Stanford | ENGINEERING
Electrical Engineering

STANFORD DAWN

Software-Defined Hardware Without Sacrificing Performance

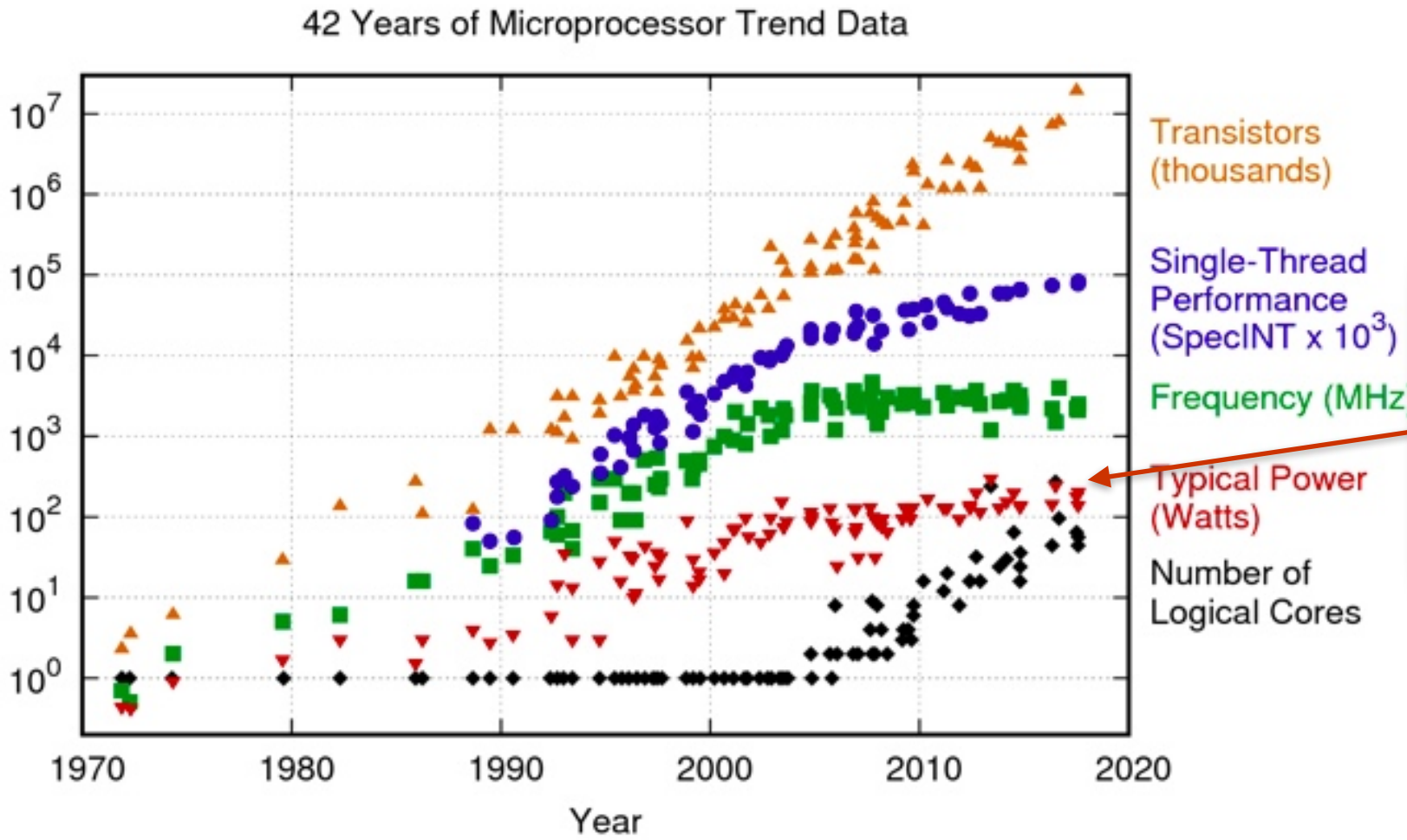
Matthew Feldman

Slides:
https://tinyurl.com/yafqu5km

Introduction

The transition from instruction-based architectures to custom hardware

The CPU Power Wall



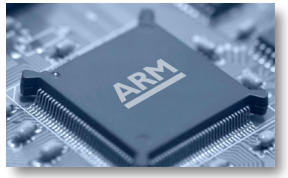
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

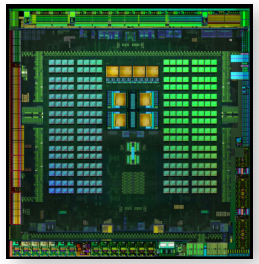
Packing more transistors onto a chip will no longer work because of limits on power delivery and heat dissipation.

Compute Devices at a Glance

Energy-optimized CPU



Throughput-Oriented Device GPU

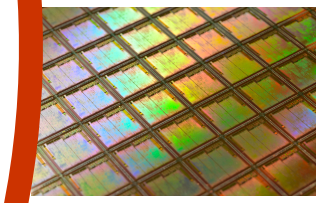


Programmable DSP

HEXAGON

Programmable Logic FPGA

Application-Specific Integrated Circuit (ASIC)



Least efficient
Easiest to program
Least expensive

Specialization is the key

Hardest to program

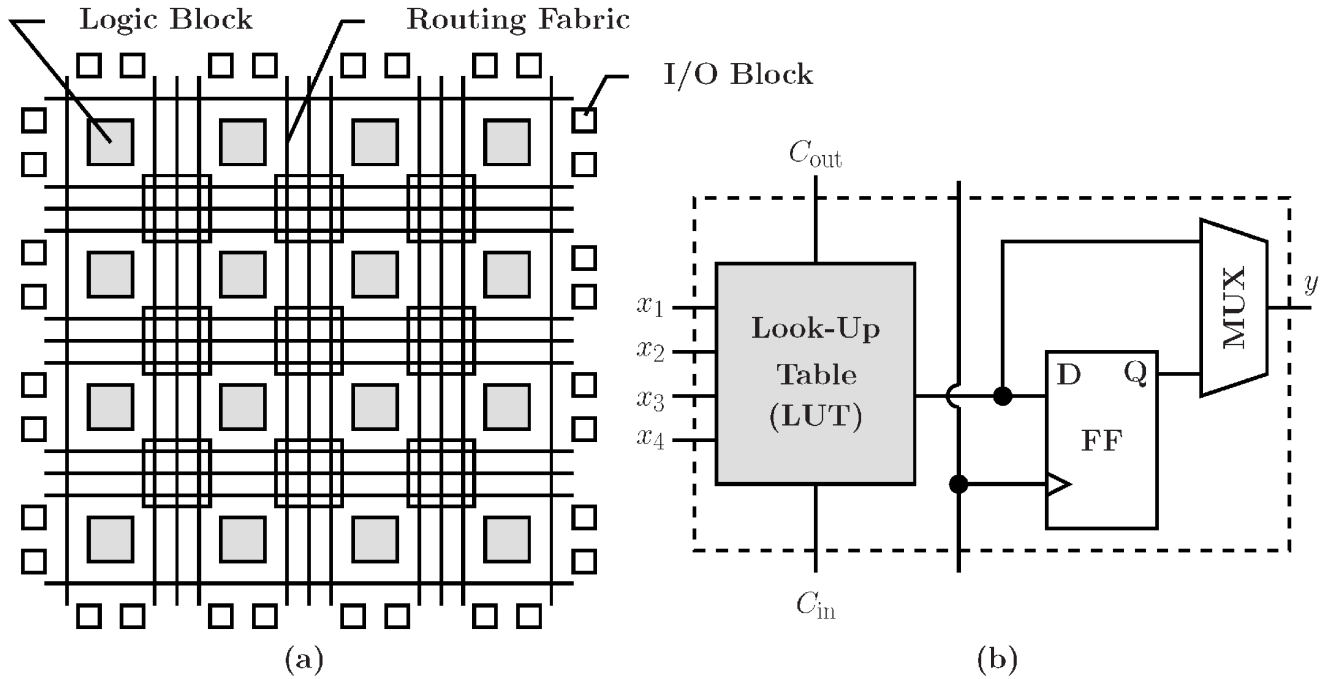
Most efficient
Not programmable
Most expensive

We are going to focus here

Credit: CS149

FPGA Crash Course

- Field-programmable gate array
  - Reconfigurable logic device consisting of
    - On-chip Memory (BRAMs) - ~10s Mb
    - Logic Cells (LUTs + FFs) - ~1M
    - Processing blocks (DSPs) - ~1000s

