Contributions to Aetherling

David Akeley

September 9, 2018

Abstract

This is a list of my contributions to the Aetherling project. I start with a section summarizing tasks I worked on, then I expand on the tasks in further sections.

1 Summary

1. Line Buffer Specifications

   I proposed a new specification of Aetherling's line buffer node and wrote a document ("The Line Buffer Manifesto") describing the benefits of this redesign. The previous line buffer design was hard to parallelize due to difficult-to-satisfy constraints on its parameters' divisibility, and also did not support downsampling ("stride"). The redesign addresses these issues.

2. Functional Simulator

   I wrote a functional simulator for Aetherling, which includes a simulation of the intended behavior of the redesigned line buffer. This allows the user to test the functionality of an Aetherling DAG using pure Haskell code.

3. Demonstration Apps

   I designed two demonstration Aetherling DAGs: Gaussian blur (7×7 stencil) and mipmap generation. These DAGs can be tested using the functional simulator along with a library (`ImageIO.hs`) for converting between simulator values and png images on disk. These apps also demonstrate the applications of the redesigned line buffer.

4. Simplifying Ops

   An op is a node of an Aetherling DAG (along with its children, sometimes). Previously there were multiple ways to express the same functionality. I simplified the ops to minimize redundancy.

5. Helper Functions

   I wrote some Haskell helper functions for creating ops and simple patterns of ops. These helpers check for invalid parameters and substitute for functionality lost through the op simplification.

6. Ready-Valid Meta-Op

   By default, Aetherling pipelines are composed of ops representing circuits with synchronous timing: they wait for a certain number of warm-up cycles, then emit outputs on a repeating schedule (phase). I designed a `ReadyValid` op that represents the idea of wrapping a portion of an Aetherling DAG with a ready-valid interface. I modified the compose operators (`|&|` and `|>>=|`) to properly handle the ‘ReadyValid‘ op. The user is prevented from composing an op with ready-valid timing with one with synchronous timing, and, when composing two ready-valid ops in sequence, the throughput-matching behavior of the Aetherling type system is suppressed.

7. ComposePar Retiming

   Aetherling includes a `ComposePar` op that represents placing circuits (child ops) in parallel. The user is not required to ensure that each parallel path has the same latency (sequential latency – the count of the number of register delays along a path). I wrote a pass that walks an Aetherling DAG searching for `ComposePar` ops, modifying its child ops if needed such that all paths have the same latency. The pass finds an optimal solution that minimizes the number of register bits added to the circuit.

8. Fractional Underutilization and Phase

   In the Aetherling team, “phase” refers to the repeating pattern of valid and garbage values input to/generated by a synchronously timed op. For example, an op that generates 1 valid output every 3 cycles would have an output phase of `[True, False, False]` (Two out of every three cycles, the op generates garbage).

   The choice of phase for an op with integer underutilization (1 valid input/output per X clock cycles) is obvious, but with fractional underutilization (X valid per Y clocks), there are several reasonable phase choices. When several such underutilized ops are joined together, there’s no reason to assume that their phase patterns will match.

   Since the Aetherling type system only exposes type and throughput information to the user, it’s vital that the system take care of phase matching automatically. I proposed a scheme that assigns each fractional utilization ratio a standardized phase. (The earlier phase corresponds to $\frac{1}{3}$ utilization). This allows the complexity of phase matching to be confined to one op in the system, `SequenceArrayRepack`. Along with the ComposePar retiming pass, this makes it so that users only have to be concerned with type and throughput matching, allowing them to view phase and latency as performance, rather than correctness, issues.

9. Tests Written

   I gained a lot of experience writing tests as part of my work on the Aetherling project. These tests include tests for the functional simulator, tests for the hardware line buffer David Durst is designing, tests for the compose operators (`|&|` and `|>>=|`), and tests for the `ComposePar` retiming passes.

2 Line Buffer Specifications

3 Functional Simulator

4 Demonstration Apps

5 Simplifying Ops & Helper Functions

Previously, many Aetherling arithmetic ops (e.g. `Add`) took a type parameter, which could be an array type. This meant that there were multiple ways to express the same operation. For example, a bit xor could be expressed as `XOr T_Bit` or `Add T_Bit`, and elementwise addition of two 4-arrays-of-int could be expressed as `Add $ T_Array 4 T_Int` (4-array type parameter) or as a `MapOp 4` over a scalar `Add T_Int`.

