Contributions to Aetherling

David Akeley

September 10, 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.

0 Summary

Most of my work is on the Haskell portion of Aetherling. Aetherling supports a Haskell intermediate representation (IR) of hardware pipelines as an abstract syntax tree (AST) of ops. These ops include arithmetic ops (e.g. multiplication) and meta-ops that take other ops as children (e.g. `MapOp`, which duplicates an op in parallel; `ComposeSeq`, which wires the outputs of one op to the inputs of another).

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 pipeline (in IR form) using pure Haskell code.

3. Demonstration Apps

   I designed two demonstration Aetherling pipelines: Gaussian blur (7×7 stencil) and mipmap generation. These pipelines 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 & Helper Functions

   An op is a node of an Aetherling AST (along with its children). Previously there were multiple ways to express the same functionality. I simplified the ops to minimize redundancy, mainly by removing unneeded type parameters for arithmetic ops. Then, I wrote some Haskell IR 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.

5. 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 pipeline 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.

6. 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 pipeline (in Haskell IR form), 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.

7. 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 informa-

tion 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.

8. 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.

1 Line Buffer Specifications

One of the main goals of Aetherling is to facilitate automatic parallelization of hardware pipelines. To this end, Aetherling implements speed-up passes that determine the correct way to parallelize each type of op in an Aetherling pipeline. For example, an addition op can be parallelized by being wrapped in a map; an existing map can be parallelized by increasing the map's parallelism (lane count).

This parallelization turned out to be difficult for the original line buffer op Aetherling supported. The line buffer reads in an image as a stream of pixels (streamed in left-to-right, row-by-row), and emits as output a stream of rectangular portions of the image (we call these rectangles "windows"). Ops downstream from the line buffer can then base their results on pixels streamed in at different points in time.

The original line buffer was scheduled as follows: each time a pixel is streamed in, an output window with the streamed-in pixel at its bottom-right (highest x and y) is emitted, unless that window would include pixels outside of the image bounds. For example, if the window is 4 pixels tall and 3 pixels wide ($(4, 3)$; I use (y, x) coordinates with $(0, 0)$ being the top-left pixel[1]), pixel $(5, 5)$ would be input on the same cycle that the window with corners $(2, 3)$ and $(5, 5)$ would appear as valid output, while input pixel $(2, 0)$ would have no corresponding valid output window since the corresponding corner $(-1, -2)$ is out-of-bounds. (To make the distinction between valid and invalid outputs concrete, we say that if the line buffer were part of a ready-valid circuit, valid would be asserted only on cycles with valid output).

[1] I advocated for Aetherling to use (y, x) coordinates so that pixels would be streamed in lexicographical order.

The original line buffer could be parallelized by increasing the number of pixels streamed in per clock cycle. Since (almost) each input pixel corresponds to one output window's lower-right corner, multiple output windows would be produced per cycle as well.

This plan seemed simple, but in practice such parallelized line buffers were difficult to schedule. The main issue is that, with multiple windows output per cycle, it's possible for some to be valid and some to be invalid at the same time. This would require some complicated lane-interleaving logic downstream from the line buffer. Furthermore, even without paralellization, this line buffer would be difficult to place in a synchronously-timed circuit (i.e. non-ready-valid), since it produces an irregular output pattern. The output is stalled each time the input "passes over" the left margin of the image being streamed; for example, a line buffer with a 4-pixel wide window would produce no valid output whenever one of the first 3 pixels of a row is input.

A further drawback of the original line buffer is that it did not support an output "stride". Each output window is offset by 1 pixel from its nearest vertical and horizontal neighbors. This prevents mipmapping and pyramid-based image processing algorithms from being expressed as Aetherling pipelines.

2 Functional Simulator

3 Demonstration Apps

4 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*; `AndInt` 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[2] 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.[3]

5 ReadyValid Op

6 ComposePar Retiming

7 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.[4]

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

[2]Temporarily named `manifestoLineBuffer`.

[3]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.