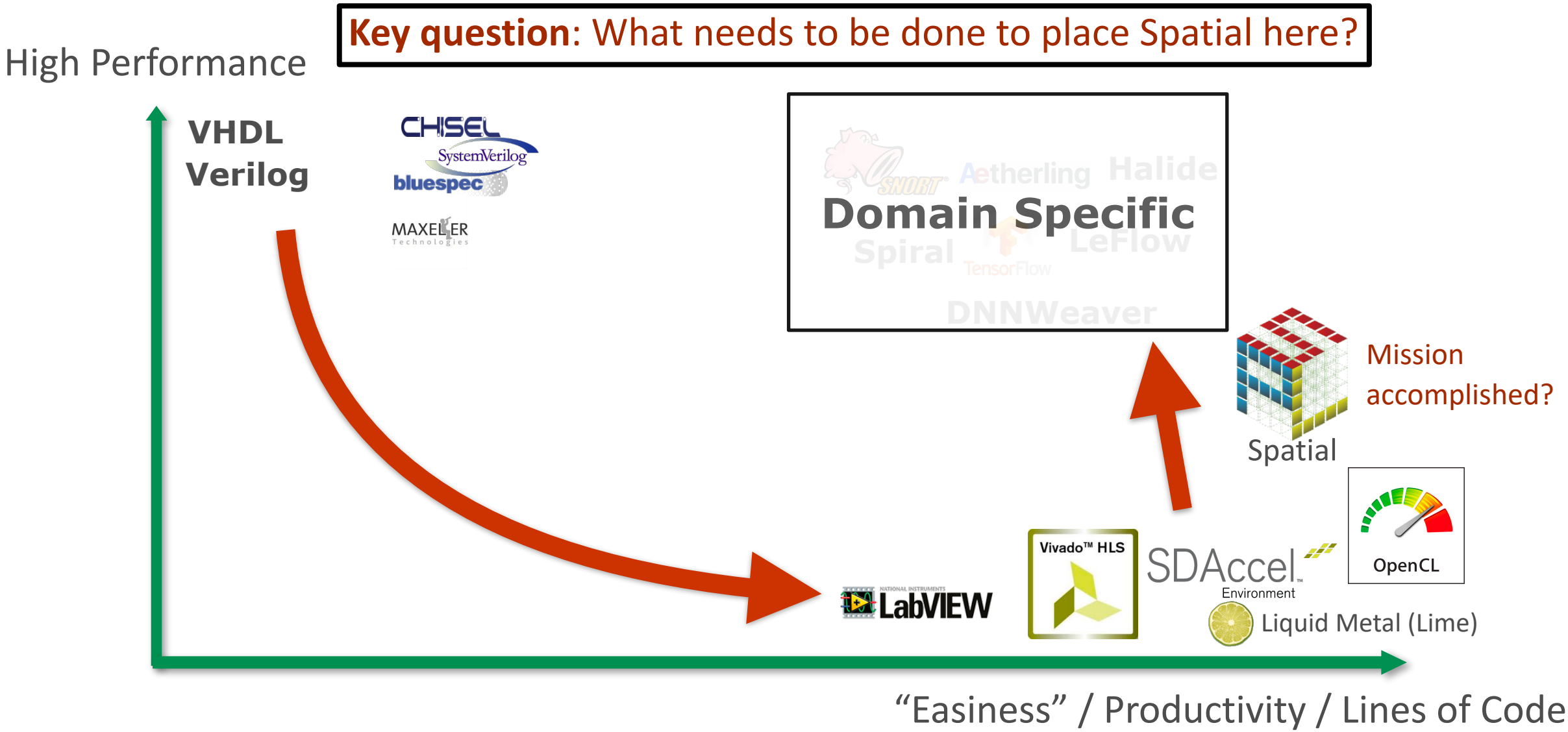


You want to design an accelerator for your next algorithm…

Which language should you use? [1]

[1] N. Kapre et al, "Survey of domain-specific languages for FPGA computing," FPL 2016

Choosing a Language



Hardware Design At a Glance

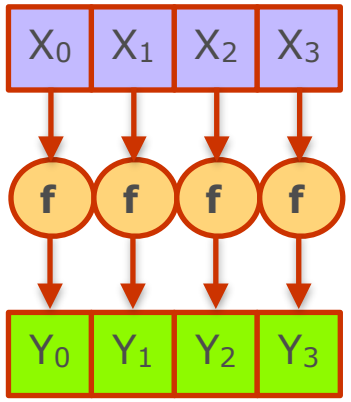
- To create a good accelerator, the designer must **optimize computation and memory accesses** in order to **keep all parts of the circuit active at all times**
- In order to do this, the designer can make decisions about
  - **Parallelism** - Run operations concurrently
  - **Data Locality** - Manually manage on-chip scratchpads
  - **Control Flow** - Orchestrate how loops execute relative to each other

Key Contributions in This Talk

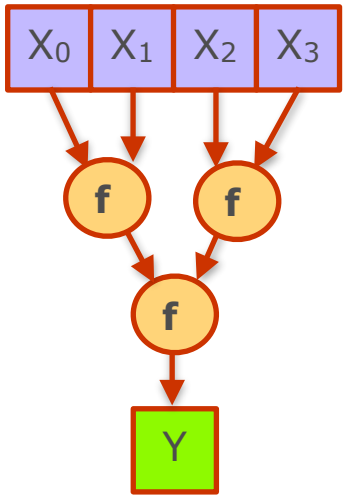
- **Problem**: Expressing hardware at a high level results in sub-optimal designs
- **Solution**: Design a language and compiler that provides
  - the **right software abstractions**
  - A **performance debugging environment** that lends itself to a **set of optimization principles**
  - Hardware-oriented **optimizations for memory partitioning**
- **Validation**: Demonstrate effectiveness on a real-world problem

Parallel Patterns

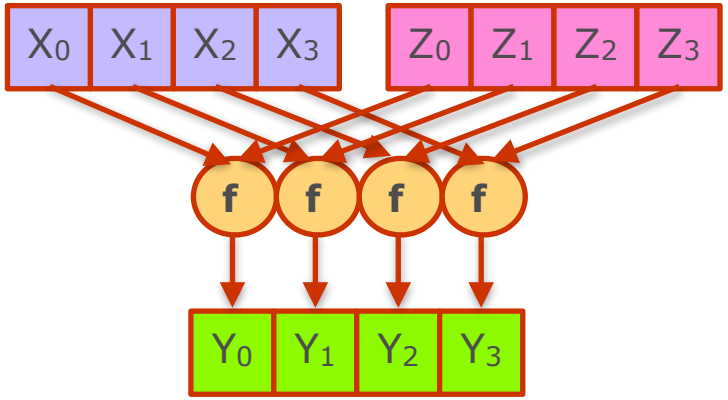
- Parallel patterns are loop abstractions with implicit information about parallelism and access patterns



Map
Element-wise function *f*



Reduce
Combine elements with (associative) function *f*



Zip
Element-wise combine function *f*

- Spatial is an *imperative language* that is designed to easily capture **parallel patterns**[2]

[2] R. Prabhakar et al. "Generating Configurable Hardware from Parallel Patterns." ASPLOS 2016

How Loops Map to Hardware

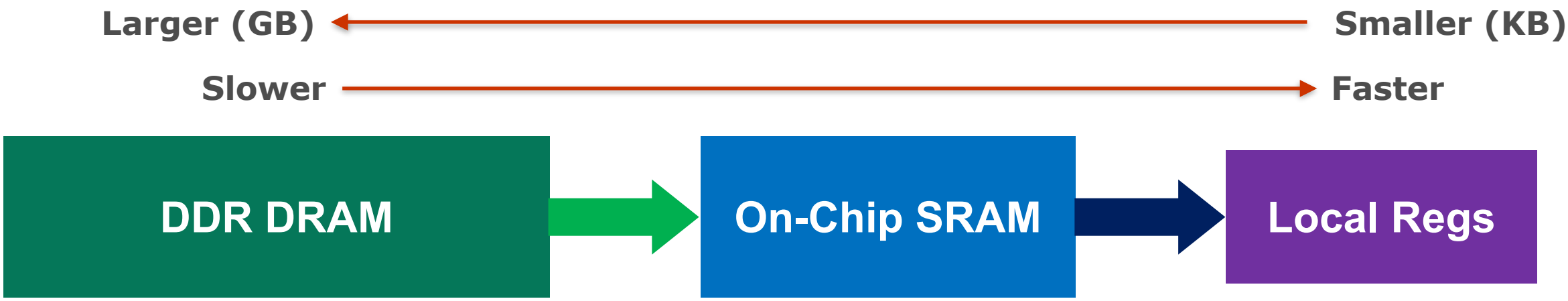
- A software "loop" is **counter chain + controller**
  - **Counter chain** - Collection of iterators that are chained together
  - **Controller** - A container for a data path or other controllers