Since each op of the Aetherling Haskell IR will eventually need to be implemented in hardware, it would be best to minimize unneeded functionality as much as possible. I eliminated the type parameter from arithmetic ops, splitting the op into two ops if bit and integer versions are both needed (e.g. 'And' for boolean *and*; 'AndInts' for bitwise *and*).

Since this change makes vectorized arithmetic harder to express directly, I wrote some helper functions for expressing array types and array operations. For example, a 16×16 matrix of integers can be expressed as `tInts[16,16]`, and an op performing elementwise addition of two matrices can be expressed as `addInts $ tInts[16,16]`. Internally, the IR represents vectorized ops as scalar ops wrapped by `MapOp`. In this example, `addInts` will return

```
MapOp 16 (MapOp 16 Add)
```

The helper functions also serve the purpose of checking for invalid parameters. For example, the function for creating a line buffer op[1] checks that the divisibility requirements are satisfied, and the `arrayReshape` function, which creates an `ArrayReshape` op that reinterprets inputs as a different type, checks that there's a reasonable mapping between input and output types.[2]

6 ReadyValid Op

7 ComposePar Retiming

8 Fractional Underutilization and Phase

Originally Aetherling only supported the concept of integer underutilization. After waiting for a certain number of warmup cycles, synchronous ops could accept inputs or create outputs on a repeating schedule of 1 valid input/output every `N` clock cycles, where `N` is an integer.[3]

The problem is that we wanted to lift the integer restriction on underutilization and allow Aetherling ops to have any fractional throughput (between 0 and 1) ("fractional underutilization"). The benefit of the original integer underutilization scheme is that two ops are guaranteed to work together when composed in sequence given that their linked ports have the same throughput ($\frac{1}{N}$). As long as the downstream op waits for L cycles before accepting the first input, where L is the latency in clock cycles of the upstream op, the timings of the two ops will match, with the upstream op feeding input to the downstream op on cycles L, $L+N$, $L+2N$, $L+3N$...

To maintain this simplicity with fractional underutilization, I proposed that each fractional throughput should be assigned one standardized phase pattern (repeating schedule of valid and garbage inputs/outputs). The details of this assignment algorithm are not important yet. The important points of this plan are that

1. Each op has one `sequentialLatency` value – the difference in clock cycles between the time when its first input arrives and its first output is produced.[4]

[1] Temporarily named `manifestoLineBuffer`.

[2] Examples: `arrayReshape [tBits[2]] [T_Bit, T_Bit]` (conversion of 2-array-of-bit to two separate bits) is valid, while `arrayReshape [tBits[3]] [T_Bit]` is not since the input has more bits than the output.

[3] Strictly speaking, Aetherling supported reciprocal-integer throughputs, which I refer to as integer underutilization since most ops can only support such throughputs through underutilization.

[4] This is still well-defined if the op has multiple ports with varying throughputs. Each phase pattern starts with a valid input/output, so the initial batch of inputs are synchronized across all ports. Same with the initial batch of outputs.

2. Each port of an op is assigned a phase pattern based on its throughput. (In Aetherling, this throughput is calculated by dividing `seqLen`, a property of the port, with `cps` (clocks per sequence), a property of the entire op).

3. Each output port of an op, in isolation, behaves like this: it emits garbage for `sequentialLatency` cycles, then cycles through its assigned phase pattern of valid and garbage outputs. (If the op is not in isolation, we'll have to wait additional cycles equal to the wait time for the first input to arrive – this is equivalent to the sum of the upstream ops' `sequentialLatency`s). Note that a consequence of the earlier definition of `sequentialLatency` as the difference in time between the first input and first output, the phase pattern must start with a valid output.

With these standardized phases, given that we properly delay downstream ops, we can be certain that two ports' schedules will match if their fractional throughputs match (checked by the Aetherling type system), just as the case was for the earlier system that only supported integer-reciprocal throughputs.

