Pipelining CGRA Applications

Jack Melchert

Motivation

Problem: The applications running on our CGRA run at very low frequencies

gaussian	68MHz
harris color	25MHz
unsharp	18MHz
camera pipeline	37Mhz
resnet	12MHz

The maximum frequency of the CGRA is 780 MHz at 1.1 V so we want applications to run at least at 500 MHz

Motivation

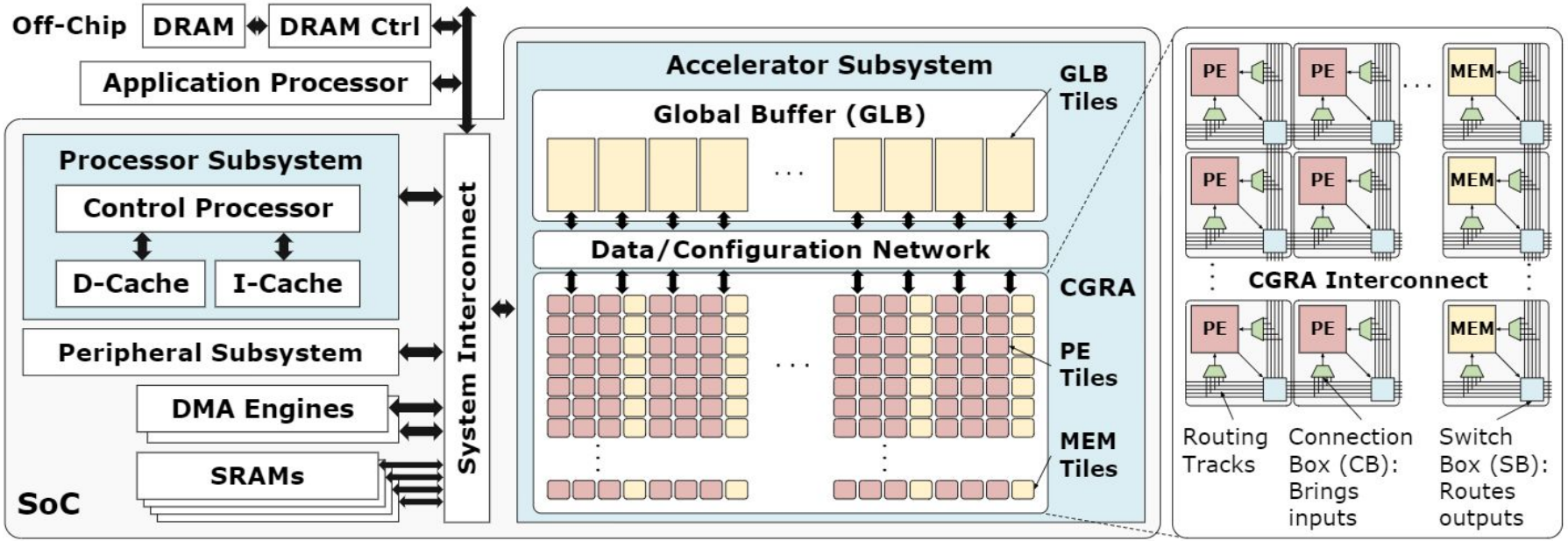
Solution: Pipeline applications

- Un-pipelined applications have a critical paths that typically start or end at a memory tile or outside of the array
- Adding pipelining registers to these paths should dramatically increase the maximum frequency of the applications

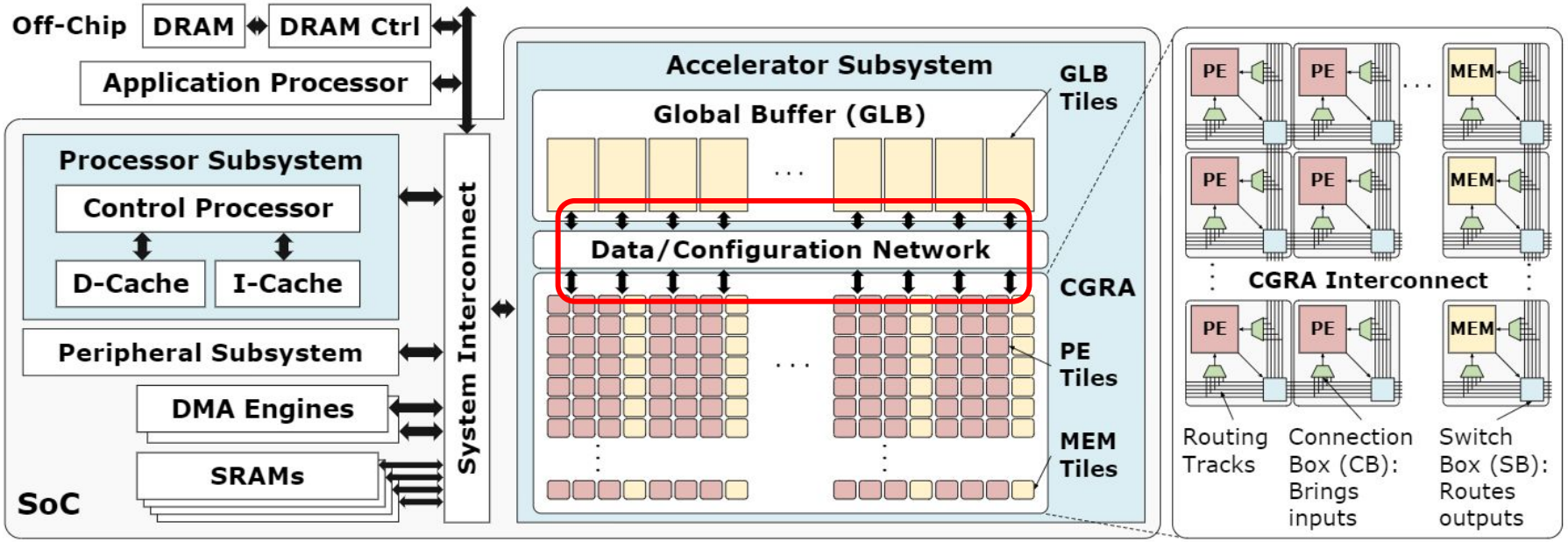
Goals

- Understand and model the delays in the array
  - Measure all contributions to the critical path
  - Integrate that data into a timing analysis tool that will accurately model the critical path in an application
- Create an automated pipelining approach
  - Our applications and hardware change frequently, so an automated approach is required
- Reach 500 MHz for all applications running on the array

