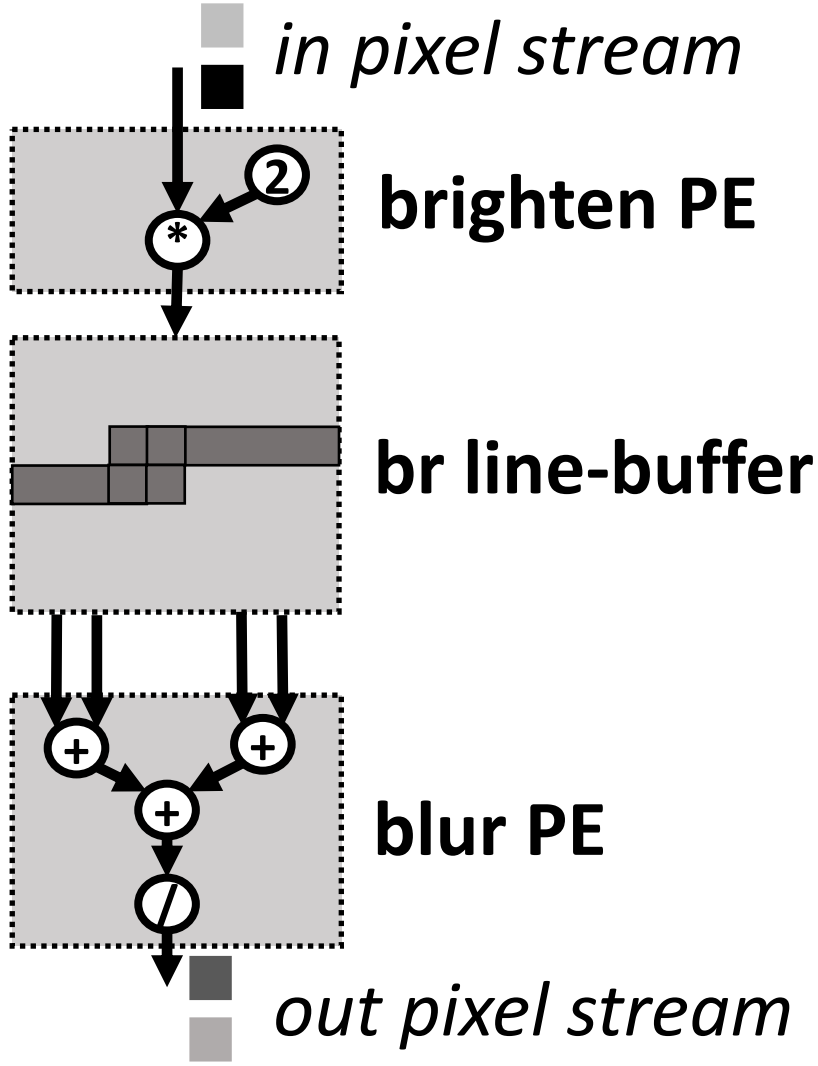
Adding Ready-Valid Channels to Clockwork

Dillon Huff

Goal: Turn programs written as for loops into high-performance dataflow architectures

```
for r in [0, 63]:
  for c in [0, 63]:
    br[r,c] = 2*in[r,c]

for r in [0, 31]:
  for c in [0, 31]:
    out[r,c] =
      (br[2r,2c]+br[2r,2c+1]+
       br[2r+1,2c]+br[2r+1,2c+1])/4
```



The original Clockwork produces a single, statically scheduled module for the application

```
for r in [0, 63]:
  for c in [0, 63]:
    br[r,c] = 2*in[r,c]

for r in [0, 31]:
  for c in [0, 31]:
    out[r,c] =
      (br[2r,2c]+br[2r,2c+1]+
       br[2r+1,2c]+br[2r+1,2c+1])/4
```

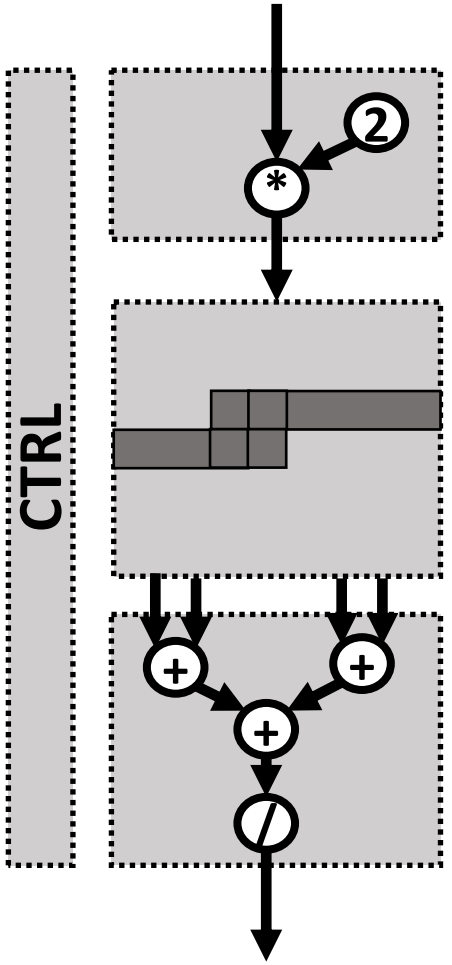
Scheduling →

```
shift_register<66> br;
for (int r=0; r<=63; r++)
 for (int c=0; c<=63; c++){
#pragma HLS pipeline II=1

  br.push(2*in.read());

  if ((r-1) % 2 == 0 &&
      (c-1) % 2 == 0)
    out.write((
     br.peek(65) + br.peek(64) +
     br.peek(1)  + br.peek(0)
     )/4);
}
```

HLS →



Arguments to the function become FIFOs

```
for r in [0, 63]:
  for c in [0, 63]:
    br[r,c] = 2*in[r,c]

for r in [0, 31]:
  for c in [0, 31]:
    out[r,c] =
      (br[2r,2c]+br[2r,2c+1]+
       br[2r+1,2c]+br[2r+1,2c+1])/4
```