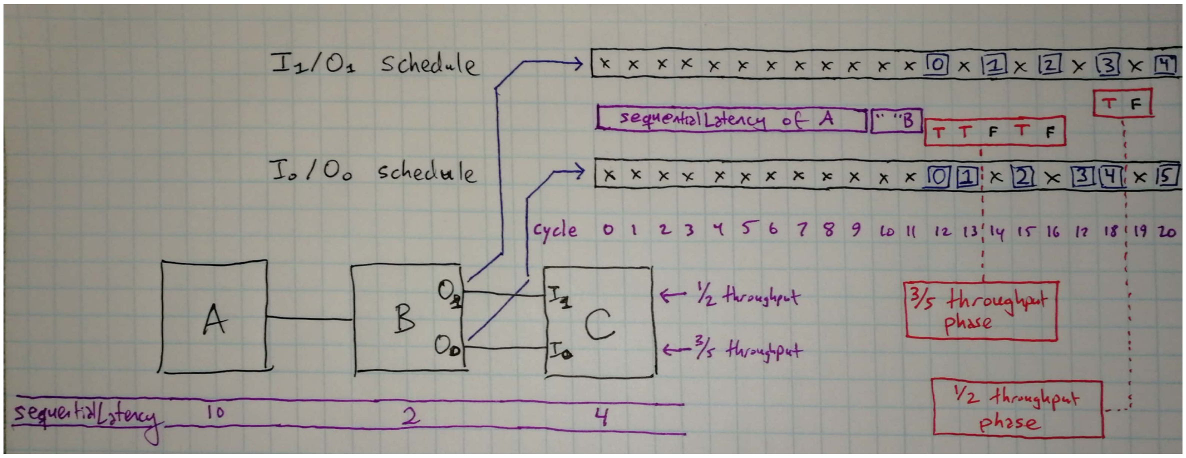
8.1 Pipeline Example

Consider the example pipeline `A |>>=| B |>>=| C` (`|>>=|` means sequential compose, i.e. wire the outputs of left op to inputs of right op). Suppose that `A` has `sequentialLatency` 10, `B` has `sequentialLatency` 2, `C` has `sequentialLatency` 4, `B` has two output ports O_0 and O_1 with throughputs $\frac{3}{5}$ and $\frac{1}{2}$ respectively, which matches input ports I_0, I_1 of `C`. The standard phase for $\frac{3}{5}$ throughput is `[True, True, False, True, False]`, and for $\frac{1}{2}$, `[True, False]`. (`True` denotes a clock cycle with valid input/output; `False` denotes garbage).

Consider the output pattern of O_0. We'll start with 12 cycles of garbage, since 12 is the sum of `A` and `B`'s `sequentialLatency`s, then it'll create valid outputs on cycles 12, 13, 15, 17, 18, 20, 22... (since the $\frac{3}{5}$ phase indicates the $0^{th}, 1^{st}, and 3^{rd}$ cycles in each 5-cycle pattern is valid). Meanwhile, valid output will be sent through O_1 on cycles 12, 14, 16, 18, 20, 22...

This matches what `C`'s input ports I_0 and I_1 expect. They'll wait 12 cycles since 12 is the sum of the upstream ops' `sequentialLatency`s, then, by looking up the same phase pattern, `C` can reproduce the exact schedule that `B` used. (I_0 expects input on cycles 12, 13, 15, 17...; I_1 on cycles 12, 14, 16...).

Finally, note that the initial inputs and initial outputs are always synchronized across ports. O_0 and O_1 start output on cycle 12, and I_0 and I_1 start expecting input on cycle 12. This means that the definition of `sequentialLatency` is well-defined. Using the definition, we expect that each input port of `B` should start accepting input on cycle 10, and each output port of `C` should emit its first valid output on cycle 16.