Contributions to Critical Path Delay

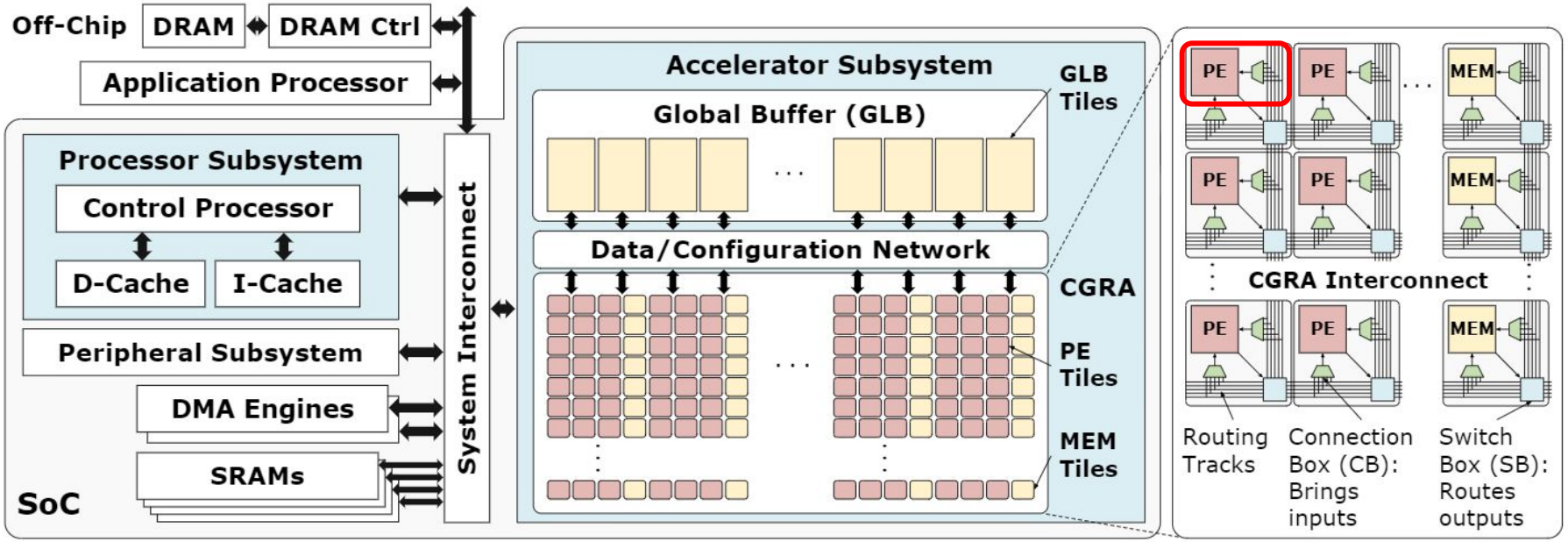


Contributions to Critical Path Delay



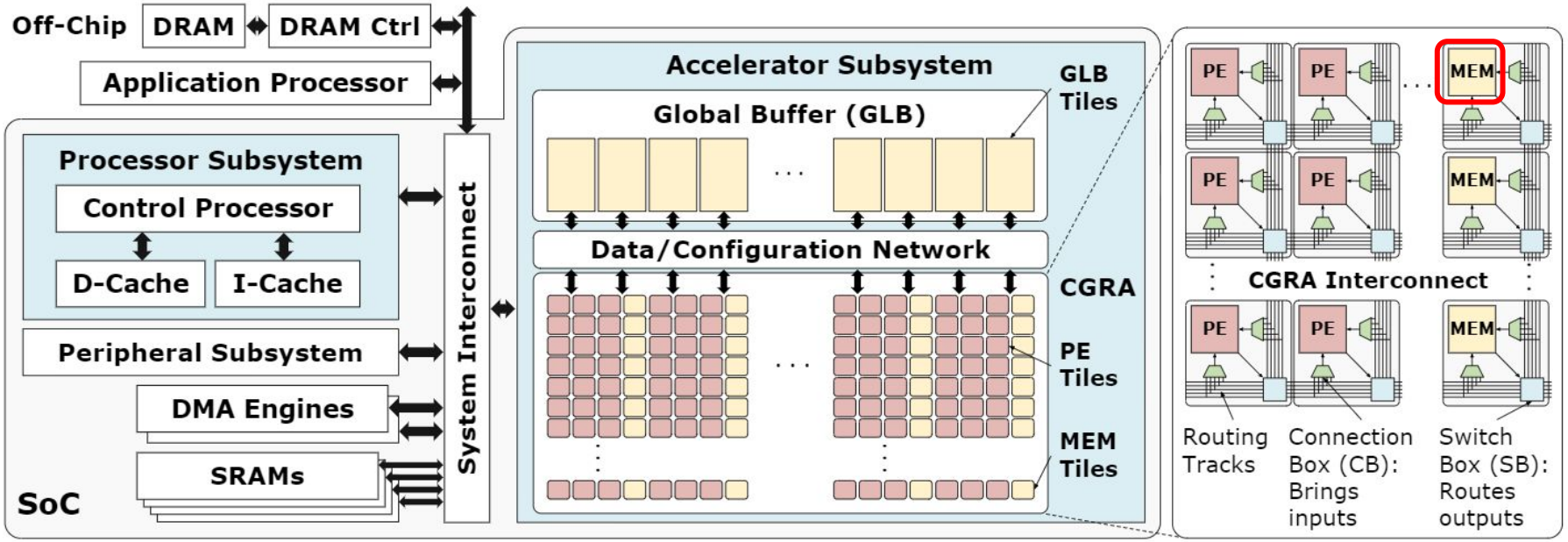
Global Buffer Delay: 1.1 ns

Contributions to Critical Path Delay



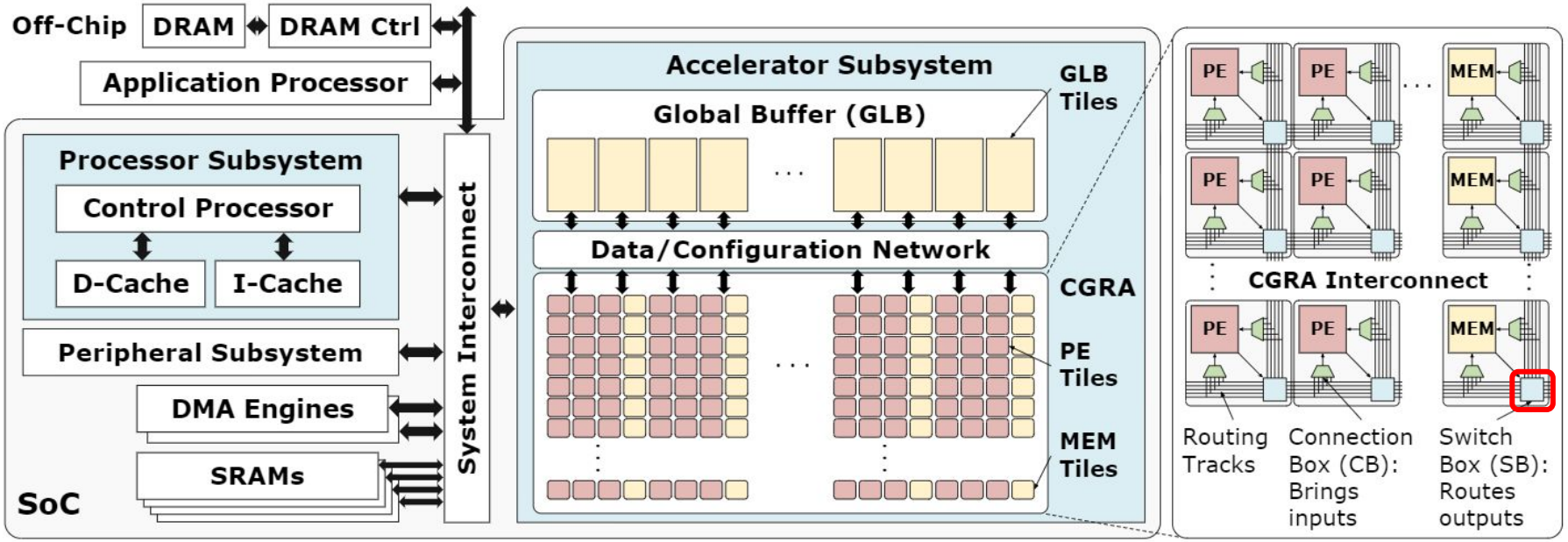
PE Delay: 0.48-0.70 ns

Contributions to Critical Path Delay



MEM Delay: 0 ns

Contributions to Critical Path Delay



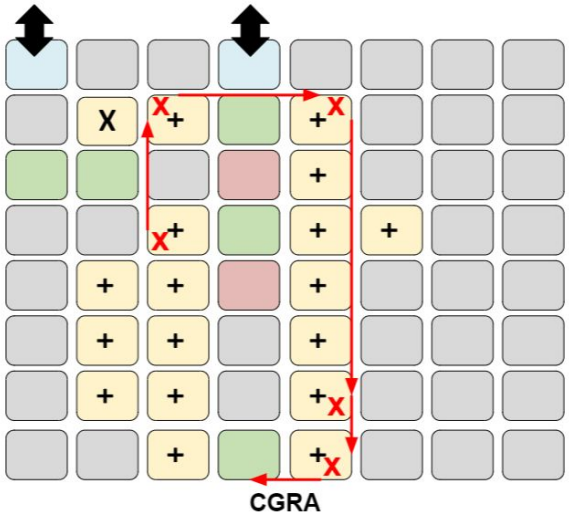
SB Delay: 0.14 ns

Critical Path Model

Contributions:

- Global Buffer: 1.1 ns
- Switch Boxes: 0.14 ns
- PEs: 0.48-0.7 ns

After about 3 hops in the interconnect, a critical path with 1 PE will be dominated by the interconnect