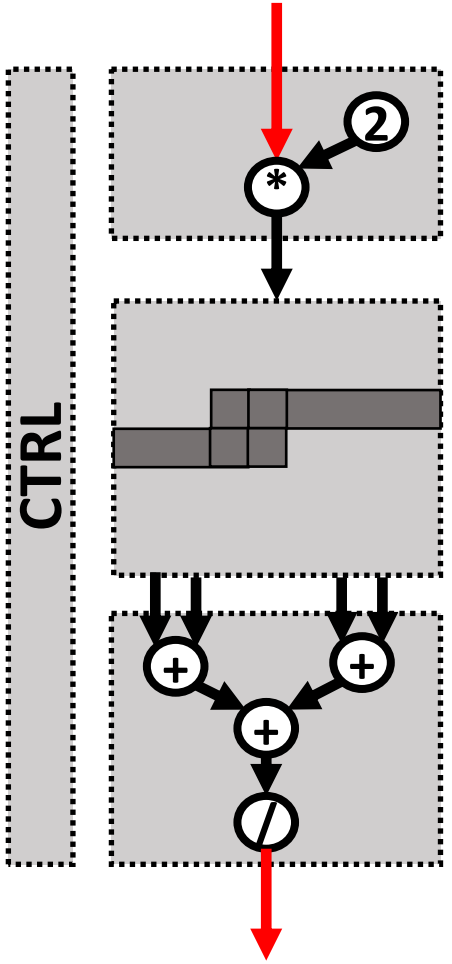
Scheduling →

```
shift_register<66> br;
for (int r=0; r<=63; r++)
 for (int c=0; c<=63; c++){
#pragma HLS pipeline II=1

 br.push(2*in.read());

 if ((r-1) % 2 == 0 &&
     (c-1) % 2 == 0)
  out.write((
   br.peek(65) + br.peek(64) +
   br.peek(1)  + br.peek(0)
   )/4);
}
```

HLS →



Internal arrays become re-use buffers

```
for r in [0, 63]:
  for c in [0, 63]:
    br[r,c] = 2*in[r,c]

for r in [0, 31]:
  for c in [0, 31]:
    out[r,c] =
      (br[2r,2c]+br[2r,2c+1]+
       br[2r+1,2c]+br[2r+1,2c+1])/4
```

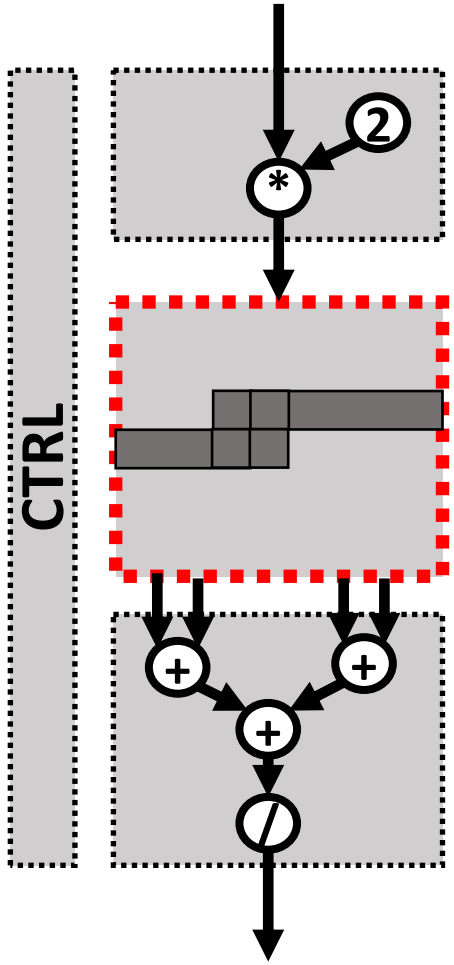
Scheduling →

```
shift_register<66> br;
for (int r=0; r<=63; r++)
 for (int c=0; c<=63; c++){
#pragma HLS pipeline II=1

 br.push(2*in.read());

 if ((r-1) % 2 == 0 &&
     (c-1) % 2 == 0)
  out.write((
   br.peek(65) + br.peek(64) +
   br.peek(1)  + br.peek(0)
   )/4);
}
```

HLS →



And all control logic is implemented by a single, global controller

```
for r in [0, 63]:
  for c in [0, 63]:
    br[r,c] = 2*in[r,c]

for r in [0, 31]:
  for c in [0, 31]:
    out[r,c] =
      (br[2r,2c]+br[2r,2c+1]+
       br[2r+1,2c]+br[2r+1,2c+1])/4
```

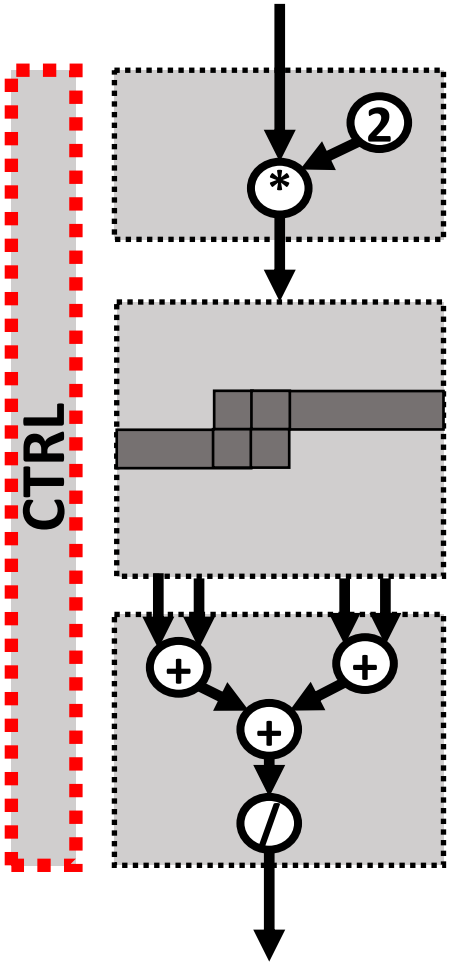
Scheduling →

```
shift_register<66> br;
for (int r=0; r<=63; r++)
 for (int c=0; c<=63; c++){
#pragma HLS pipeline II=1

  br.push(2*in.read());

  if ((r-1) % 2 == 0 &&
      (c-1) % 2 == 0)
    out.write((
     br.peek(65) + br.peek(64) +
     br.peek(1)  + br.peek(0)
     )/4);
}
```