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

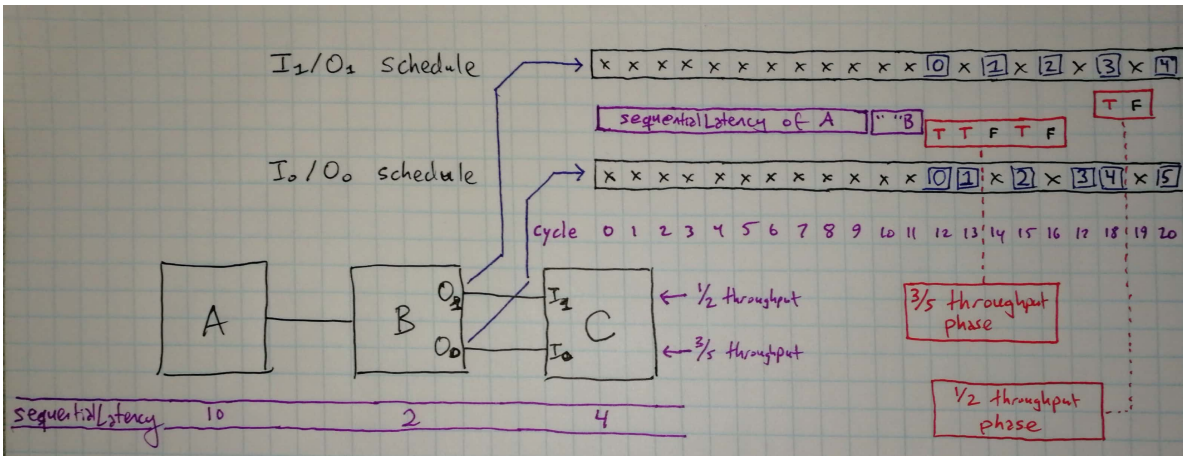
1. Each op has one `sequentialLatency` value – the difference in clock cycles between the time when its first input arrives and its first output is produced.[5]
2. Each port of an op is assigned a phase pattern based on its throughput. (In Aetherling, this throughput is calculated by dividing `seqLen`, 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.

7.1 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).

[5] 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.



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

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 O_0 will 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...).

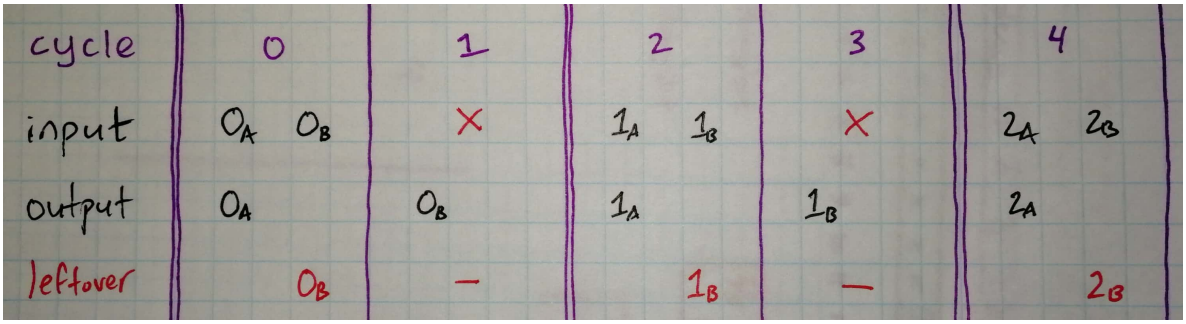
Finally, note that the initial inputs and initial outputs are always synchronized across ports. O_0 and O_1 start output on cycle 12, and I_0 and I_1 start expecting input on cycle 12. This means that the definition of `sequentialLatency` is well-defined. Using the definition, we expect that each input 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.

7.2 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. On each even-numbered cycle, one entry of the input 2-array won't

fit in the output 1-array, and will be "leftover" (i.e. buffered) for output on the next clock cycle.



`SequenceArrayRepack (1,2) (2,1)` repacking 2-arrays to 1-arrays.