IO Tile

PE Tile

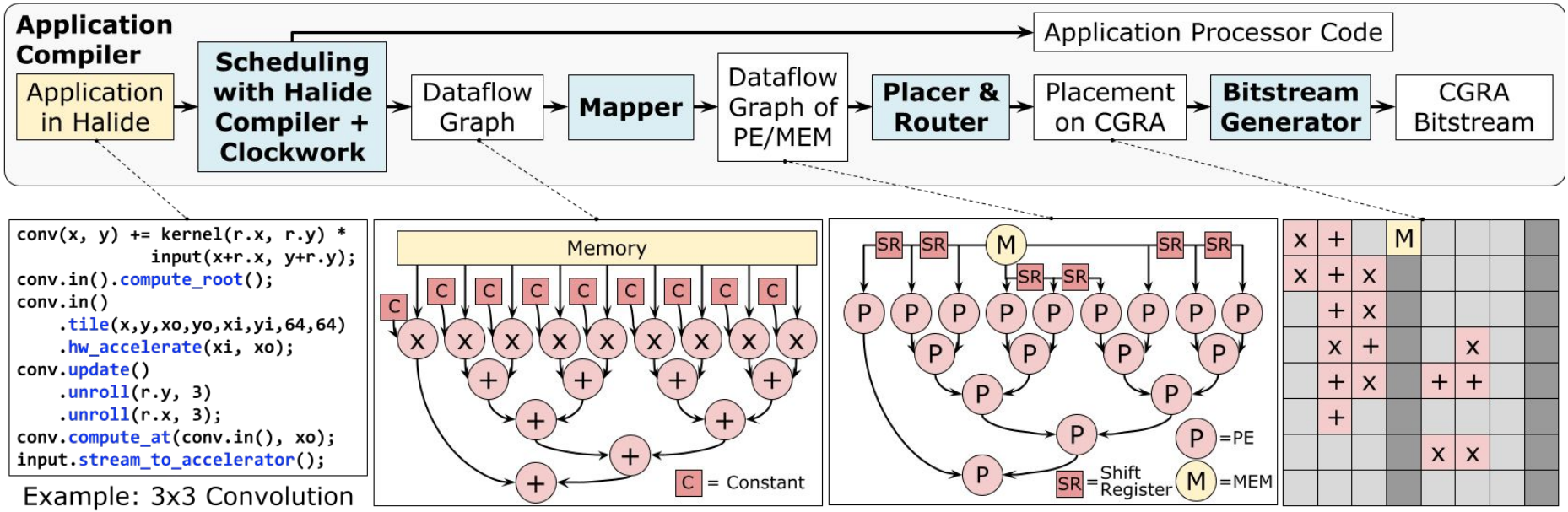
MEM Tile

Register in Tile

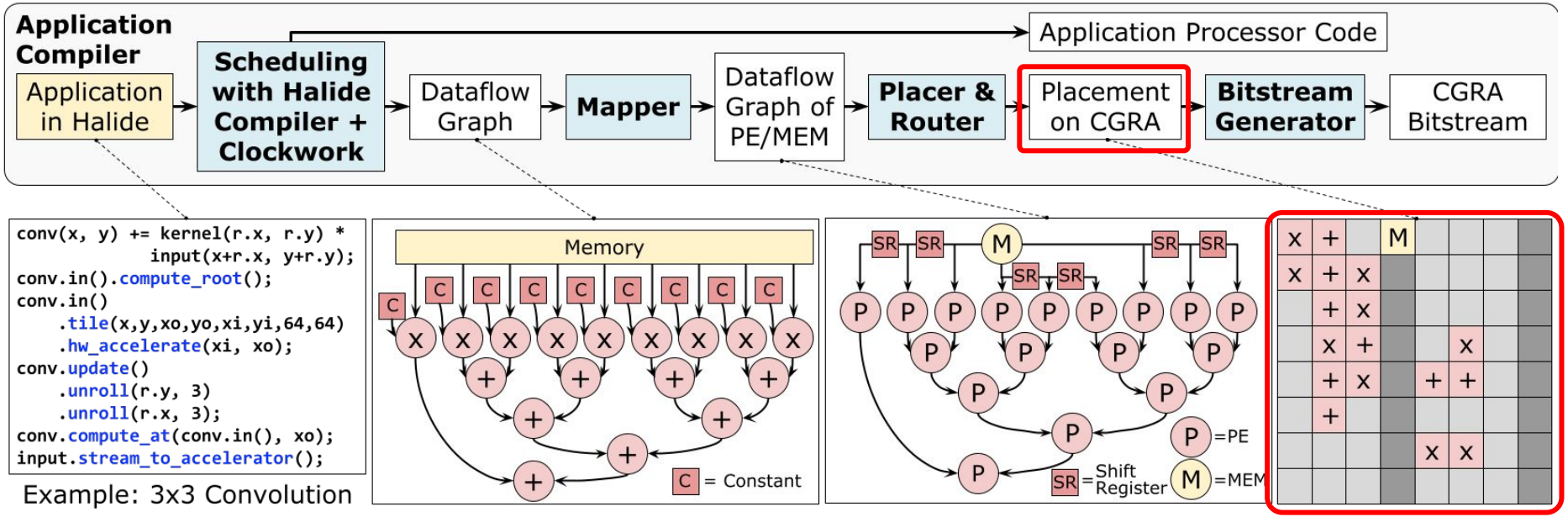
X Operation Occurs

Operation	Critical Path(ns)
5 Adds	5*.52
11 SwitchBox	11*0.28
Sum	4.14

Critical Path Model



Critical Path Model

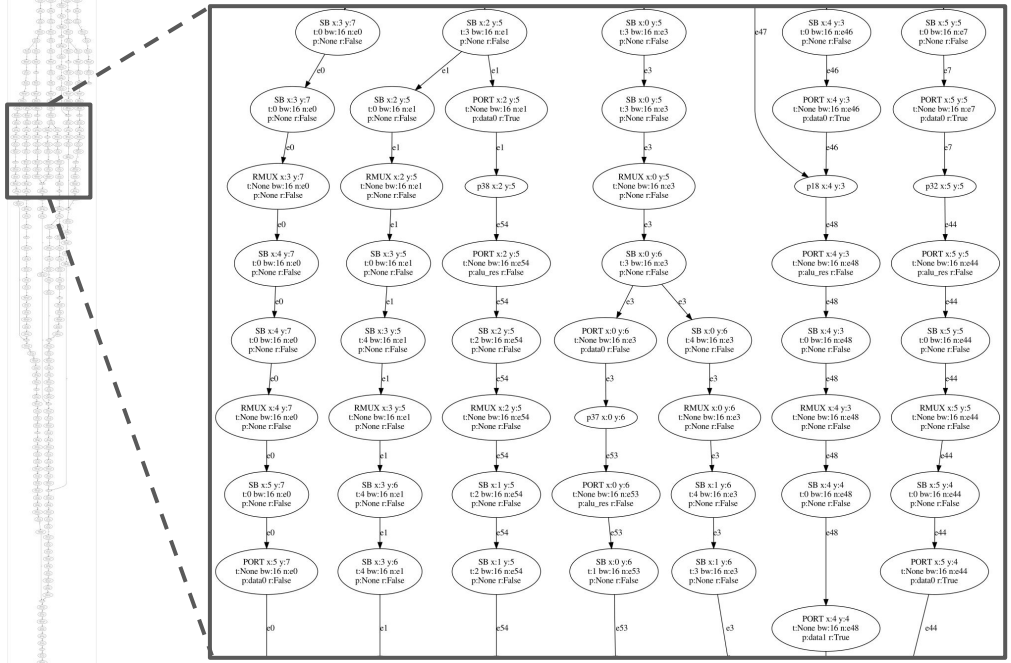


We must construct the delay model using the output of the place and route tool

Critical Path Model

- We can represent our applications as graphs composed of place and route primitives
  - PEs
  - MEMs
  - I/Os
  - Switch boxes
  - Register MUXs
  - Pipelining registers
  - Ports
- Then we can do static timing analysis to determine the critical path


i10
i9
r43
r44
m42
p34
r45
m8
p18
p33
p16
p17
p21
p19
p38
p15
p32
p37
p35
r1
p36
p20
r66
p39
p40
r65
p22
p14
r62
r63


SB x:3 y:7 t:0 bw:16 n:e0 p:None r:False
SB x:2 y:5 t:3 bw:16 n:e1 p:None r:False
SB x:0 y:5 t:3 bw:16 n:e3 p:None r:False
SB x:4 y:3 t:0 bw:16 n:e46 p:None r:False
SB x:5 y:5 t:0 bw:16 n:e7 p:None r:False
e0
e1
e3
e46
e7
SB x:3 y:7 t:0 bw:16 n:e0 p:None r:False
SB x:2 y:5 t:0 bw:16 n:e1 p:None r:False
PORT x:2 y:5 t:None bw:16 n:e1 p:data0 r:True
SB x:0 y:5 t:3 bw:16 n:e3 p:None r:False
PORT x:4 y:3 t:None bw:16 n:e46 p:data0 r:True
PORT x:5 y:5 t:None bw:16 n:e7 p:data0 r:True
e47
RMUX x:3 y:7 t:None bw:16 n:e0 p:None r:False
RMUX x:2 y:5 t:None bw:16 n:e1 p:None r:False
p38 x:2 y:5
RMUX x:0 y:5 t:None bw:16 n:e3 p:None r:False
p18 x:4 y:3
p32 x:5 y:5
e48
e54