HLS →

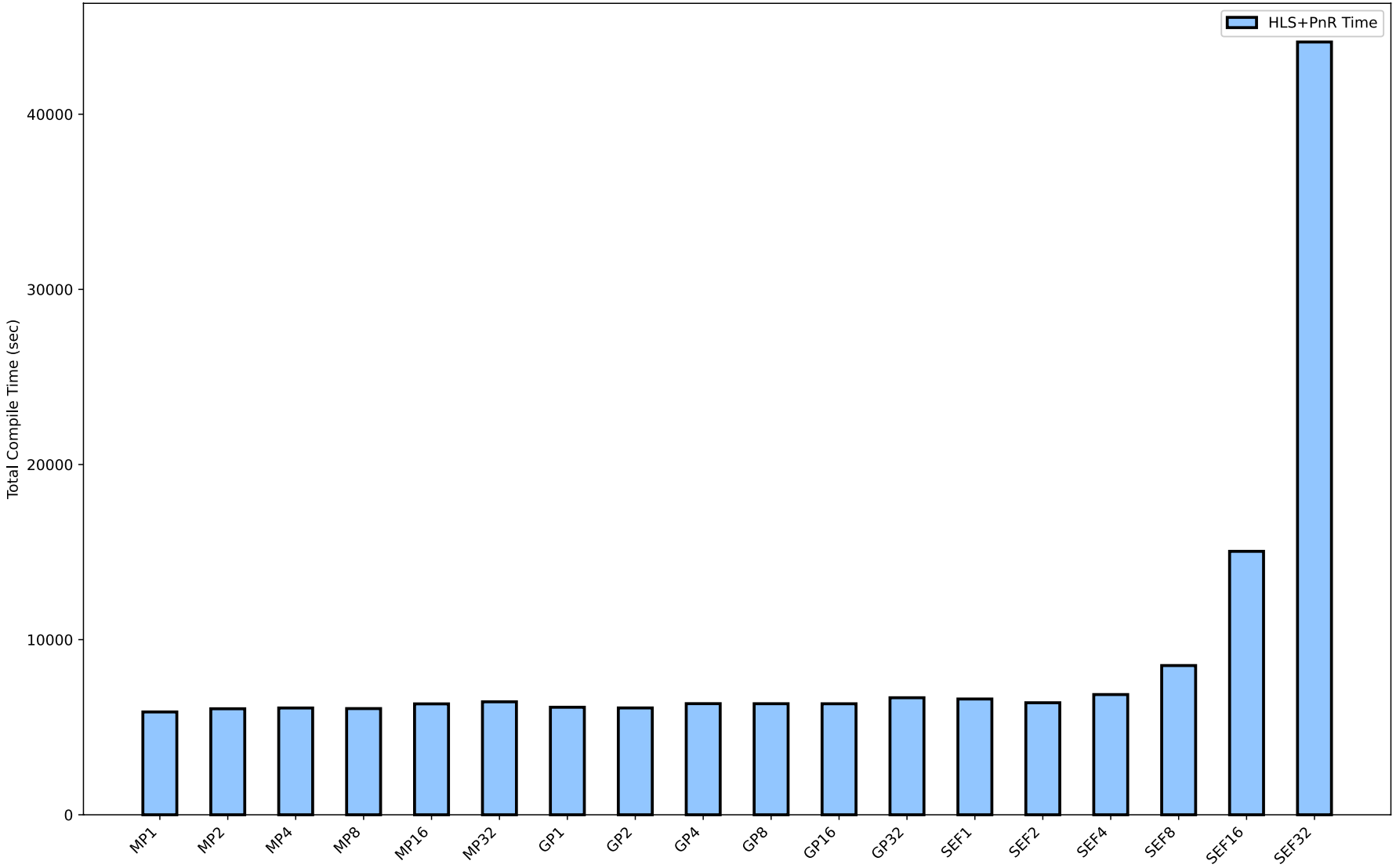


This strategy gives very high performance

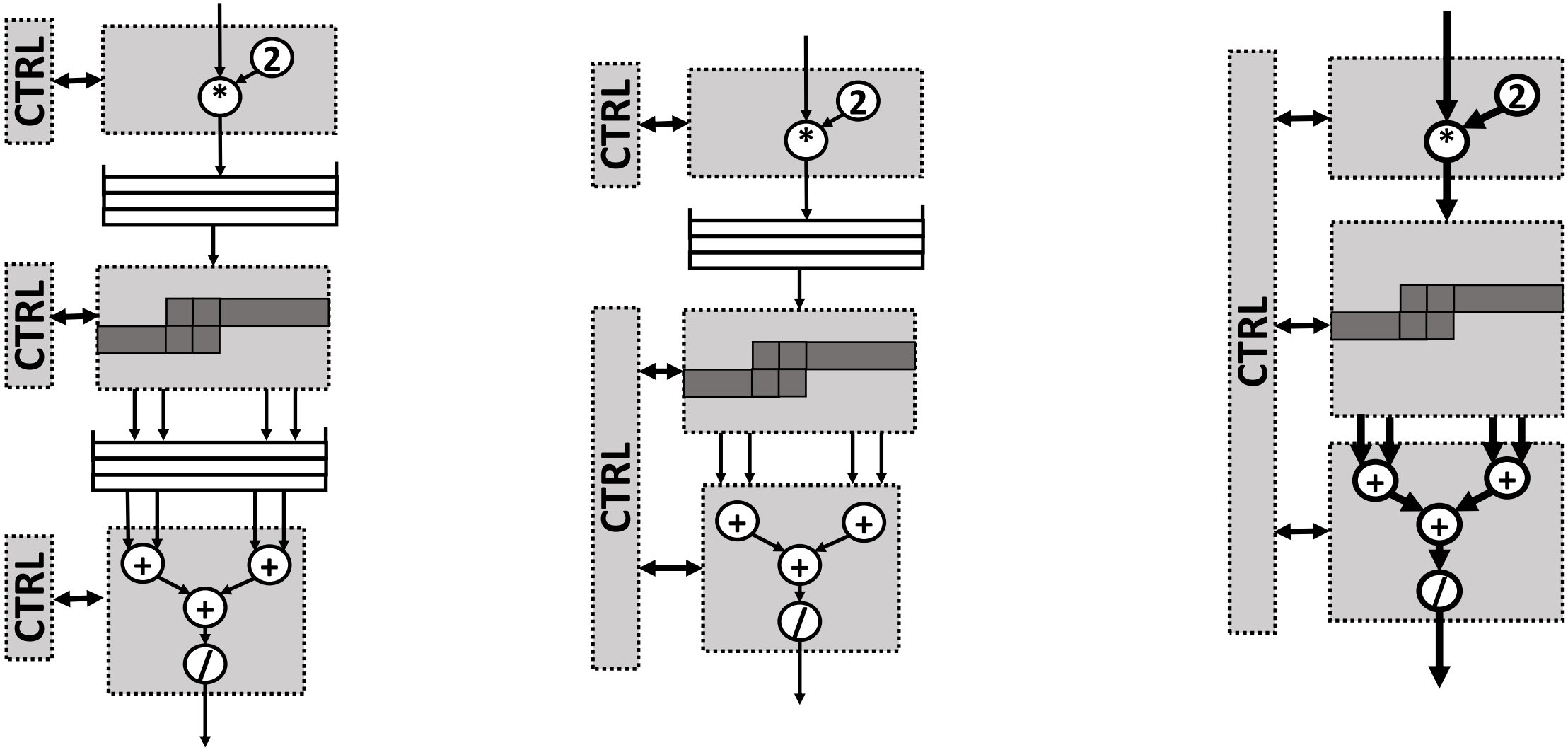
- All arithmetic / control is in the same module, so more opportunities for optimization are visible to synthesis tools
- There is no padding or fat on re-use buffers
- No energy or storage overhead from moving things in and out of FIFOs