- Controllers are nested:
  - **Inner -** contains datapaths of *only* primitive nodes
  - **Outer -** contains *only* other controllers (called "children")

```
Foreach(N by 1) { i =>  // Outer controller
    Foreach(M by 1) { j => mem(i,j) = i+j }              // Inner controller
    Foreach(P by 1) { j => if (j == 0) … = mem(i,j) }    // Inner controller
}
```

How Arrays Map to Hardware

- Each “array” is manually placed on a resource:
  - DRAM - off-chip, burst-addressable
  - SRAM - on-chip, word-addressable
  - Register - on-chip, single word



Key Contributions in This Talk

- **Problem**: Expressing hardware at a high level results in sub-optimal designs
- **Solution**: Design a language and compiler that provides
  - the **right software abstractions**
  - A **performance debugging environment** that lends itself to a **set of optimization principles**
  - Hardware-oriented **optimizations for memory partitioning**
- **Validation**: Demonstrate effectiveness on a real-world problem

A Deep Dive Into Performance

Performance Analysis

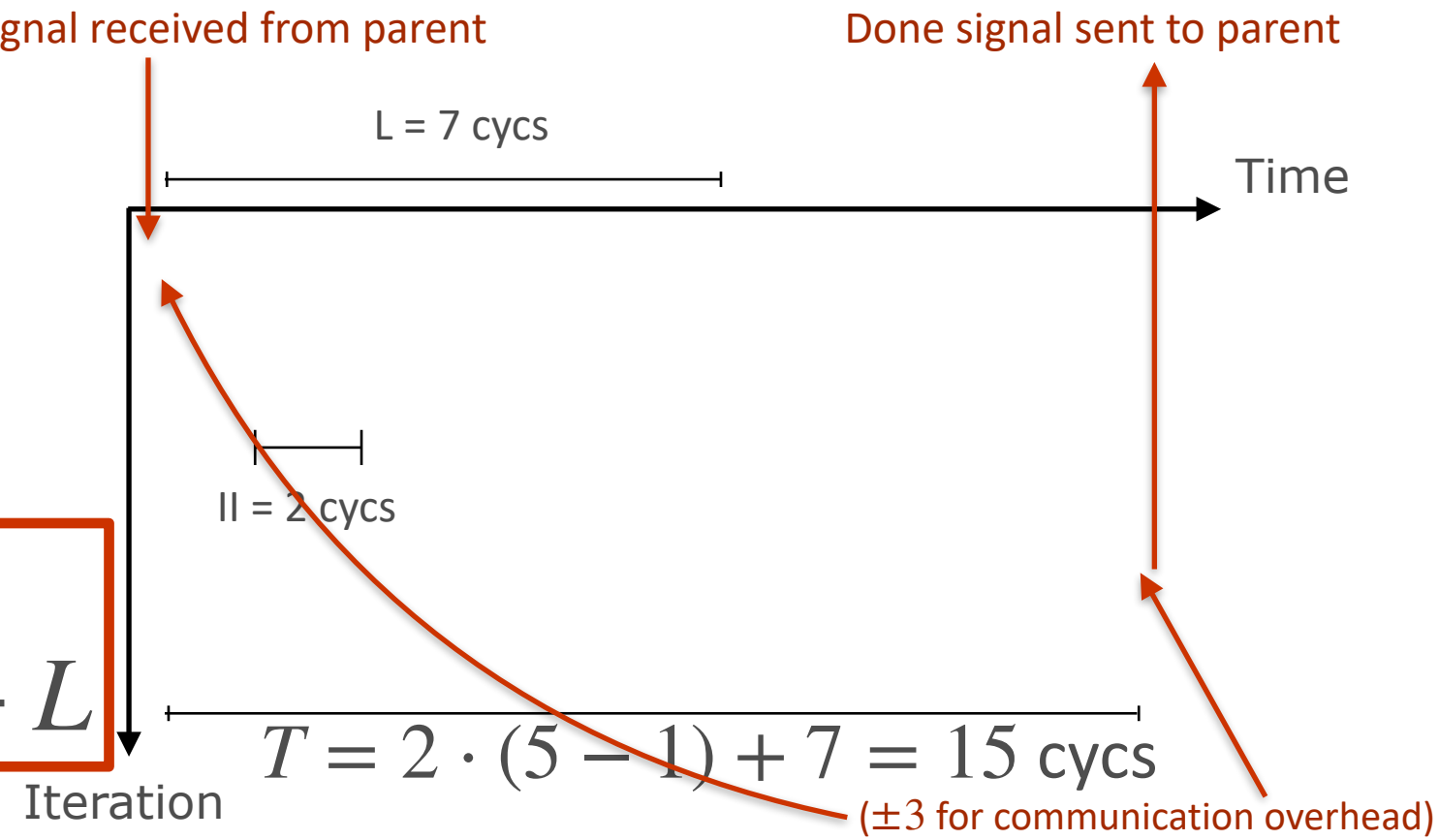
- The runtime of a controller (**T**) depends on its latency (**L**), initiation interval (**II**), and number of iterations (**iters**)

```
Foreach(5 by 1) {i =>
  … // L = 7, II = 2
}
```

Abstract Example

Key Equation:

$$T = II \cdot (iters - 1) + L$$



Performance Analysis

- **Initiation Interval** (**II**) and **Latency** (**L**) can be tricky to nail down.

```
Foreach(5 by 1) {i =>
  … // L = 7, II = 2
}
```

For **inner** controllers, II and L are static and computed at compile-time based on the structure of the datapath

Key Equation:

$$T = II \cdot (iters - 1) + L$$

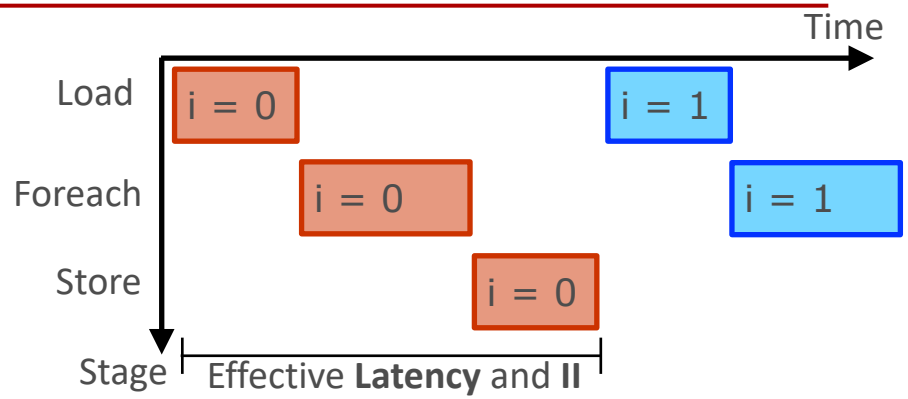
For **outer** controllers, we must approximate them based on the *schedule* of the controller

Outer Controller Scheduling

- **Outer controller** schedules are:
  - **Sequential** - No overlapping of child controllers
  - **Pipelined** - Coarse-grained overlapping of child controllers
  - **Stream** - Data-driven execution of child controllers
  - **Fork-Join** - Parallel execution of all child controllers
  - **Fork** - Selective execution of one child per-iteration