SB x:4 y:7 t:0 bw:16 n:e0 p:None r:False
SB x:3 y:5 t:0 bw:16 n:e1 p:None r:False
PORT x:2 y:5 t:None bw:16 n:e54 p:alu_res r:False
SB x:0 y:6 t:3 bw:16 n:e3 p:None r:False
PORT x:4 y:3 t:None bw:16 n:e48 p:alu_res r:False
PORT x:5 y:5 t:None bw:16 n:e44 p:alu_res r:False
SB x:4 y:7 t:0 bw:16 n:e0 p:None r:False
SB x:3 y:5 t:4 bw:16 n:e1 p:None r:False
SB x:2 y:5 t:2 bw:16 n:e54 p:None r:False
PORT x:0 y:6 t:None bw:16 n:e3 p:data0 r:False
SB x:0 y:6 t:4 bw:16 n:e3 p:None r:False
SB x:4 y:3 t:0 bw:16 n:e48 p:None r:False
SB x:5 y:5 t:0 bw:16 n:e44 p:None r:False
RMUX x:4 y:7 t:None bw:16 n:e0 p:None r:False
RMUX x:3 y:5 t:None bw:16 n:e1 p:None r:False
RMUX x:2 y:5 t:None bw:16 n:e54 p:None r:False
p37 x:0 y:6
RMUX x:0 y:6 t:None bw:16 n:e3 p:None r:False
RMUX x:4 y:3 t:None bw:16 n:e48 p:None r:False
RMUX x:5 y:5 t:None bw:16 n:e44 p:None r:False
SB x:5 y:7 t:0 bw:16 n:e0 p:None r:False
SB x:3 y:6 t:4 bw:16 n:e1 p:None r:False
SB x:1 y:5 t:2 bw:16 n:e54 p:None r:False
PORT x:0 y:6 t:None bw:16 n:e53 p:alu_res r:False
SB x:1 y:6 t:4 bw:16 n:e3 p:None r:False
SB x:4 y:4 t:0 bw:16 n:e48 p:None r:False
SB x:5 y:4 t:0 bw:16 n:e44 p:None r:False
e53
PORT x:5 y:7 t:None bw:16 n:e0 p:data0 r:False
SB x:3 y:6 t:4 bw:16 n:e1 p:None r:False
SB x:1 y:5 t:2 bw:16 n:e54 p:None r:False
SB x:0 y:6 t:1 bw:16 n:e53 p:None r:False
SB x:1 y:6 t:3 bw:16 n:e3 p:None r:False
PORT x:4 y:4 t:None bw:16 n:e48 p:data1 r:True
PORT x:5 y:4 t:None bw:16 n:e44 p:data0 r:True