- 55% reduction in LUT use compared to a state-of-the-art stencil compiler [1]

[1] Huff, Dai, Hanrahan. Clockwork: Resource-Efficient Static Scheduling for Multi-Rate Image Processing Applications on FPGAs. FCCM 2021 (to appear)

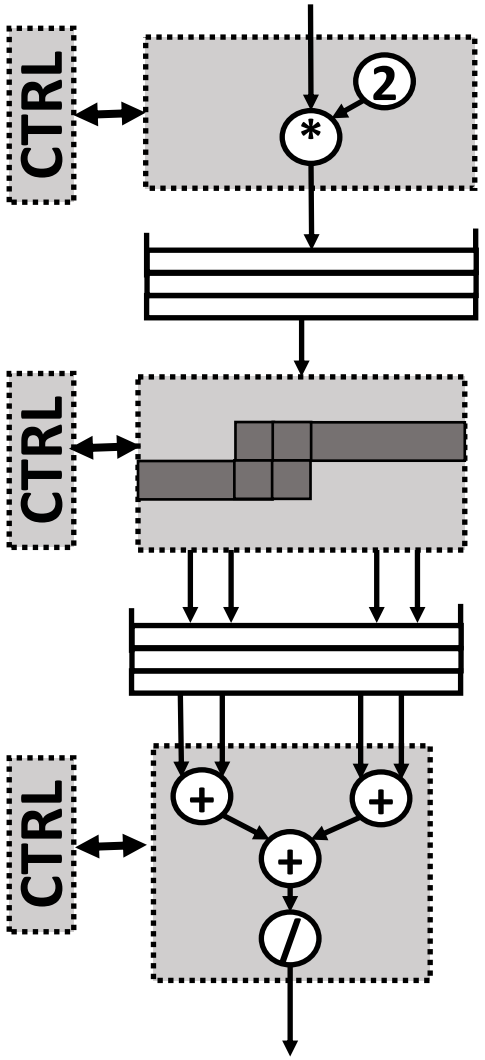
But large, statically scheduled designs stress the toolchain

