools have accompanying simulators [13], [15], [16], [41], [42]. RTL simulation can faithfully model a complete hardware design, but implementing a design in RTL requires specialized hardware expertise and carries a high engineering burden. An alternative is integrating a more abstract simulation with RTL using a multi-level tool such as PyMTL [23]. We view the EQueue dialect as a complement to these frameworks that makes it easier to generate and transform higher-level models before completing a more detailed implementation.

Application-specific simulators: Many efforts have constructed architecture-specific simulators, for domains including sparse linear algebra [4], stencils [6], and DNN inference [5], [18], [26], [32]. While these simulators are fast and accurate, they are challenging to construct from scratch. The EQueue dialect dialect provides a faster way to build simulators.

MLIR methodology: It is appealing to use existing MLIR dialects that already offers various transformations and ways to express computations and hardware. For example, MLIR's *Async* dialect [25] models asynchronous execution. It cannot, however, associate code with specific processing units in a hardware structure. CIRCT [7] is an ongoing project to apply MLIR's methodology to hardware design tools. It encompasses many dialects, including a *Handshake* dialect representing asynchronous processes that can compile to a *FIRRTL* dialect for circuit-level transformations and then to *LLHD* dialect to describe RTL. The *HIR* dialect [24] describes hardware with explicit scheduling and binding, which serves as a better IR than pure LLVM for HLS-like compilation from software to hardware. Both HIR and the CIRCT dialects are abstractions for generating concrete hardware implementations, not high-level abstractions for modeling concurrency and data movement for efficient simulation. The EQueue dialect differs by explicitly representing execution units and mapping event-triggered computations onto them.

Compiler-driven DSE: Compilation is an efficient way to perform design space exploration (DSE), especially in the specific domain of tensor computations. Interstellar [43] uses Halide [30] to explore DNN accelerator designs. Union [14] uses MLIR programs as inputs to optimize spatial DNN accelerators by analyzing tensor operations expressed in Linalg or Affine dialect with MAESTRO [18] and Timeloop [26] as cost models. Similar to the EQueue methodology, these frameworks benefit from separating modeling from representation for rapid iteration. However, all of these approaches

target a specific category of computation and hardware: they map high-level DNN dataflow mappings to synchronous PE arrays of regular structures. The EQueue dialect aims to address *general* hardware simulation, including programmable architectures that do not resemble systolic arrays, such as the AI Engine (Section VII). EQueue may also be a good fit for extending Union with support for explicit representations of hardware components.

IX. Conclusion

Hardware simulation frameworks need a separation of simulation from representation. The simulation flow for EQueue programs lowers the bar for designers with abstract representations, exposes intermediate optimization stages, and makes it easy to apply changes with reusable compiler passes.

Acknowledgements

This work was supported by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research is also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). It was supported by NSF awards #1845952 and #1723715.

References

[1] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 89–108.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[3] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[4] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[5] S. B. Choi, S. S. Lee, and S. J. Jang, "CNN inference simulator for accurate and efficient accelerator design," in *2019 International SoC Design Conference (ISOCC)*, 2019, pp. 283–284.

[6] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[7] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder *et al.*, "MLIR as hardware compiler infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[8] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versal architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[9] Google, "Trace event format," https://github.com/catapult-project/catapult/blob/master/tracing/docs/getting-started.md, accessed: 4-29-2021.

[10] Google, "The trace event profiling tool," http://dev.chromium.org/developers/how-tos/trace-event-profiling-tool, accessed: 5-28-2021.

[11] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines." *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[12] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.

[13] Intel Inc., "Intel software development emulator," https://software.intel.com/content/www/us/en/develop/articles/intel-software-development-emulator.html, accessed: 5-8-2021.

[14] G. Jeong, G. Kestor, P. Chatarasi, A. Parashar, P.-A. Tsai, S. Rajamanickam, R. Gioiosa, and T. Krishna, "Union: A unified HW-SW co-design ecosystem in MLIR for evaluating tensor operations on spatial accelerators," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 30–44.

[15] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanovic, "Evaluation of RISC-V RTL with FPGA-accelerated simulation," in *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[16] D. Kim, A. Izraelevitz, C. Celio, H. Kim, B. Zimmer, Y. Lee, J. Bachrach, and K. Asanovicc, "Strober: Fast and accurate sample-based energy simulation for arbitrary RTL," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 128–139.

[17] H. Kung, "Why systolic architectures?" *IEEE Annals of the History of Computing*, vol. 15, no. 01, pp. 37–46, 1982.

[18] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[19] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 242–251.

[20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of Moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[21] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From image processing DSL to efficient FPGA acceleration," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 51–57.

[22] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proceedings of the Chapel Hill Conference on VLSI*. Chapel Hill NC, 1985, pp. 363–376.

[23] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 280–292.

[24] K. Majumder and U. Bondhugula, "HIR: An mlir-based intermediate representation for hardware accelerator description," *arXiv preprint arXiv:2103.00194*, 2021.

[25] MLIR Project, "Async dialect," https://mlir.llvm.org/docs/Dialects/AsyncDialect/, accessed: 5-11-2021.

[26] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[27] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2014.

[28] C. Ptolemaeus, *System design, modeling, and simulation: using Ptolemy II*. Ptolemy.org, Berkeley, 2014, vol. 1.

[29] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.

[30] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[31] H. Rong, "Programmatic control of a compiler for generating high-performance spatial hardware," *arXiv preprint arXiv:1711.07606*, 2017.

[32] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[33] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A multi-level intermediate representation for hardware description languages," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 258–271.

[34] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[35] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-chip programmable platform in Versal ACAP architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 212–221.

[36] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[37] A. Varga, "Discrete event simulation system," in *Proc. of the European Simulation Multiconference (ESM'2001)*, 2001, pp. 1–7.

[38] Xilinx Inc., "Super sampling rate FIR filters implementation on the AI Engine." https://github.com/Xilinx/Vitis-Tutorials/tree/master/AI_Engine_Development/Design_Tutorials/02-super_sampling_rate_fir, accessed: 7-21-2021.

[39] Xilinx Inc., "Versal ACAP AI Engine programming environment - user guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1076-ai-engine-environment.pdf, accessed: 4-28-2021.

[40] Xilinx Inc., "Versal: The first adaptive compute acceleration platform (ACAP)," https://www.xilinx.com/support/documentation/whitepapers/wp505-versal-acap.pdf, accessed: 5-1-2021.

[41] Xilinx Inc., "Vivado high-level synthesis," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed: 5-8-2021.

[42] Xilinx Inc., "Vivado simulator," https://www.xilinx.com/products/design-tools/vivado/simulator.html, accessed: 5-8-2021.

[43] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina *et al.*, "Interstellar: using Halide's scheduling language to analyze DNN accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.