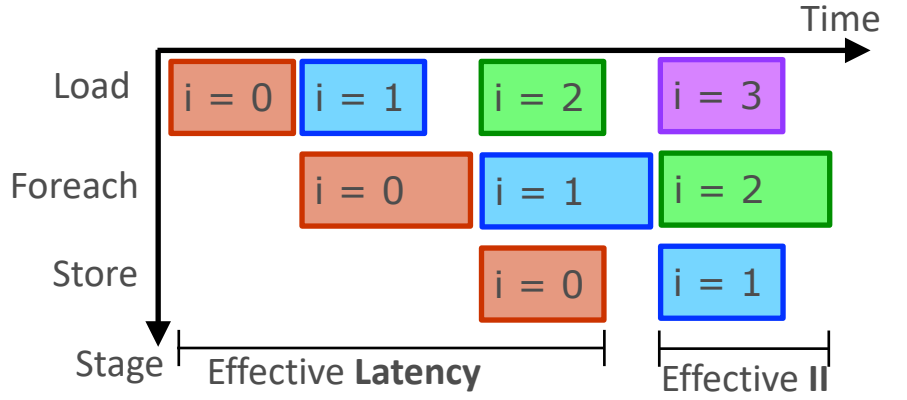
A Closer Look at Schedules

```
Sequential.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```

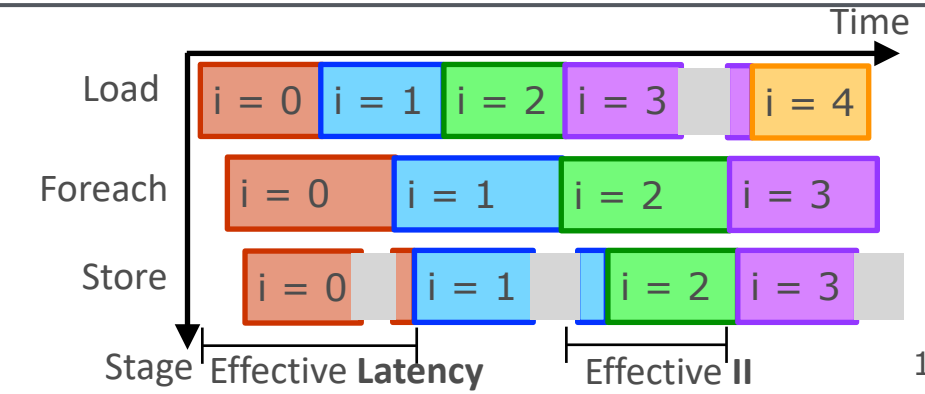


Note: **Foreach** with no annotation is implicitly "Pipelined"

```
Pipelined.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram2 store sram2
}
```



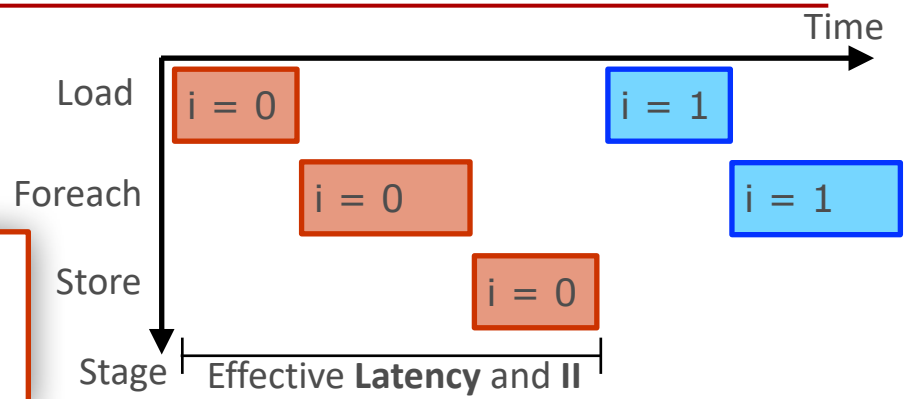

```
Stream.Foreach(…){i =>
  fifoIn load dram
  Foreach(M by 1){ j => fifoOut.enq(fifoIn.deq() * j) }
  dram2 store fifoOut
}
```



A Closer Look at Schedules

```
Sequential.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```

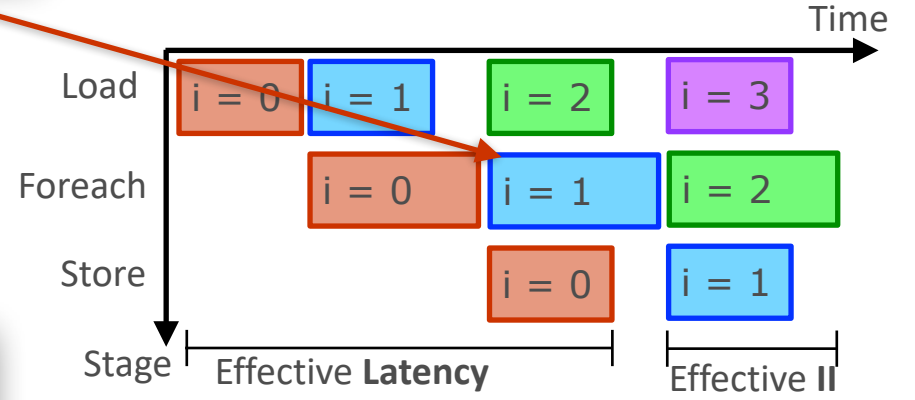
When the pipeline is full, it is in **steady-state** and the longest stage determines II



Note: **Foreach** with no annotation is implicitly "Pipelined"

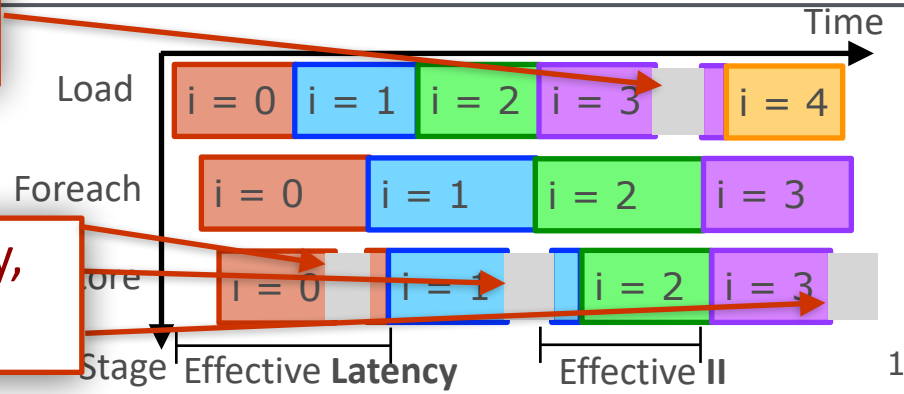
```
Pipelined.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram2 store sram2
}
```

When an intermediate FIFO is full, the producer stage is **stalled**.



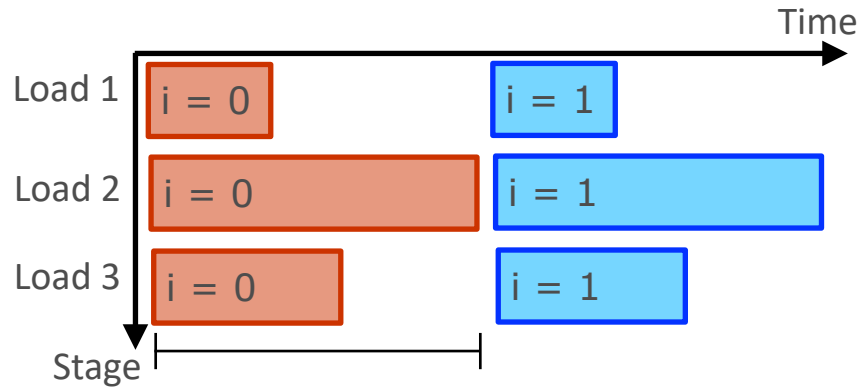

```
Stream.Foreach(…){i =>
  fifoIn load dram
  Foreach(M by 1){ j => fifoOut.enq(fifoIn.deq() * j) }
  dram2 store fifoOut
}
```

When an intermediate FIFO is empty, the consumer stage is **starved**.