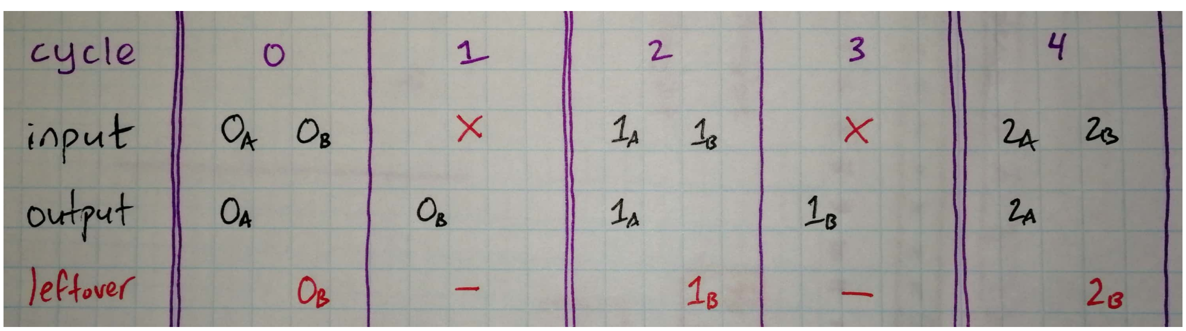


Visualization of scheduling based on `sequentialLatency` wait and repeating phase pattern.

8.2 Phase Assignment Algorithm

The throughput-to-phase assignment algorithm is based on the `SequenceArrayRepack` op. This op converts between sequences of arrays of different sizes. (In Aetherling terms, a sequence is a stream of valid values delivered across multiple, not-necessarily-contiguous clock cycles – in the earlier example, port O_0 delivers a 3-sequence of outputs on cycles 12, 13, and 15).

For example, a (fully utilized) `SequenceArrayRepack` that converts from 1-sequences of 2-arrays to 2-sequences of 1-arrays would take a valid 2-array input on cycles 0, 2, 4... and, using values unpacked from the input arrays, generate a 1-array output on each clock cycle.



`SequenceArrayRepack` repacking 2-arrays to 1-arrays. Each sequence gets its own digit.

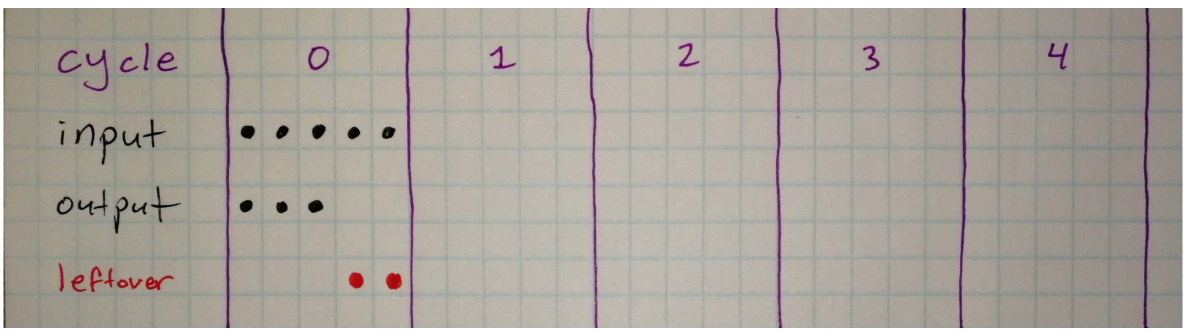
To determine the pattern for an $\frac{x}{Y}$ throughput, we figure out what pattern would be most convenient for the input of a fully-utilized `SequenceArrayRepack` converting x-sequences of Y-arrays to Y-sequences of x-arrays. (This is a "narrowing" repack:

since

$$\text{throughput} < 1 \implies x < Y$$

the output arrays are narrower than the input arrays). The repack is fully utilized, so the output comes out at maximum rate: X outputs over X clock cycles (in Aetherling terms, `cps=X`).

It would be most convenient for the `SequenceArrayRepack` if each input array came in as soon as it was needed, but no sooner. (This is convenient in the sense that it minimizes internal buffering). Earlier, I set the $\frac{3}{5}$ throughput phase to `[True, True, False, True, False]`. This pattern can be reproduced by working through what a fully-utilized 3-sequence, 5-array to 5-sequence, 3-array `SequenceArrayRepack` input schedule should be. Make a table with 5 columns, one for each clock cycle 0-4 (fully-utilized $\implies$ `cps=5`), and have entries for inputs, outputs, and leftovers in each column.



Visual aid. Fill in the chart for the next example.