Static Timing Analysis

- Arrival time at any node N:
  - arrival[N] = delay[N] + max(arrival[predecessors[N]])

- Simple algorithm for computing arrival time at every node
  - Use a topological ordering to traverse the DAG
  - Calculate arrival time delay[N] + max(arrival[predecessors[N]])

Critical Path Model - Evaluation

Application	Modeled (MHz)	Measured (MHz)	% Difference
Gaussian v1	158	200	23.5%
Gaussian v2	295	280	5.2%
Gaussian v3	515	420	20.3%
Gaussian v4	793	600	27.7%
Harris v1	30	25	18.2%
Harris v2	137	160	15.5%
Harris v3	335	300	11.0%
Harris v4	373	360	3.5%

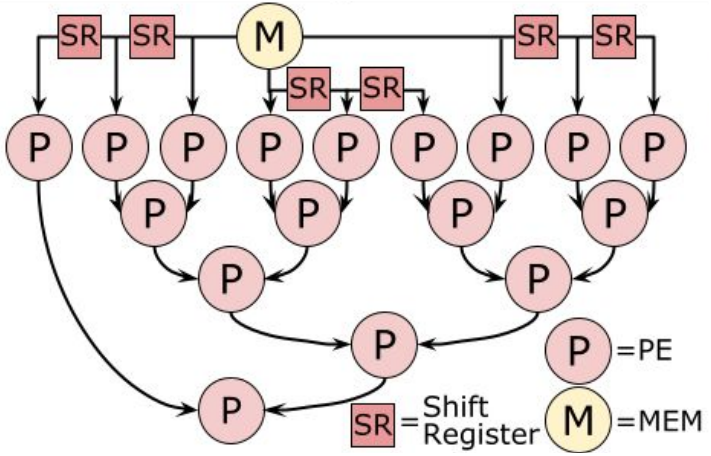
Pipelining Approaches

1. Pipeline at the compute mapping stage
    - Insert pipeline regs before every PE
    - Before/after every output/input IO

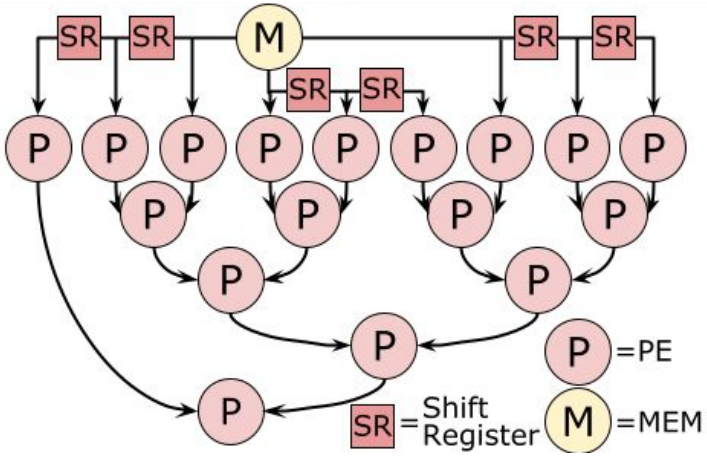
2. Pipeline at the place and route stage
    - Iteratively break the critical path determined by the STA tool
    - Re-analyze and determine new critical path
    - Continue until target is met

Compute Pipelining

- At compute mapping, we know how many PEs we will use and how they are connected
- Have all information needed to do branch delay matching
  - Ensures all paths from one memory to another are the same number of cycles
- All registers are added to the compute graph
  - Need to be packed into the PEs or placed onto the routing fabric



Compute Pipelining



Benefits:

- Easy
- Branch delay matching is simple

Drawbacks:

- No routing information
- No real information about which paths will be longest
- Ignores lots of routes that aren’t represented in the compute kernels

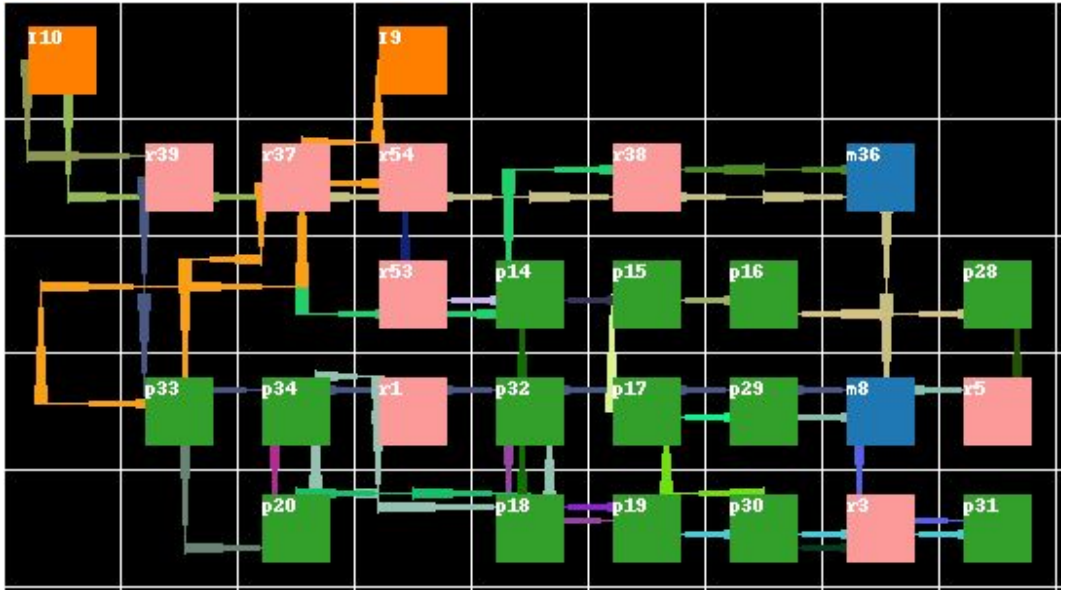
Compute Pipelining Results

Application	Before Pipelining	After Pipelining
gaussian	68MHz	180 MHz
harris color	25MHz	160 MHz
unsharp	18MHz	80 MHz
camera pipeline	37Mhz	70 MHz