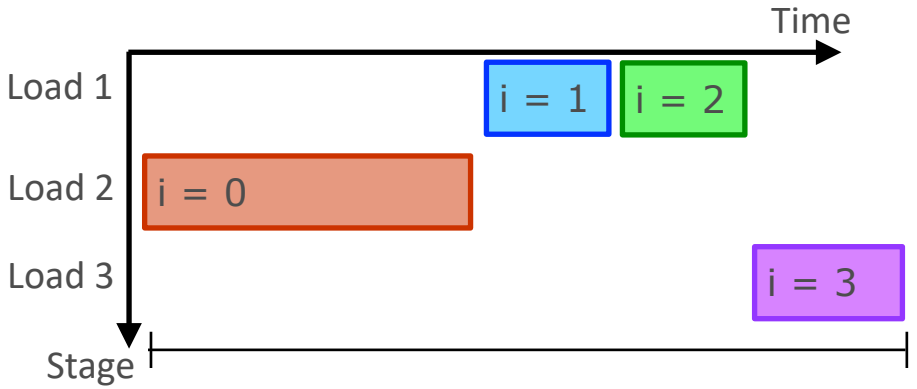
A Closer Look at Schedules

```
ForkJoin(N by 1){ i =>
  sram load dram
  sram2 load dram2
  sram3 load dram3
}
```



Effective **Latency** and **II**

```
Fork(N by 1){ i =>
   if (cond == 1) => sram load dram
   else if (cond == 2) => sram2 load dram2
   else sram3 load dram3
}
```



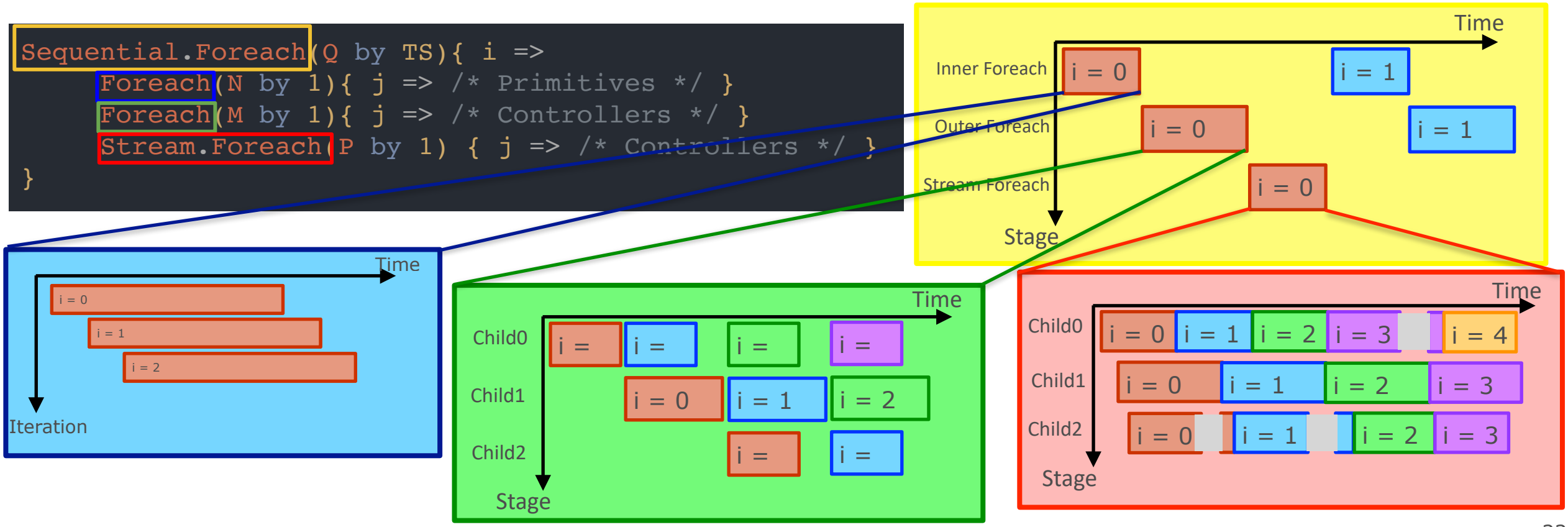
4x Effective **Latency** and **II**

Understanding the Hierarchy

Piecing the Hierarchy Together

- Consider the slice of a loop nesting with a parent Sequential Controller and three children.

```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Foreach(M by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



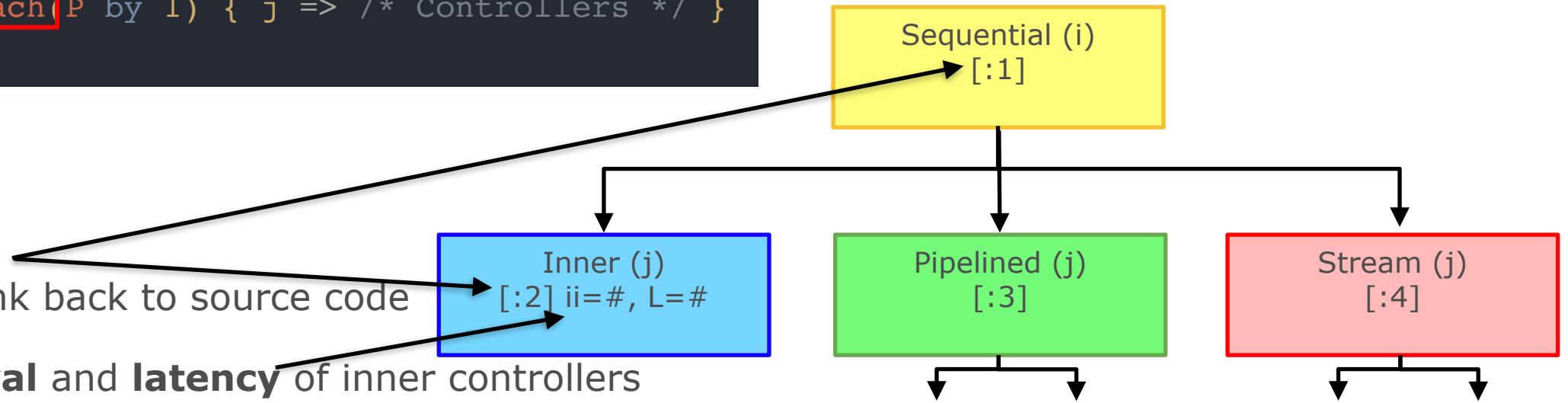
Key Contribution

- Thinking about performance this way quickly becomes very messy.
- **Controller hierarchy diagrams** are they key to achieving better performance
  - Generated and annotated automatically by Spatial
  - Convey detailed information about performance and resource utilization

Controller Hierarchy Diagrams

- Controller hierarchy diagrams distill the app into a human-readable structure

```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Foreach(M by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



Line numbers link back to source code

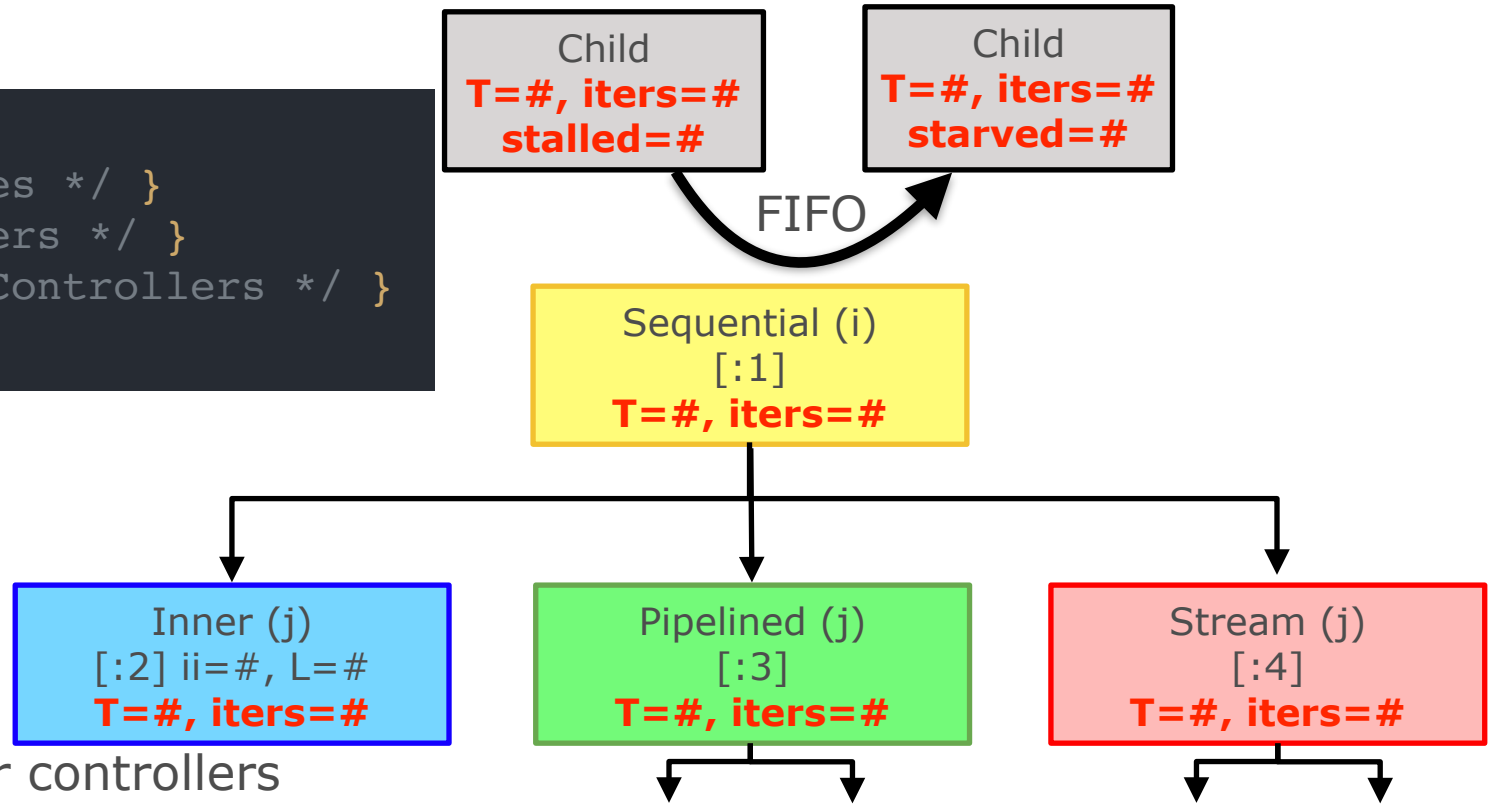
Initiation interval and **latency** of inner controllers