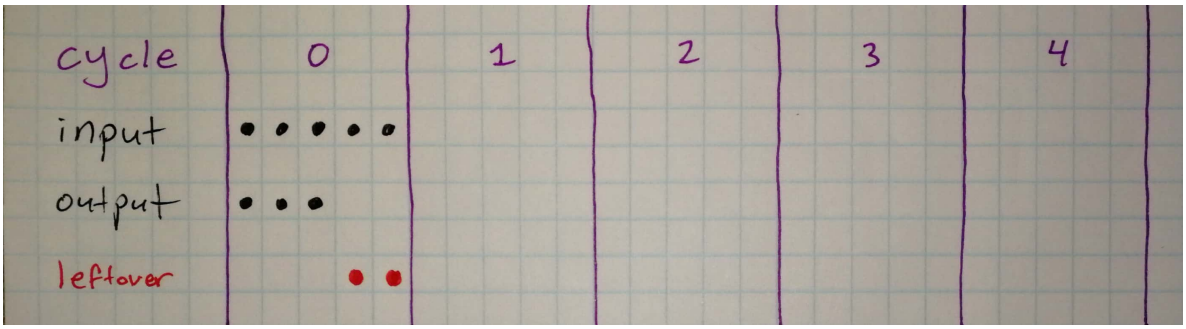
Each sequence gets its own digit.

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: the output arrays are narrower than the input arrays since

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

The repack is fully utilized, so the output comes out at maximum rate: Y outputs over Y clock cycles (in Aetherling terms, output `seqLen=cps=Y`).

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.

`SequenceArrayRepack (3,5) (5,3)`

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 alone 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.[6]

7.3 SequenceArrayRepack Example

I propose that all ops conform to this phase scheme, including `SequenceArrayRepack` itself. It's a bit hard to explain my proposed `SequenceArrayRepack`'s behavior because of pontential 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 $\frac{3}{6} = \frac{1}{2}$.[7]

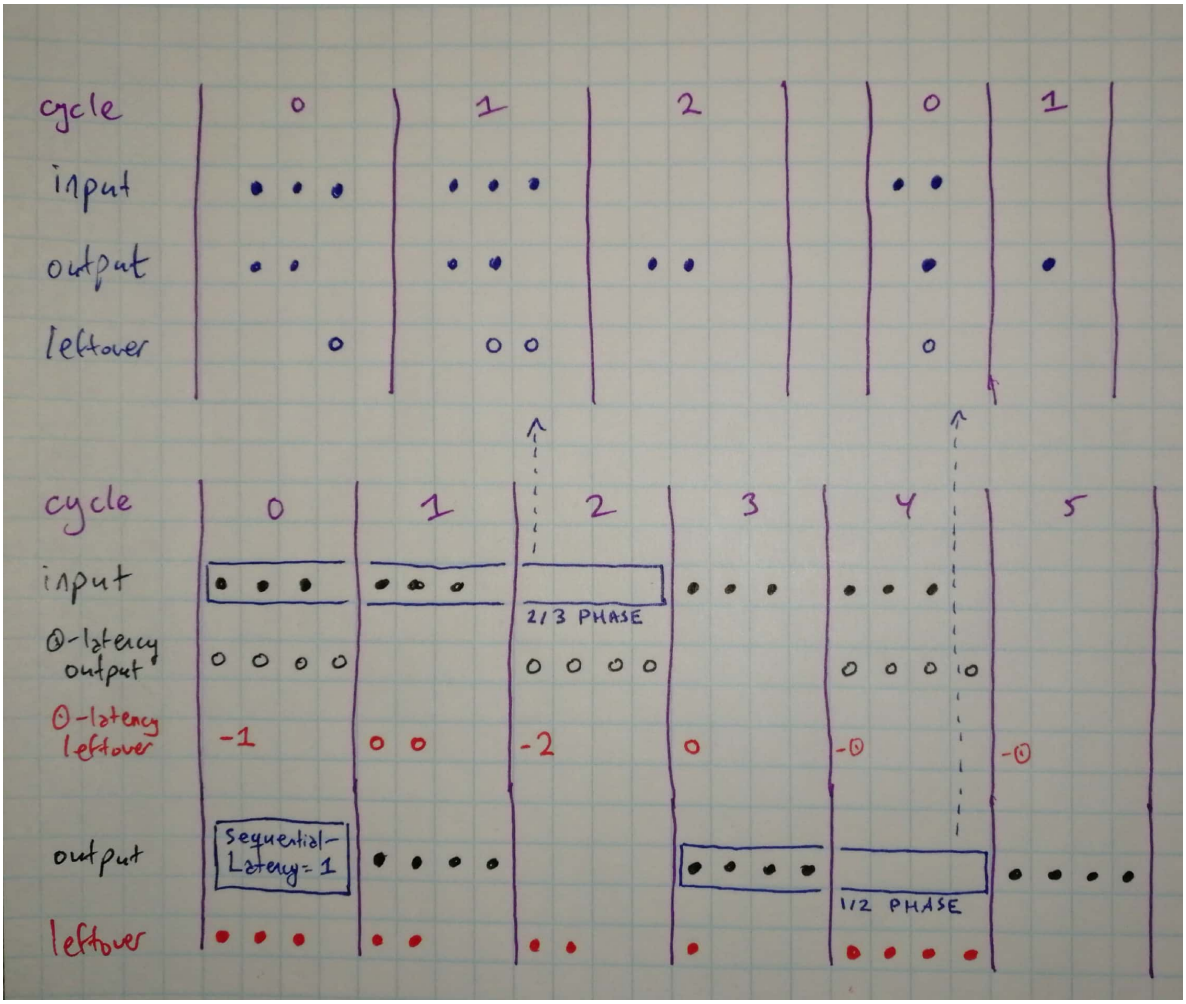
[6]That there are 0 leftover entries at the end is notable: we would just repeat if we continued the phase pattern for another 5 cycles. If the throughput fraction $\frac{x}{Y}$ were not in reduced form, we would still get the same phase (recall that phase repeats, so `[True, False]`-repeated $\equiv$ `[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.