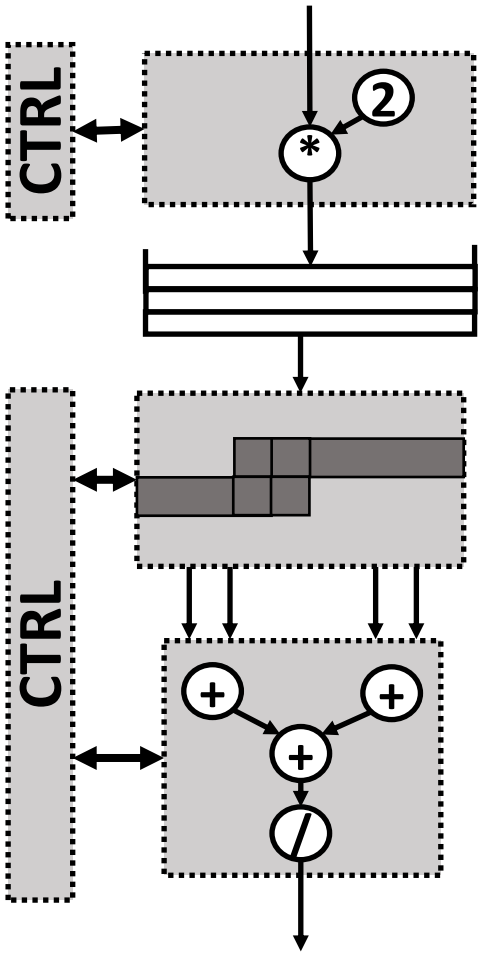


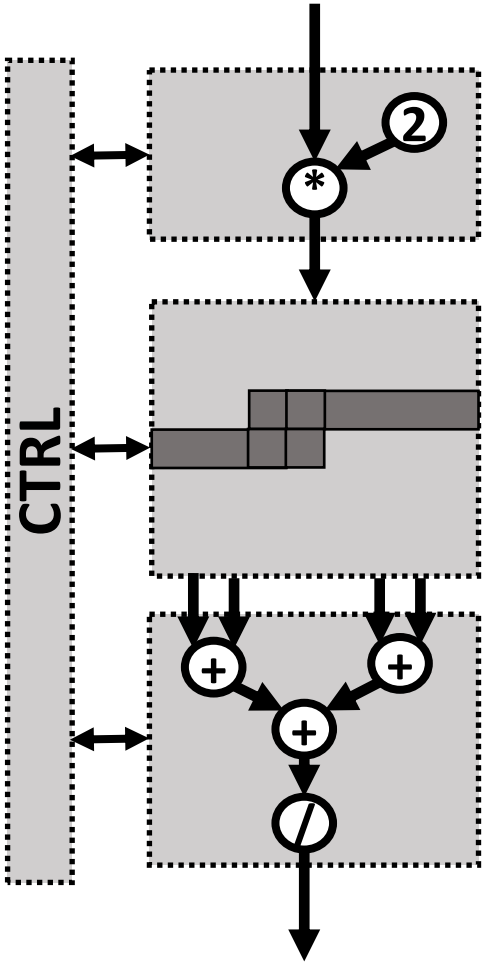
Solution: Break the application into smaller processes that communicate through FIFOs



Q: How large should the channels be?







Answer: Large enough to prevent deadlock and reach the target throughput

- **Deadlock:** If channels between modules are not large enough the design can get stuck

- **Throughput:** If channels along reconverging paths with different latencies are not long enough the design can stall unnecessarily

To prevent deadlock the channels just need to be large enough that the static schedule we compute *could* execute without blocking [1]

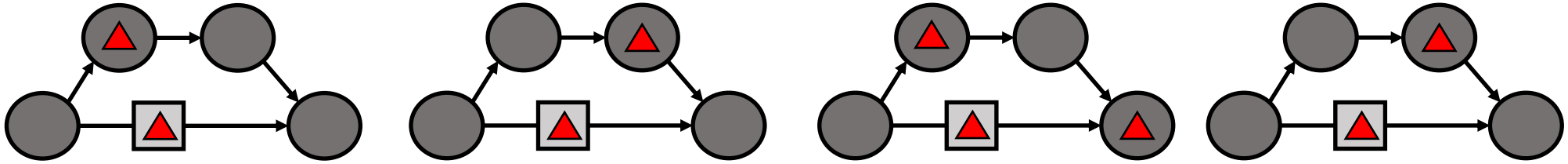
- The channel sizes can be computed using the same program analysis we use to size re-use buffers

- Or it can be done by simulation, but this will be slow

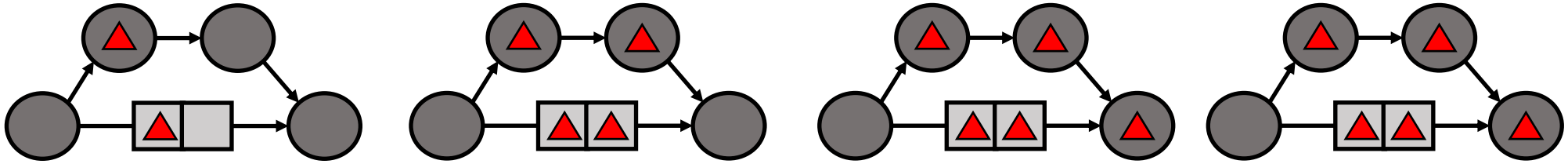
[1] Verdoolaege. Polyhedral process networks. Handbook of Signal Processing Systems 2013

The compiler also needs to pad FIFOs to balance reconvergent paths: slack matching

Without slack matching



With slack matching



time

A technical problem to watch out for: artificial deadlock

- This happens when the statically scheduled pipelines block on reads to empty channels and do not flush the already issued operations later in the pipeline
- This problem [1] can be prevented by careful syntax transformations of HLS code that are discussed here [2]
- And it won't happen if the design is slack matched aggressively to prevent writers from blocking

[1] Dai, Tan, Hao, and Zhang. Flushing-enabled loop pipelining for high-level synthesis. DAC 2014
[2] Chi, Choi, Cong, and Wang. Rapid cycle-accurate simulator for high-level synthesis. FPGA 2019