On each clock cycle, one 3-array output must be produced. Represent this by drawing 3 dots in the output cells. This 0^{th} cycle's output must come from some input, so one 5-array input must come in on the 0^{th} cycle, leaving 2 array entries left over that must be stored in `SequenceArrayRepack`'s internal buffer. Draw 5 dots in the 0^{th} input cell and 2 dots in the 0^{th} leftover cell to represent this.

2 array entries are not enough to populate the 1^{st} cycle's 3-array output, so on that cycle we must read in another 5-array input. By cycle 1 then, the repack will have read in 10 and output 6 array entries, leaving 4 stored internally. Draw 5, 3, and 4 dots in the first column to represent this (input, output, leftover).

Now that there's 4 array entries buffered, we can produce another 3-array output on the 2^{nd} cycle without reading in another array. By the earlier rule, we defer reading in another valid input for now, producing the 3-array output using buffered inputs. Fill in 3 output dots and 1 leftover dots in column 2 to represent this.

Fill in the last 2 columns using the same rule. There should be input 5-arrays on cycles 0, 1, and 3, leading to the phase pattern `[True, True, False, True, False]`. For each cycle 0-4, there should be 2, 4, 1, 3, and finally 0 leftover entries.[5]

[5]That there are 0 leftover entries at the end is notable: we would just repeat if we continued the

8.3 SequenceArrayRepack Example

I think some of the confusion over my proposed `SequenceArrayRepack` comes from confusion between the concrete `SequenceArrayRepack` being analyzed and the imaginary `SequenceArrayRepack`s used to determine phase patterns.

Let's work through the behavior of a repack that accepts as input 4-sequences of 3-arrays, produces 3-sequences of 4-arrays, and has `cps=6`. This repack has an input throughput of $\frac{4}{6} = \frac{2}{3}$ and an output throughput of $\frac{3}{6} = \frac{1}{2}$.[6]

Determining the input and output phases requires us to construct a couple of `SequenceArrayRepack`s that don't really resemble the repack we're analyzing – we'll get an input phase of `[True, True, False]` and an output phase of `[True, False]`. If we just repeat this phase pattern without any added latency, some array entries will be scheduled for output before they were scheduled for input, so we need to have `sequentialLatency=1`.