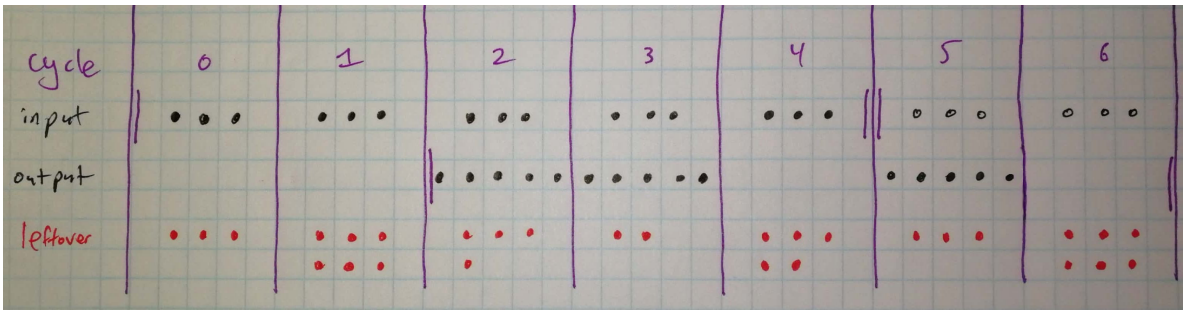
[7]As explained in another footnote it doesn't really matter if these fractions are reduced, but it makes the example cleaner.

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 – analyzing their input schedule as we did above, we'll get an input phase of `[True, True, False]` and an output phase of `[True, False]`. Focusing back on the original `SequenceArrayRepack` we're analyzing, 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`.



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 without added latency). The top two charts represent the imaginary line buffers used to determine the input and output phase.

As another example, consider the widening counterpart of the first line buffer example. This widening line buffer converts from 5-sequences of 3-arrays to 3-sequences of 5-arrays, with `cps=5`. (Figure 1)



Widening repack schedule (`SequenceArrayRepack (5,3) (3,5)`).

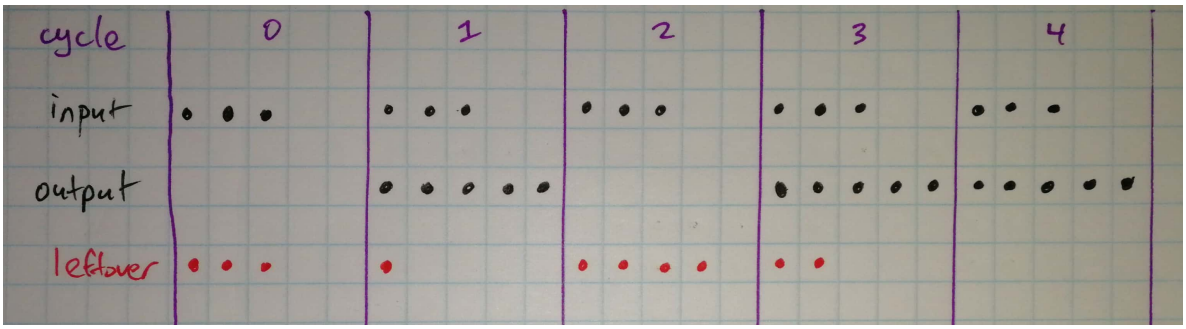
Figure 1: `sequentialLatency=2`; the initial valid output comes 2 cycles after the initial valid input.