Evaluating the Results

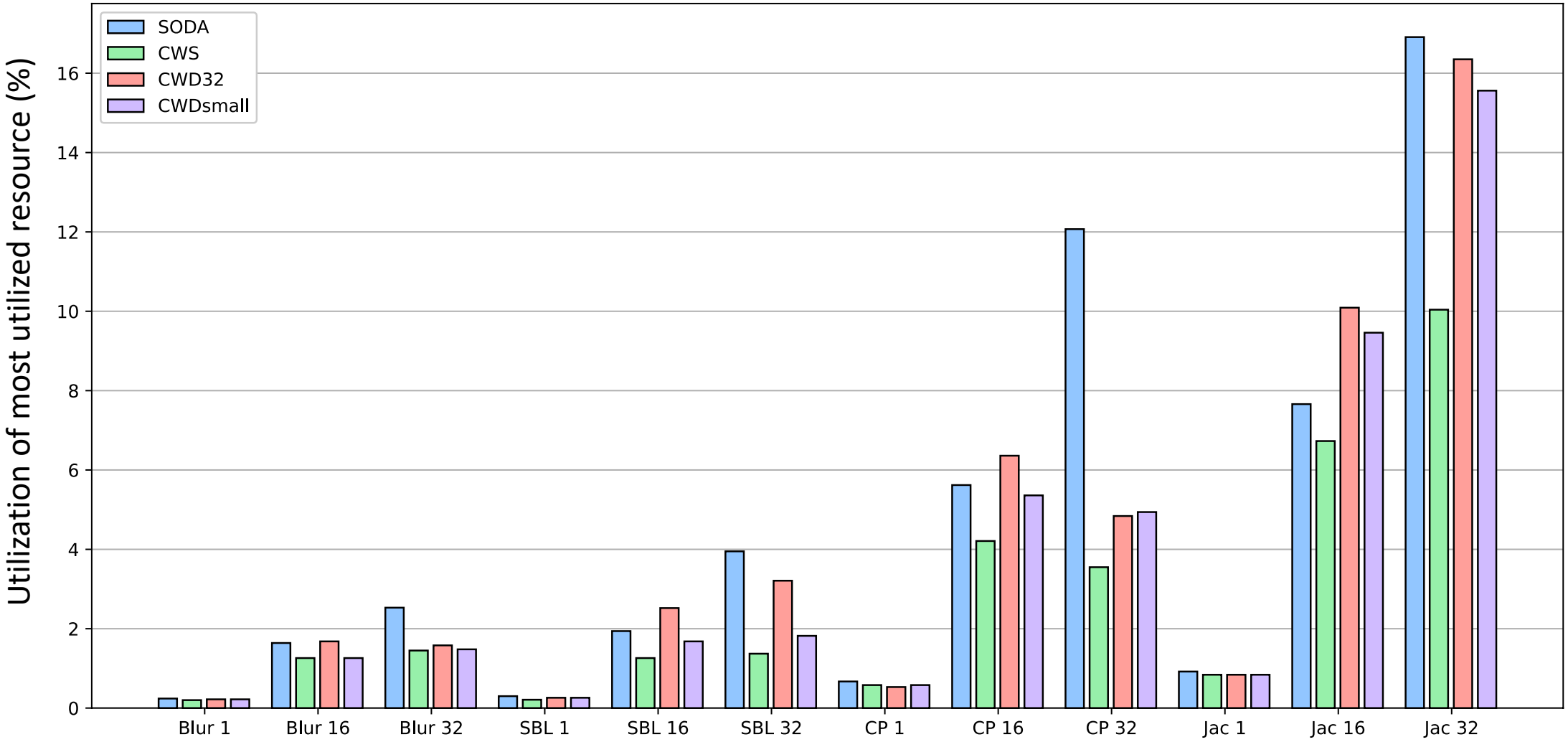
To evaluate the cost of channels vs. static scheduling we have tried 4 different strategies

- **SODA [1] –** Puts 32 deep channels between all PEs* and all banks of re-use buffers. Best paper award nominee at ICCAD 2018
- **CWS –** Completely static code generation. No channels
- **CWD32 –** All PEs and re-use buffers for a given stage in the same process. All channels have slack of 32
- **CWDsmall -** All PEs and re-use buffers for a given stage in the same process. All channels as small as possible for the throughput target
- All designs targeted 250MHz on a Virtex7 VU9P and used 16 bit fixed point arithmetic

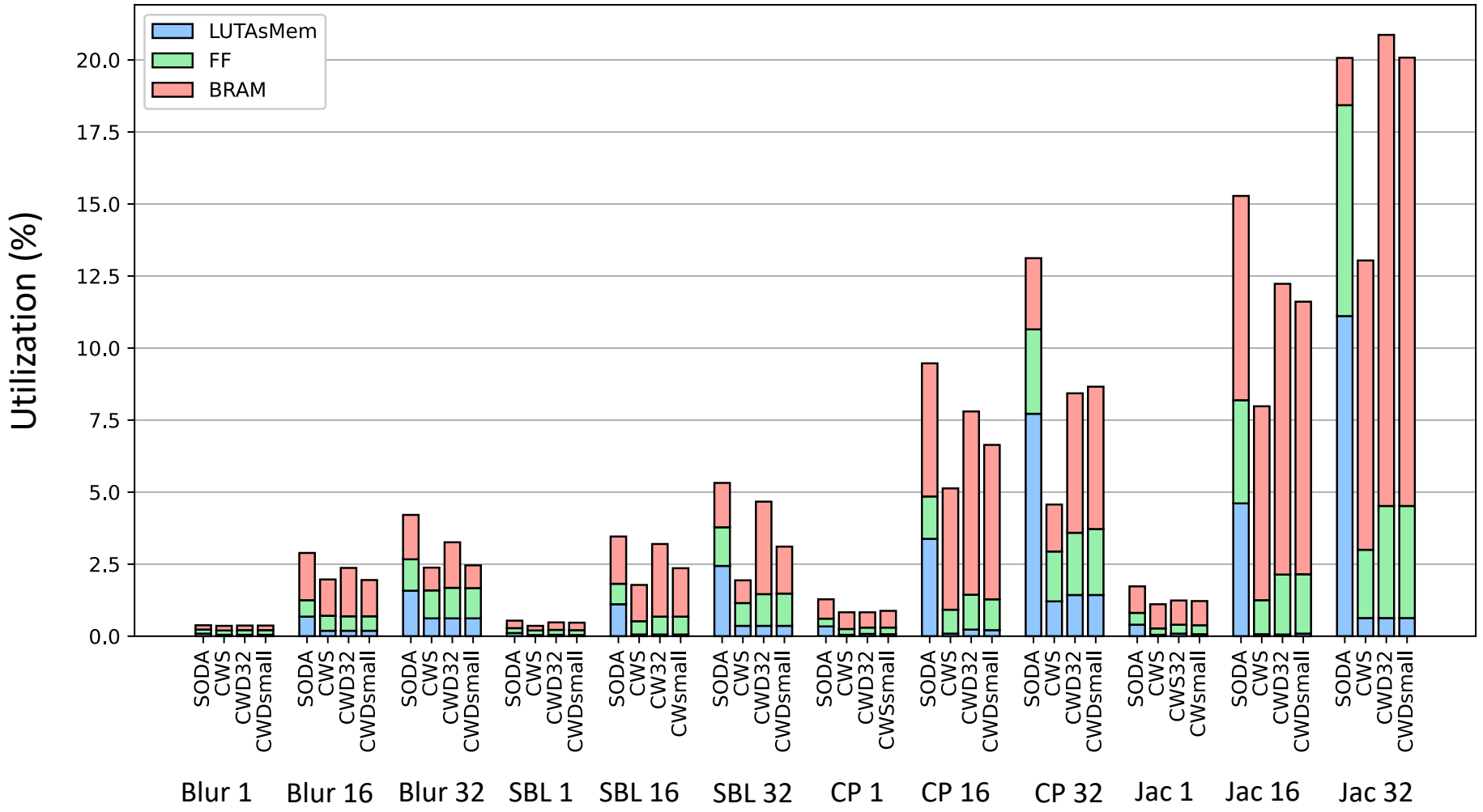
[1] Chi, Cong, Wei, Zhou. SODA: stencil with optimized dataflow architecture. ICCAD 2018