Controller Hierarchy Diagrams

- Controller hierarchy diagrams distill the app into a human-readable structure

Stream dependencies and **stall/starve** counts embedded

```
Sequential.Foreach(Q by TS){ i =>
      Foreach(N by 1){ j => /* Primitives */ }
      Foreach(M by 1){ j => /* Controllers */ }
      Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



KEY:
Black text - Compile-time property
Red text - Run-time property

Line numbers link back to source code

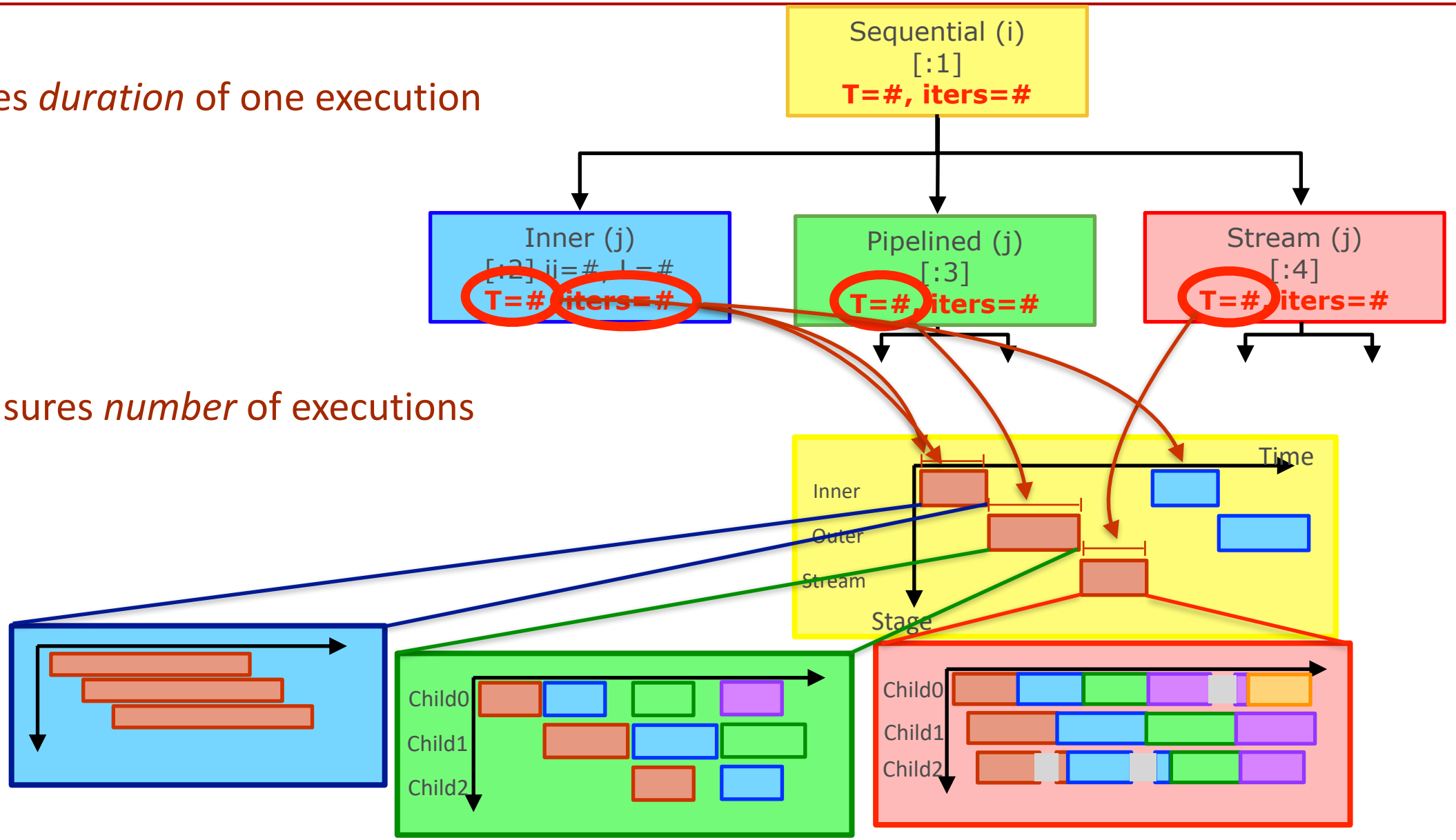
Initiation interval and **latency** of inner controllers

T and **iteration** counts automatically injected after running application

Connecting the Diagram to Waveforms

T measures *duration* of one execution

Iters measures *number* of executions



Design Concepts Captured by Hierarchy Diagrams

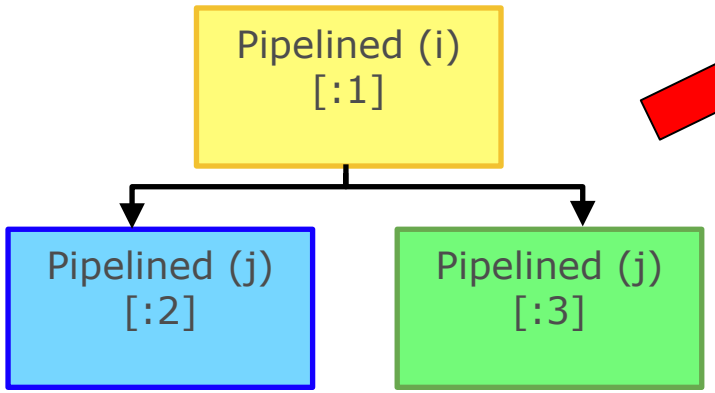
Parallelization

- There is an important distinction between **Inner** and **Outer** controller parallelization
  - **Inner controller** parallelization results in a *vectorized* datapath (i.e. SIMD parallelism)
  - **Outer controller** parallelization results in controller replication

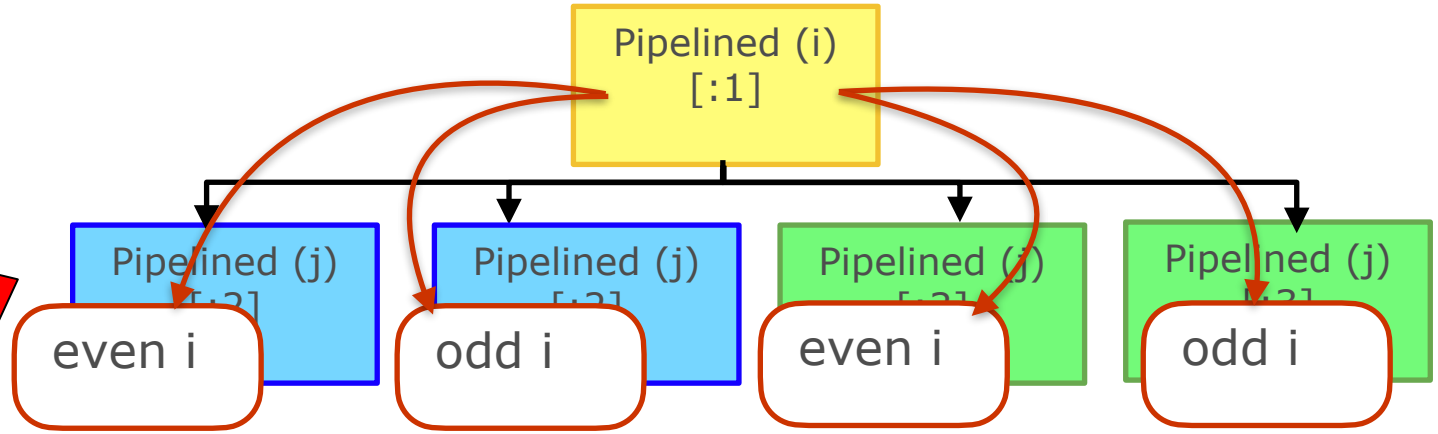
Outer Controller Parallelization

- Simply duplicating controllers does not fully exploit the parallelism