Here, the input phase is `[True]` based on input throughput 1, and the output phase is `[True, True, False, True, False]` based on looking up the phase for output throughput $\frac{3}{5}$. This phase matches what the corresponding narrowing repack (`SequenceArrayRepack (3,5) (5,3)`) expects – indeed, under this plan, this widening repack can have its schedule matched with any other op with throughput $\frac{3}{5}$.

7.4 Rationale & Alternatives

The issue with requiring a consistent phase pattern across ops is that an op may be forced to have more latency (and therefore more buffer space needed) than theoretically required. For `SequenceArrayRepack (3,5) (5,3)`, (Figure 1) the `sequentialLatency` cannot be decreased to 1 with this output phase (we'd have negative leftover inputs at cycle 2). With `sequentialLatency=2`, the widening repack always has at least one value buffered on each clock cycle, and, on cycles 1,4,6... will have a full output array's worth of leftover outputs. In principle latency and buffering could be reduced by emitting arrays at cycles 1, 3, and 4 instead of 2, 3, and 5, adopting the alternate phase

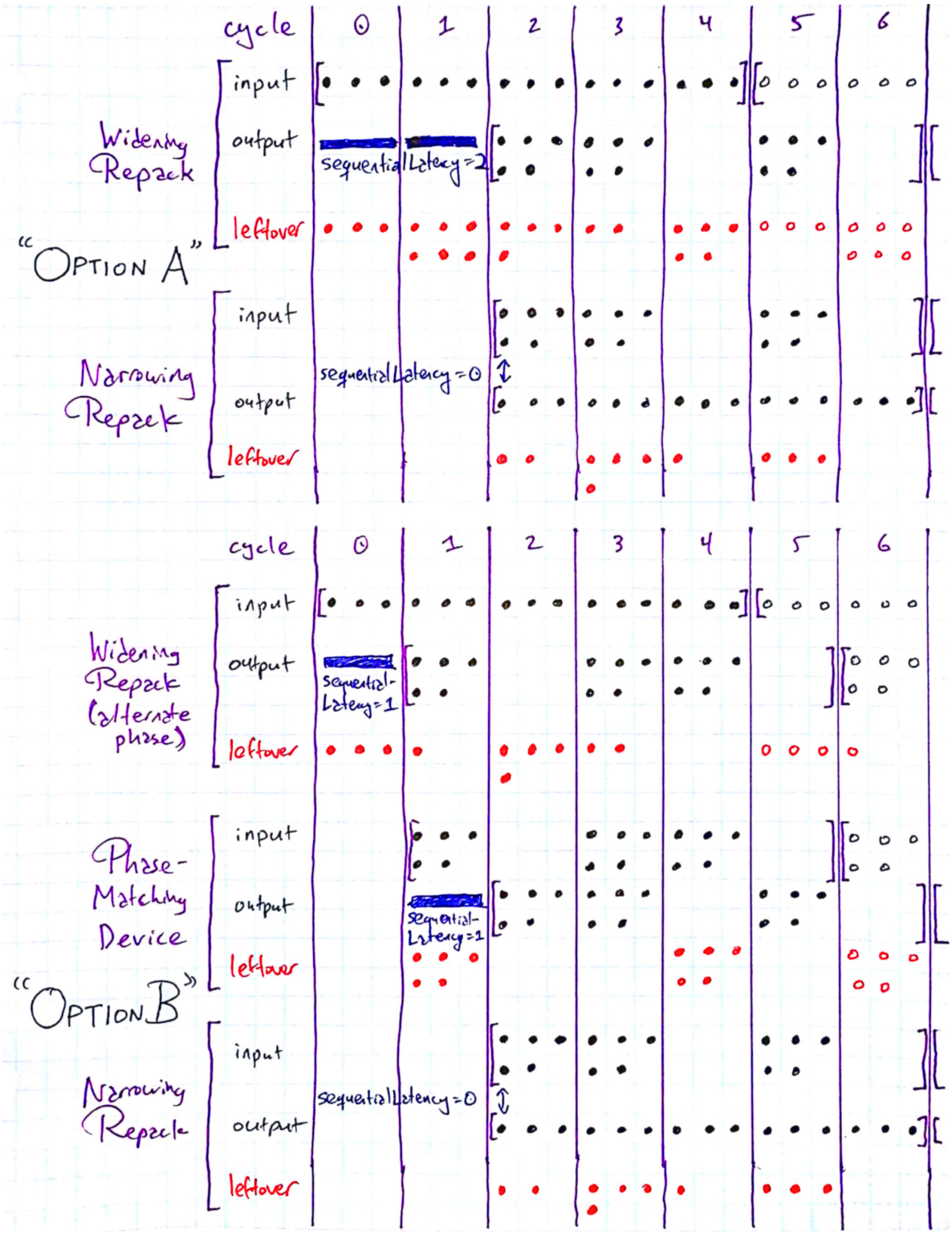
```
[True, False, True, True, False]
```
(1)