** Each ”PE” contains all the compute for the stage at 1 pixel per cycle. Much coarser grained than the PE tiles on our CGRA.*

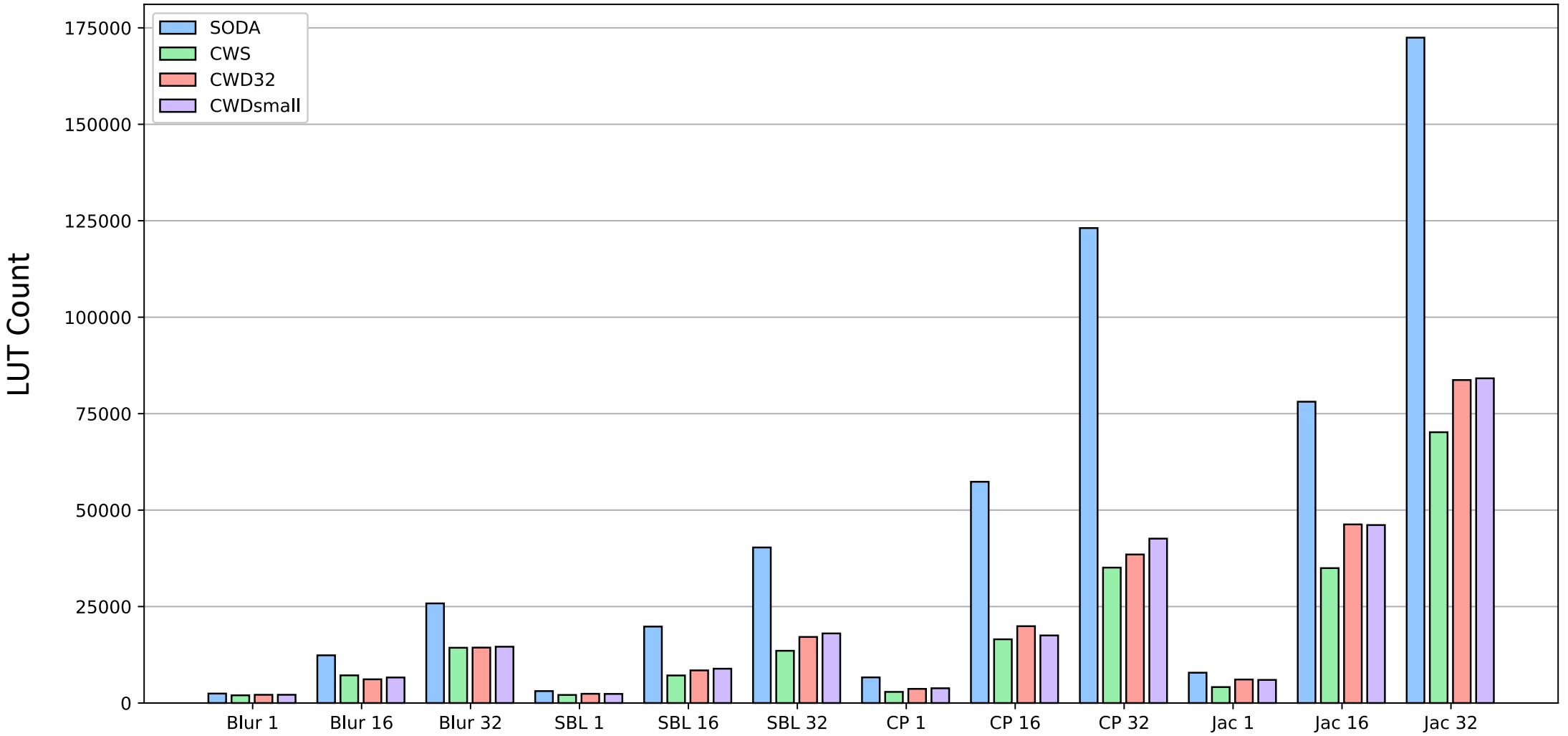
Static schedules have the best utilization