```
Foreach(N by 1 par 2){ i =>
  Foreach(…) {…}
  Foreach(…) {…}
}
```



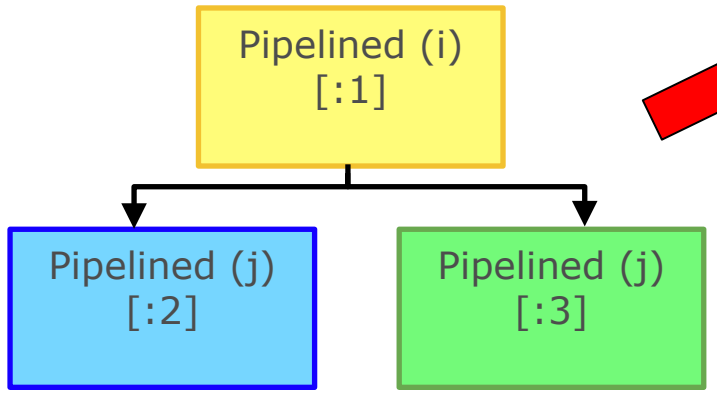
(Unparallelized Structure)



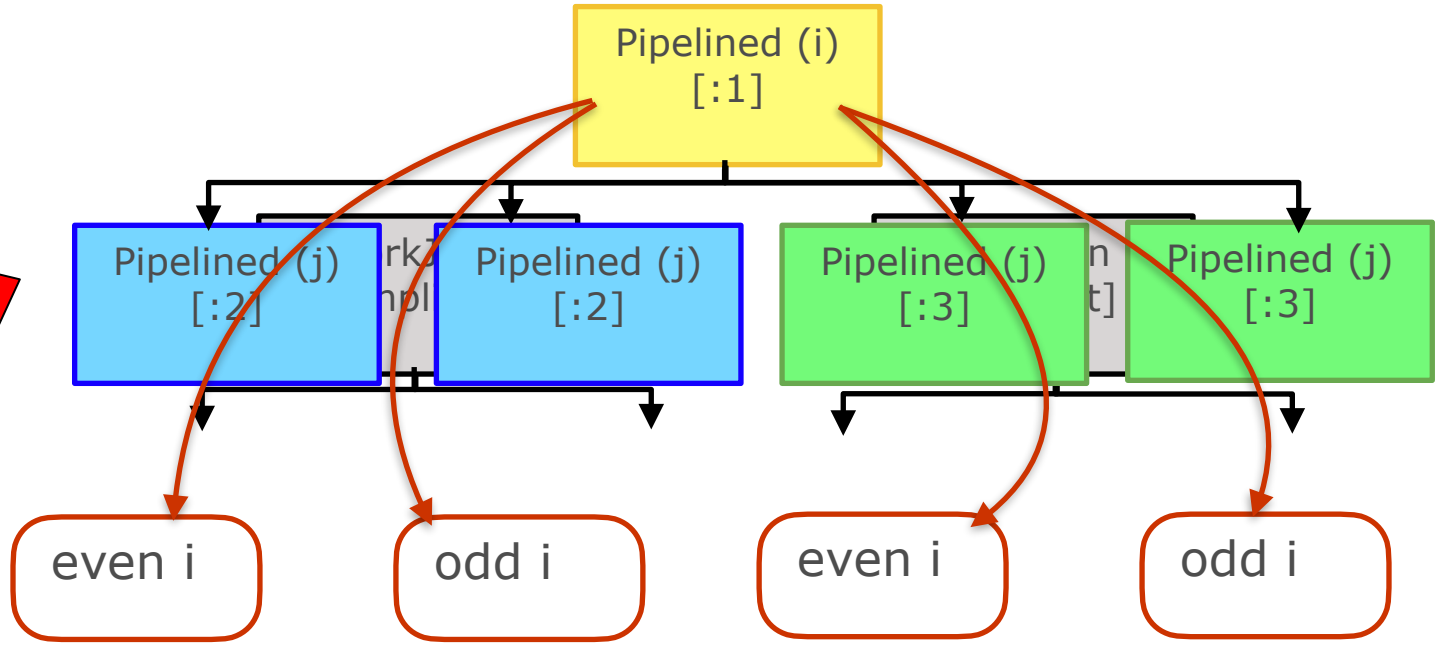
Outer Controller Parallelization

- Simply duplicating controllers does not fully exploit the parallelism
- Parallelism can be captured as either a: **Pipeline of ForkJoins (PoF)**

```
Foreach(N by 1 par 2){ i =>
  Foreach(…) {…}
  Foreach(…) {…}
}
```



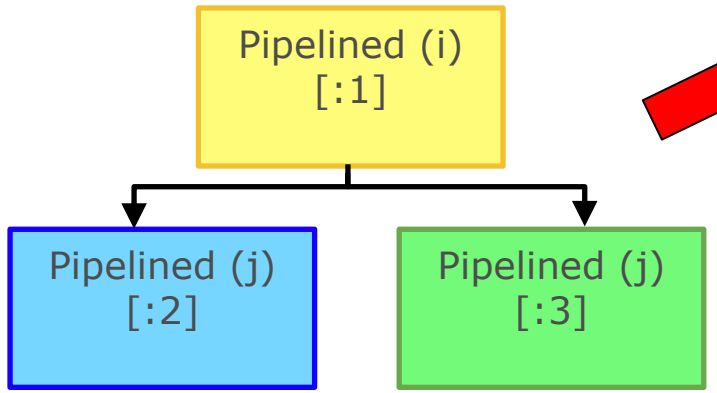
(Unparallelized Structure)



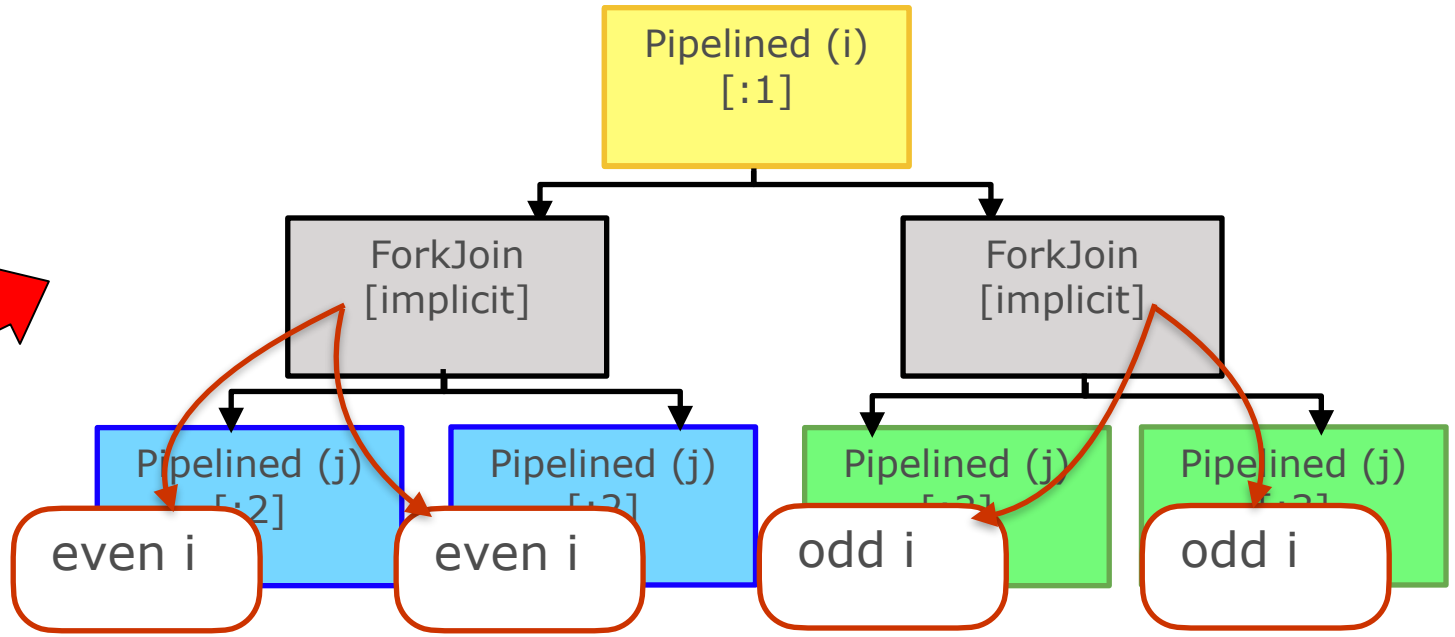
Outer Controller Parallelization

- Simply duplicating controllers does not fully exploit the parallelism
- Parallelism can be captured as either a: Pipeline of ForkJoins (PoF) or a **ForkJoin of Pipelines (FoP)**