The problem with not matching phase is that, while translating Haskell IR to actual hardware, we'll have to add additional phase-matching buffers between ops with matching latencies but different phase patterns.

As an example, consider two options for wiring a widening repack to its corresponding narrowing repack.[8] Option A is for the widening repack to adopt the phase that the narrowing repack expects (as I propose). Option B is for the widening repack to adopt the alternate output phase at (1), and for a phase-matching device to be placed between the two repack ops.

[8]This is a contrived example, but the costs illustrated also apply to more realistic circuits where there is logic inserted between the two `SequenceArrayRepack`s.



Comparison of two options: top is my proposal, botton is rejected alternative.

Information flows from top-to-bottom.

First, at least in this example, the latency decrease of the alternate widening `SequenceArrayRepack (5,3) (3,5)` is cancelled out by the 1 latency increase introduced by the phase matching device. I suspect that any circuit area decrease by reducing buffer space in the widening repack would also be cancelled out by the buffers in the phase matching device. In fact, the area of option B is probably higher that option A, because the phase matching device would need its own duplicate of the muxes and counters that the repack devices presumably use in their implementation. Think of it this way: conceptually, the widening repack in option A combines the functionality of the phase-matching device and widening repack in option B, using only one counter and set of muxes to do so.

In my view though, the major issue with automatically adding phase-matching devices is that this adds extra cost and latency not visible in the Haskell IR, which

1. makes it more difficult for the user to reason about the latency and area of a circuit represented in Haskell IR form.

2. makes my `ComposePar` retiming passes less useful (or much more difficult to write), with the true latency of different paths of the circuit impossible to compute from the Haskell IR alone, which does not provide phase information.

Another alternative I considered and rejected was a simpler throughput-to-phase mapping (or at least a mapping simpler to explain). The phase pattern for a $\frac{x}{Y}$ throughput could have just been x valid values followed by $Y - x$ garbage values. The issues with this approach are:

1. This would increase the buffer space needed in a `SequenceArrayRepack` (which is ultimately the source of fractional underutilization). Intuitively, even for widening repacks and underutilized repacks, for which my proposed phase would not be the one that is theoretically most efficient, it would still be more expensive to have all inputs or outputs scheduled at the start or end of a pattern rather than spread out through time, as my proposed phases require.

2. Using this mapping, we could only assume that two ports with the same throughput will have matched phases if the throughput fraction $\frac{x}{Y}$ is always in reduced form. The actual throughput-to-phase mapping I propose does not require fractions to be in reduced form; for non-reduced fractions $\frac{nx}{nY}$ my proposed mapping naturally creates a phase that just repeats itself n times. This seemed more elegant to me than manually reducing fractions.[9]

[9] In hindsight this isn't all that great a reason (the buffer space reason is more practical), but I include it because this is probably the real reason I prefer the more complicated-to-explain throughput-to-phase mapping I proposed.

8 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.