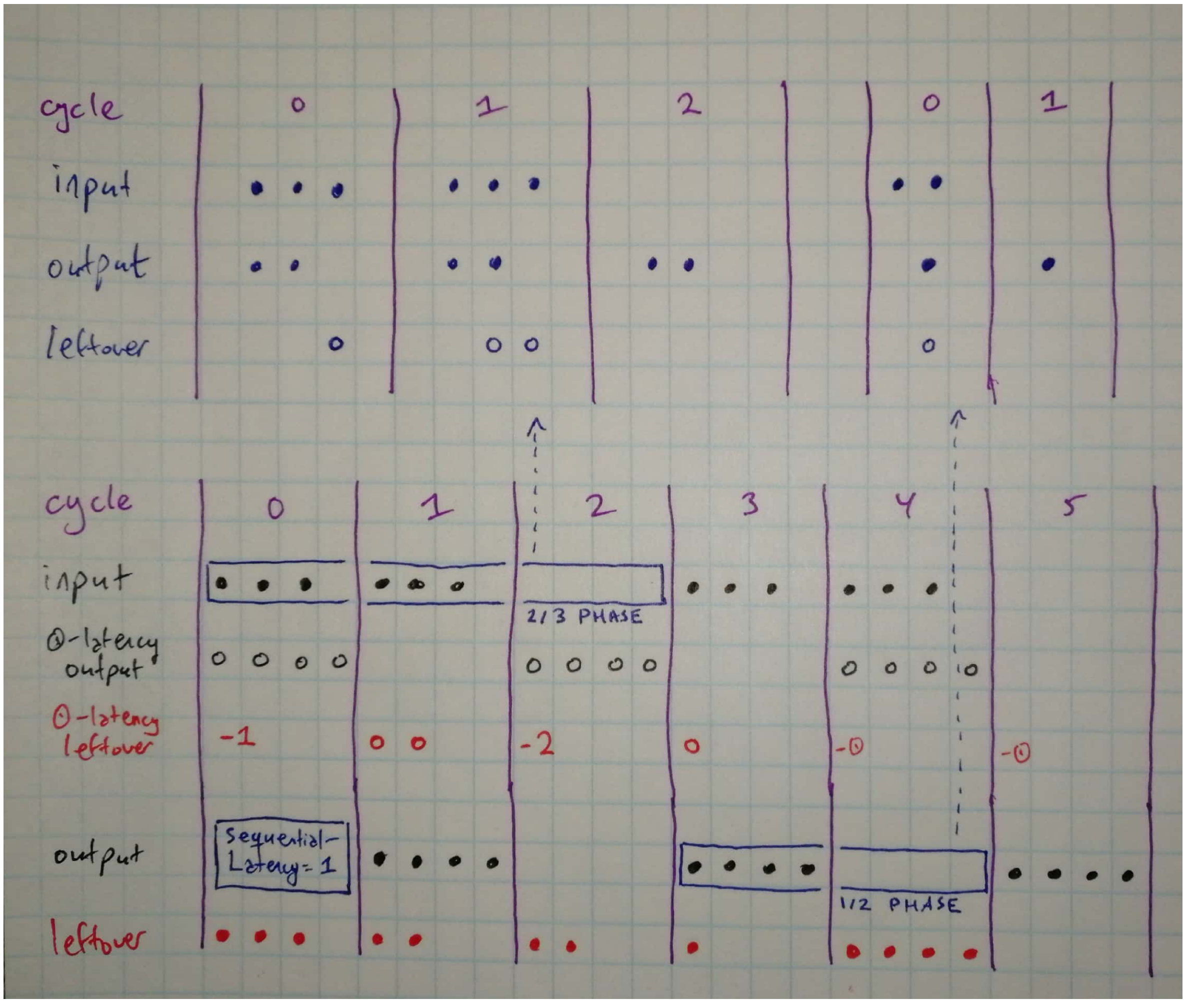
---

phase pattern for another 5 cycles. If the throughput fraction $\frac{x}{Y}$ were not in reduced form, we would still get the same phase (recall that phase repeats, so `[True, False]`-repeated $\equiv$ `[True, False, True, False]`-repeated. Although I've explained this as a fraction-to-phase mapping, in practice it's a mapping of `seqLen`$\equiv x$ and `cps`$\equiv Y$ to phase, so it's not actually guaranteed *a priori* that two ops with matching fractional throughputs will have matching phase if the $\frac{\text{seqLen}}{\text{cps}}$ fractions aren't in reduced form. That the phase is still the same without reduced fractions is pretty important to this scheme, but no one else seemed to be worried about this so it's in this footnote here.

Actually, there's one last thing to note, which is that an X-array to Y-array `SequenceArrayRepack` has the same schedule (all else being equal) as an nX-array to nY-array repack, n :integer. This is the other half of the phase being the same without reducing fractions.

If you think about the dot chart from earlier, each input, output, and leftover cell will have a multiple-of-n number of dots. The chart would be equivalent if we represented each group of n dots as a single dot, matching the original X-array to Y-array repack's chart, and therefore its schedule.

[6] As explained in another footnote it doesn't really matter if these fractions are reduced, but it makes the example cleaner.



Example of underutilized 3-sequence of 4-array to 4-sequence of 3-array `SequenceArrayRepack` (`cps=6`). The bottom chart shows the schedule of the actual repack being analyzed (negative entries on the 0-latency leftover row illustrate the problem of array entries travelling backwards in time). The top two charts represent the imaginary line buffers used to determine the input and output phase.

As another example,

8.4 Rationale

9 Test Writing

The tests I designed include

1. Haskell tests for the functional simulator I designed.

2. Python tests for the hardware line buffer David Durst is implementing in Magma based on "The Line Buffer Manifesto" specifications. These tests use the CoreIR simulator.

3. Tests for the sequential compose (`|>>=|`) and parallel compose (`|&|`) Haskell Aetherling operators. These tests check that the produced Haskell IR node is correct given valid operands, and check that the operators reject invalid operands (mismatched port types, mismatched synchronous and ready-valid timing, and mismatched throughputs with synchronous timing).

4. Tests for the `ComposePar` retiming passes. There are 18+ test cases, some of which have compose ops nested several layers deep meant to check for corner cases.