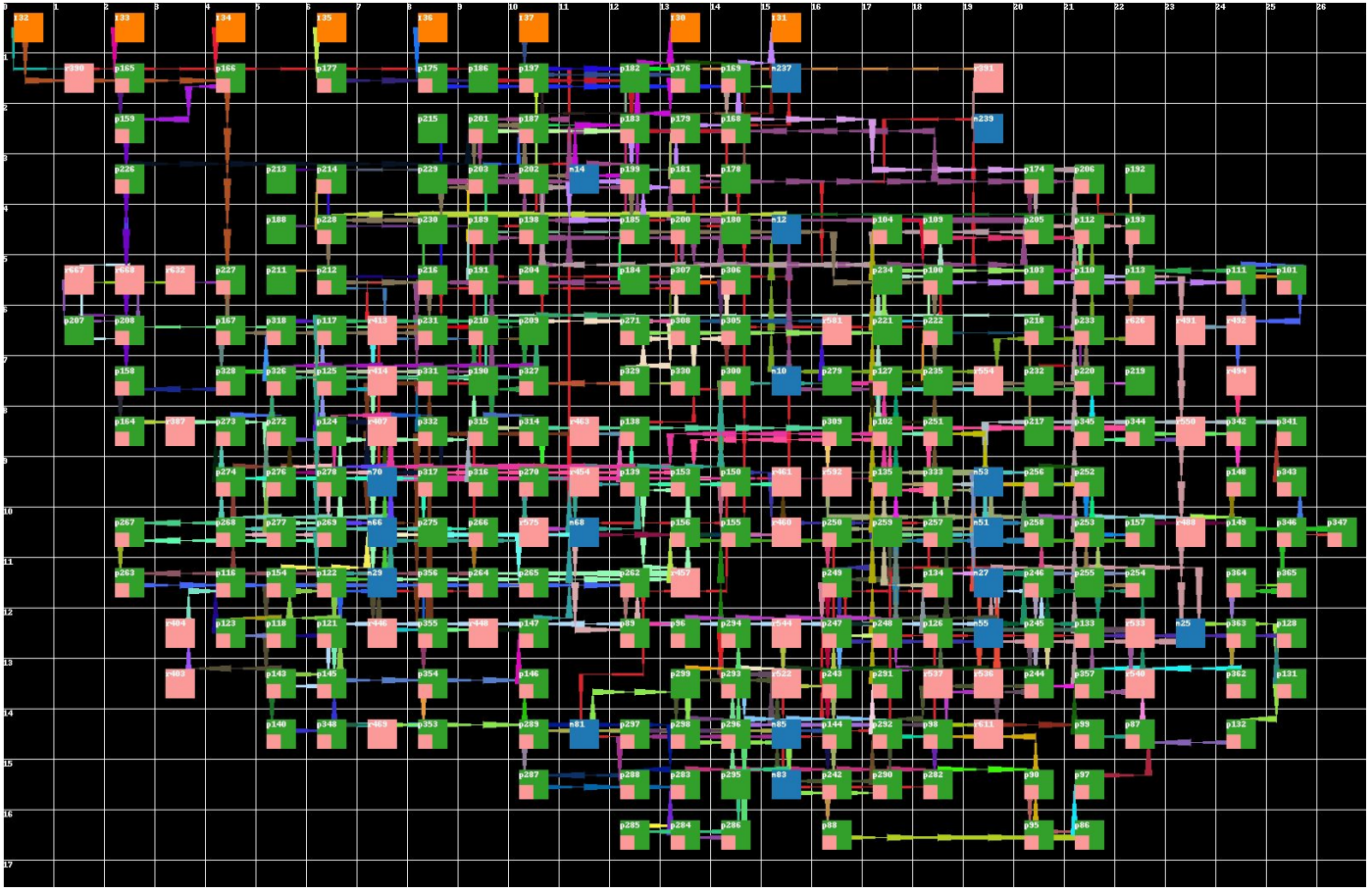
- Every application got faster, but some didn't improve much
- We expect much larger improvements
- Not close to the 500 MHz target
- Compute pipelining is not good enough