Memory mapping has a huge effect on utilization



LUT use was similar for all Clockwork variants

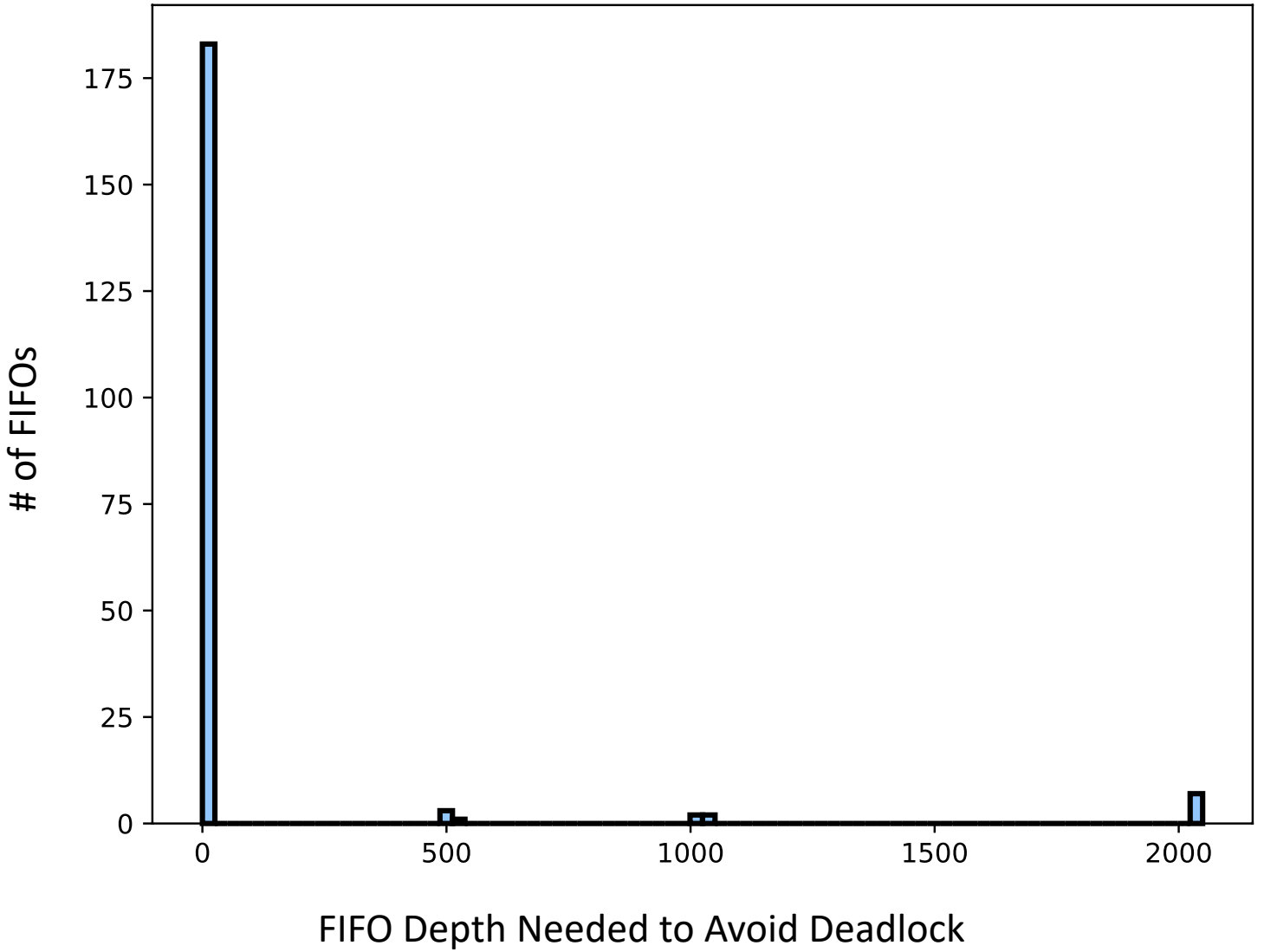


When arithmetic is expensive the only big difference is BRAM use

App	pix/clk	Compiler	LUT	LUTAsMem	FF	BRAM	DSP
LLF	1	DYN500	165697	10362	210137	286	1187
LLF	1	STAT	158396	14047	191997	214	1187
PB	1	DYN500	32311	2432	43845	79	151
PB	1	STAT	31322	3882	39328	61	151
PB	2	DYN500	35960	2369	48745	93	165
PB	2	STAT	34807	4169	42250	76	165
PB	4	DYN500	51532	3473	70382	118	240
PB	4	STAT	51537	6070	62132	84	240

**All designs used 32-bit floating-point arithmetic and ran at 250MHx*

Large FIFOs are rare



Some lessons learned about hardware compiler design

FIFOs are not a useful compiler abstraction

- During dependence analysis, scheduling, and FIFO sizing FIFOs are just arrays that are read and written in the same order by a pair of loops that represent the controllers at the ends of the FIFO

- The only time FIFOs are treated differently from other buffers is when we generate the text string (C++) that is sent to the HLS tool

The **data structures that represent scheduling and binding decisions** are the important parts of a hardware compiler

- Once you get them right other challenges (how to schedule, how to size FIFOs, how to design re-use buffers, etc.) just fall out
- For apps with no dynamism
  - The polyhedral model is great for representing schedules
  - Binding decisions are so simple we don't represent them explicitly
- For apps with more dynamism
  - The polyhedral model probably won't work
  - Binding decisions will need an explicit representation if the architecture chooses at runtime how to assign operations to functional units / memory ports

The **data structures that represent scheduling and binding decisions** are the important parts of a hardware compiler

- Once you get them right other challenges (how to schedule, how to size FIFOs, how to design re-use buffers, etc.) just fall out
- For apps with no dynamism
  - The polyhedral model is great for representing schedules
  - Binding decisions are so simple we don't represent them explicitly
- For apps with more dynamism
  - The polyhedral model probably won't work
  - Binding decisions will need an explicit representation if the architecture chooses at runtime how to assign operations to functional units / memory ports

Questions or comments?