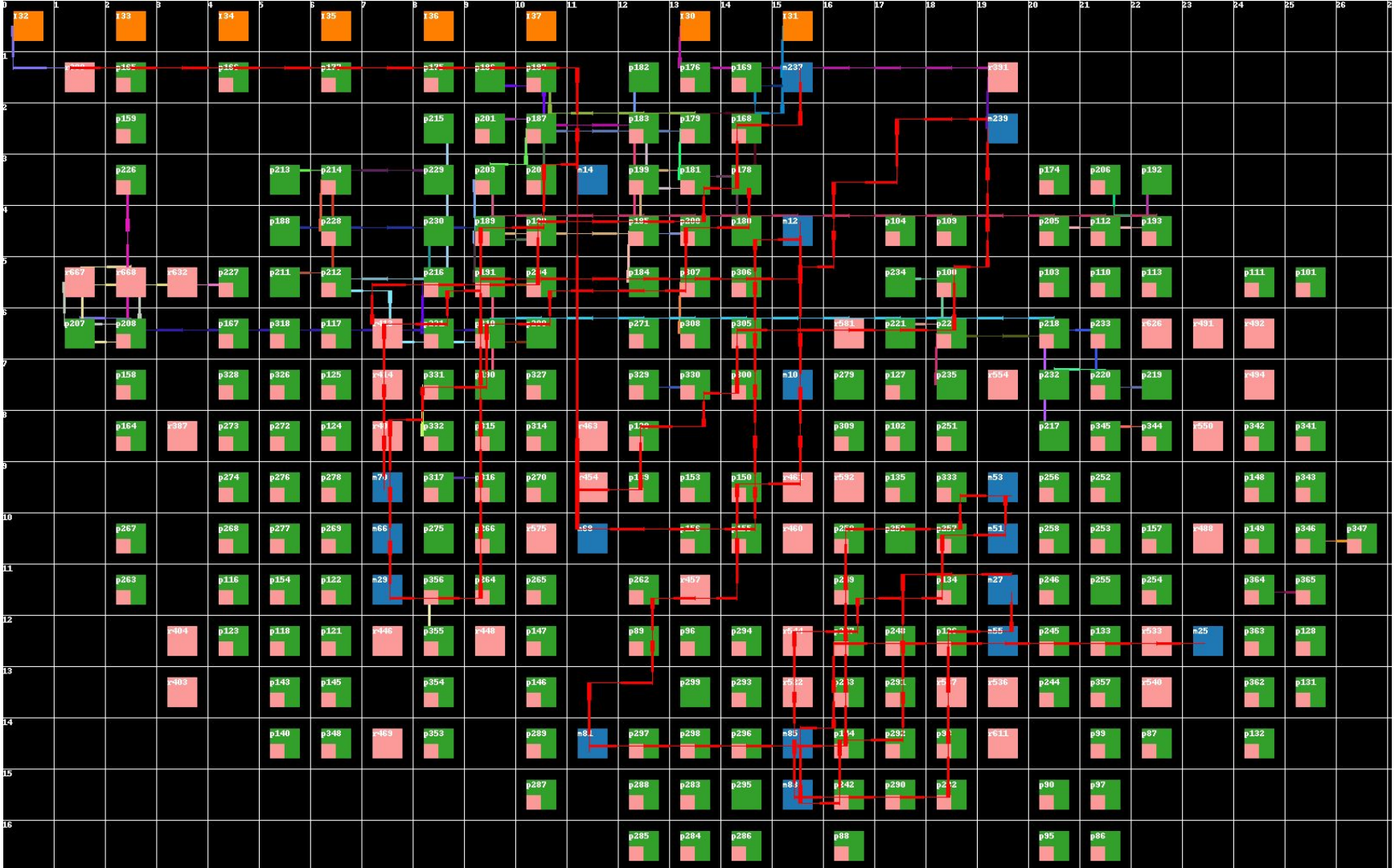
Compute Pipelining Issues

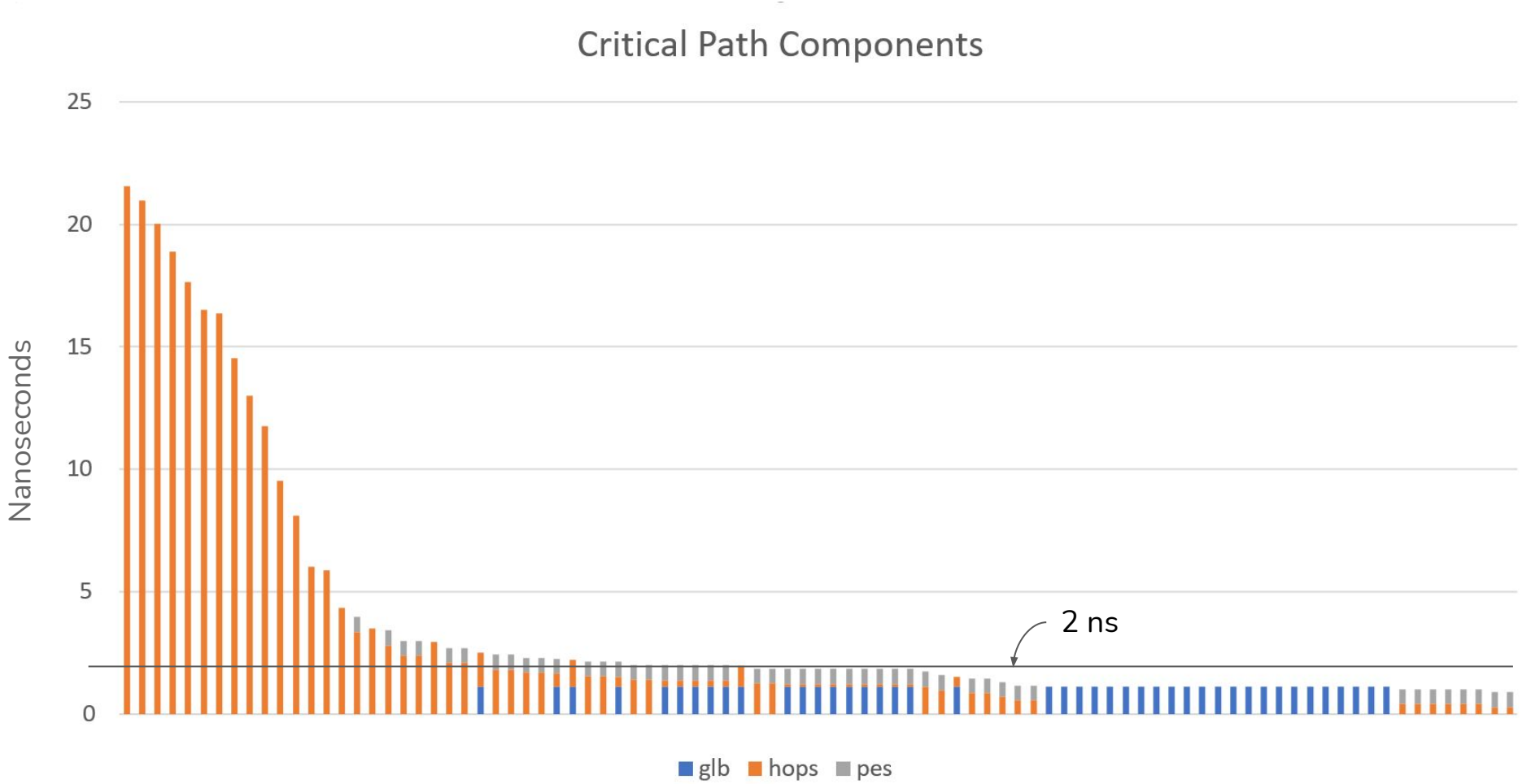
- Place and route result gets worse with more pipeline registers
  - Competing objectives: adding more pipelining to improve delay while keeping the route reasonable



Application	Before Pipelining	After Pipelining
harris color	25MHz	160 MHz






Critical Path Components
25
20
15
10
5
0
Nanoseconds
2 ns
glb
hops
pes

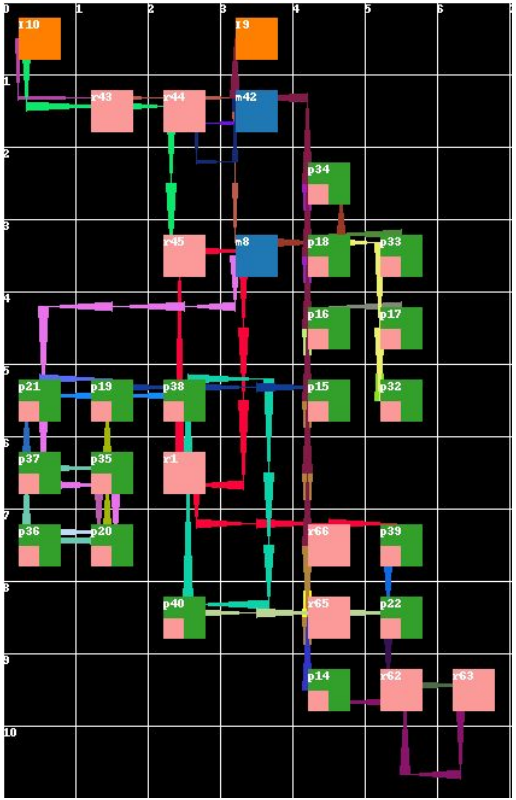
Place and Route Pipelining

Benefits:

- Have a much better idea of where delay is in the application
- Should be able to pipeline much more intelligently

Drawbacks:

- Adding pipelining registers to a routing result is not always possible
- If a route is short or contains no switch boxes the pipelining fails



Place and Route Pipelining Results

- This approach fails for every application that we tried
- Finding available registers is very difficult and ripping up and rerouting is hard

- Need a third solution to the pipelining problem

Combined Approach

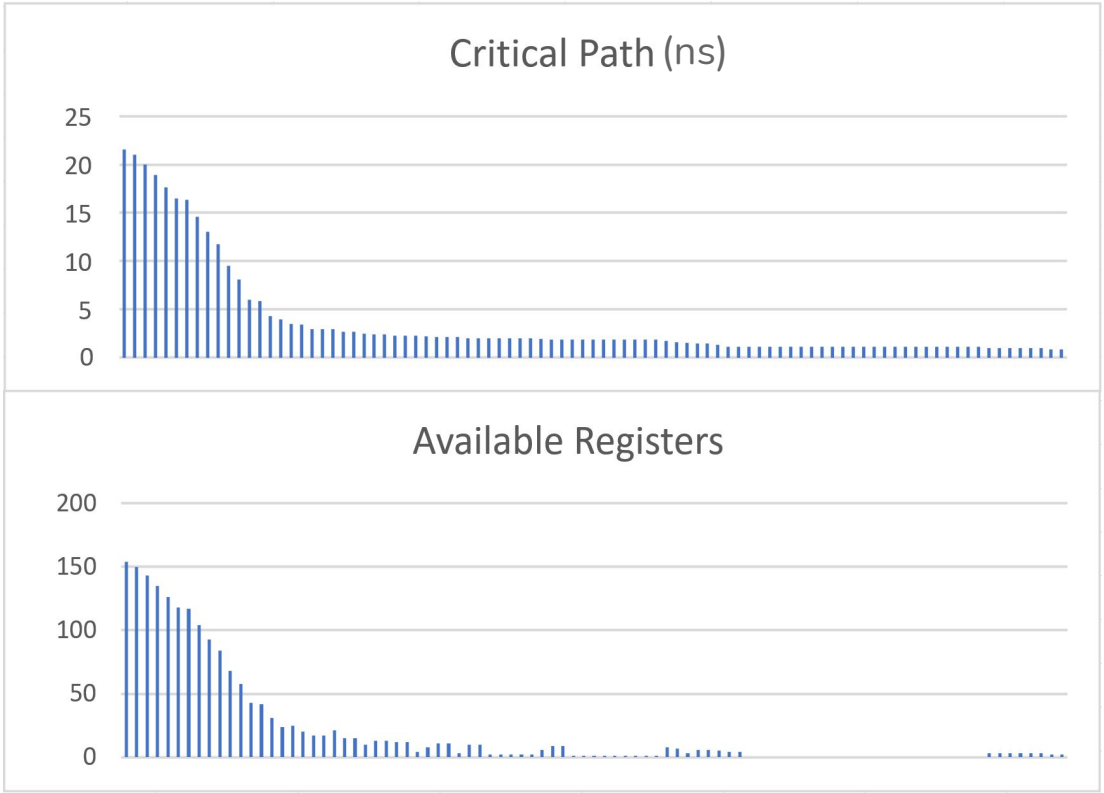
1. Pipeline application during the compute mapping stage
2. Schedule and do memory mapping
3. Place and route application
4. Do STA analysis and pipeline place and route result
5. Schedule application memories again to account for added delay
6. Generate bitstream

Combined Approach

- Overcomes issues of compute mapping pipelining approach:
  - Actual route information is available
- Overcomes issues of place and route pipelining approach:
  - Compute kernels are fully pipelined before the routing is fixed

- Potential issues:
  - Adding pipelining registers to a routing result is not always possible

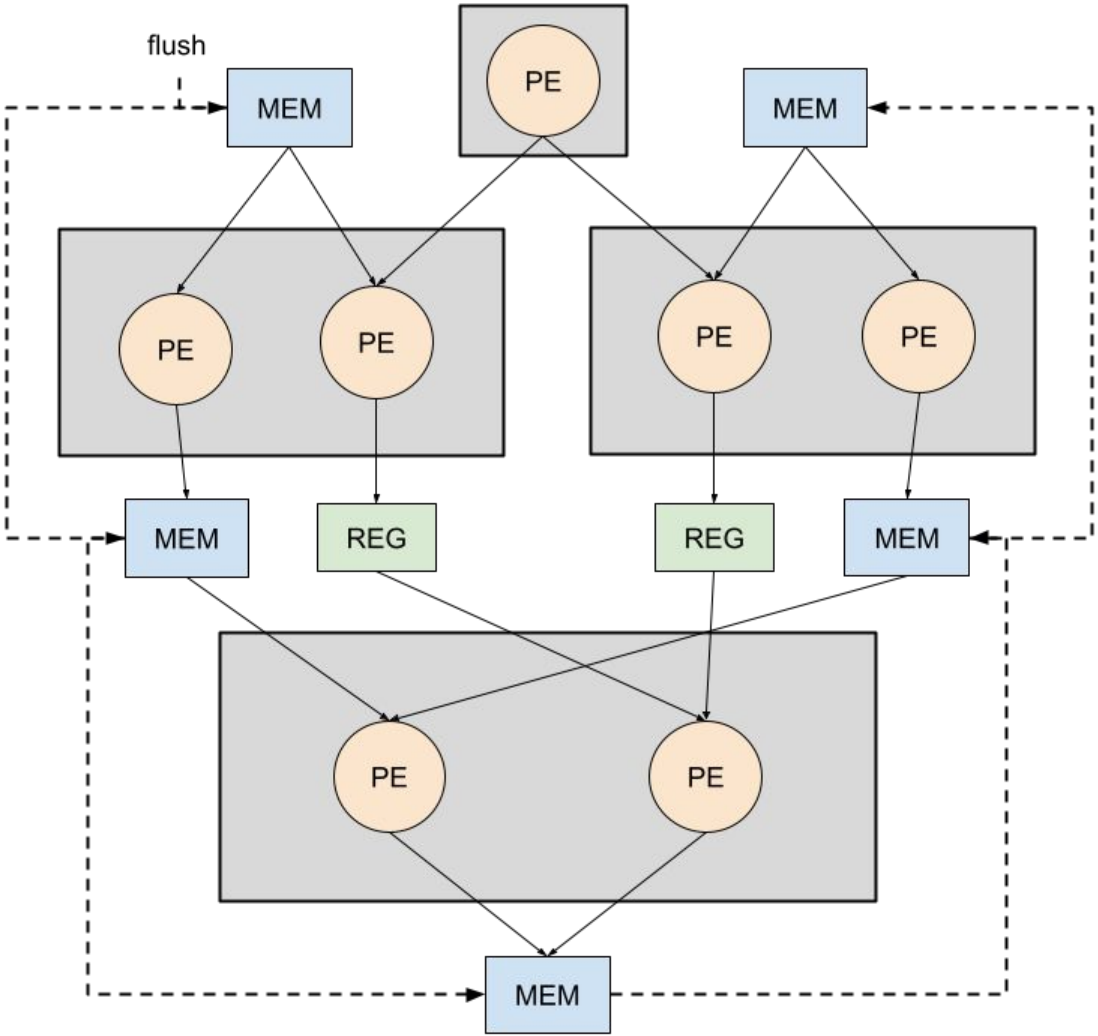
Combined Approach - Potential Issues



- In practice, long critical paths have lots of available registers

Combined Approach - Implementation Issues

- Branch delay matching and rescheduling the applications is hard
  - If you add a register to a compute kernel:
    - Branch delay match within that kernel
    - Possibly branch delay match the whole application too
    - Tell scheduling tool about the new delay
  - If you add a register outside of a compute kernel:
    - Branch delay match the whole application
    - Fix schedules post-scheduling


flush
MEM
PE
MEM
PE
PE
PE
PE
MEM
REG
REG
MEM
PE
PE
MEM

Combined Approach Results - Gaussian

	Modeled (MHz)	Measured (MHz)
Baseline	158	200
Pipelined v1	295	280
Pipelined v2	360	320
Pipelined v3	515	420
Pipelined v5	793	600

- Combined approach let us hit our 500 MHz target
- 10 registers were added post-PnR to hit 600 MHz

Combined Approach Results - Harris

	Modeled (MHz)	Measured (MHz)
Baseline	30	25
Pipelined v1	137	160
Pipelined v2	166	200
Pipelined v3	335	300
Pipelined v4	373	360

- 5 pipelining registers are added to hit 360 MHz
- Working on getting application working at higher frequencies

Takeaways

- An important comparison measurement for our chip is EDP (energy delay product)
  - (power x delay) x delay
  - Delay is extremely important to getting low EDP
- Our chip can run up to 780 MHz, so pipelining is critically important
- Compute pipelining and place and route alone are not good enough
- Combined approach results in promising results
  - Need to fix MetaMapper flow for all applications
  - Need to fix all pipelining branch delay matching problems