```
Foreach(N by 1 par 2){ i =>
  Foreach(…) {…}
  Foreach(…) {…}
}
```

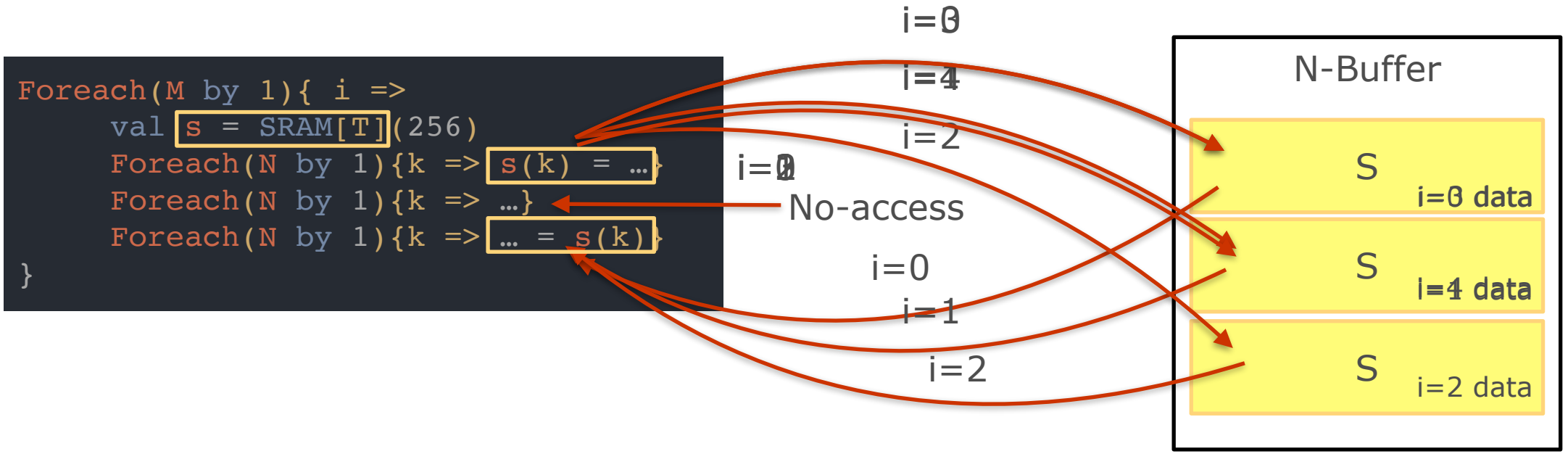


(Unparallelized Structure)



Resource Utilization

- **Buffering** is implicit duplication of a memory to protect accesses from each other in a **Pipelined** controller



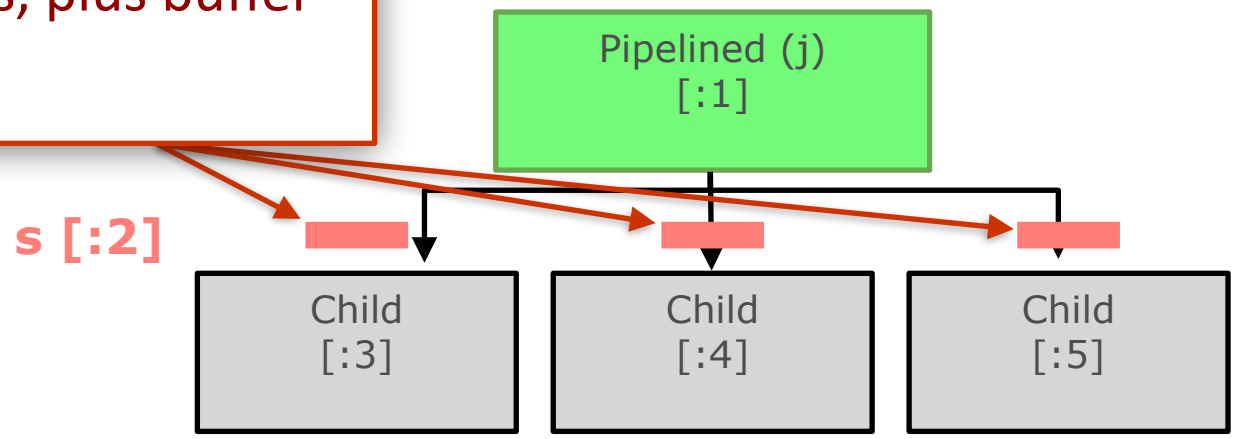
- The compiler computes buffering automatically

Resource Utilization

- **Buffering** is embedded in the controller hierarchy diagram.

```
Foreach(M by 1){ j =>
    val s = SRAM[T](256)
    Foreach(N by 1){k => s(
    Foreach(N by 1){k => …}
    Foreach(N by 1){k => … = s(k)}
}
```

The SRAM requires 3 separate 256-word memories, plus buffer management logic!



Principled Approach to Program Optimization

The steps and how the controller hierarchy diagrams help

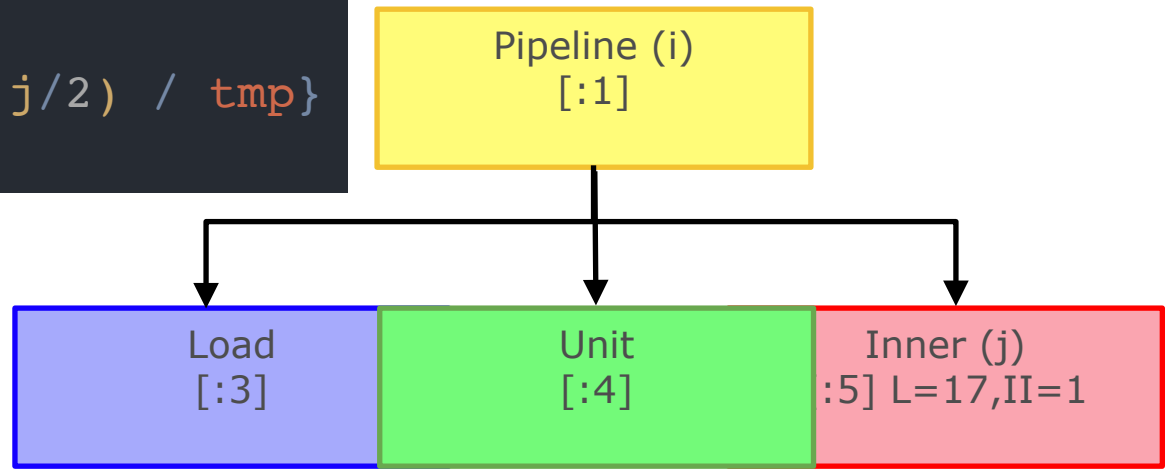
Using Controller Diagrams

- We will take a principled approach to optimization with 4 steps:
  1. Assess the bottleneck
  2. Modify the hierarchy
  3. Increase DMA efficiency
  4. Tweak scheduling

Example

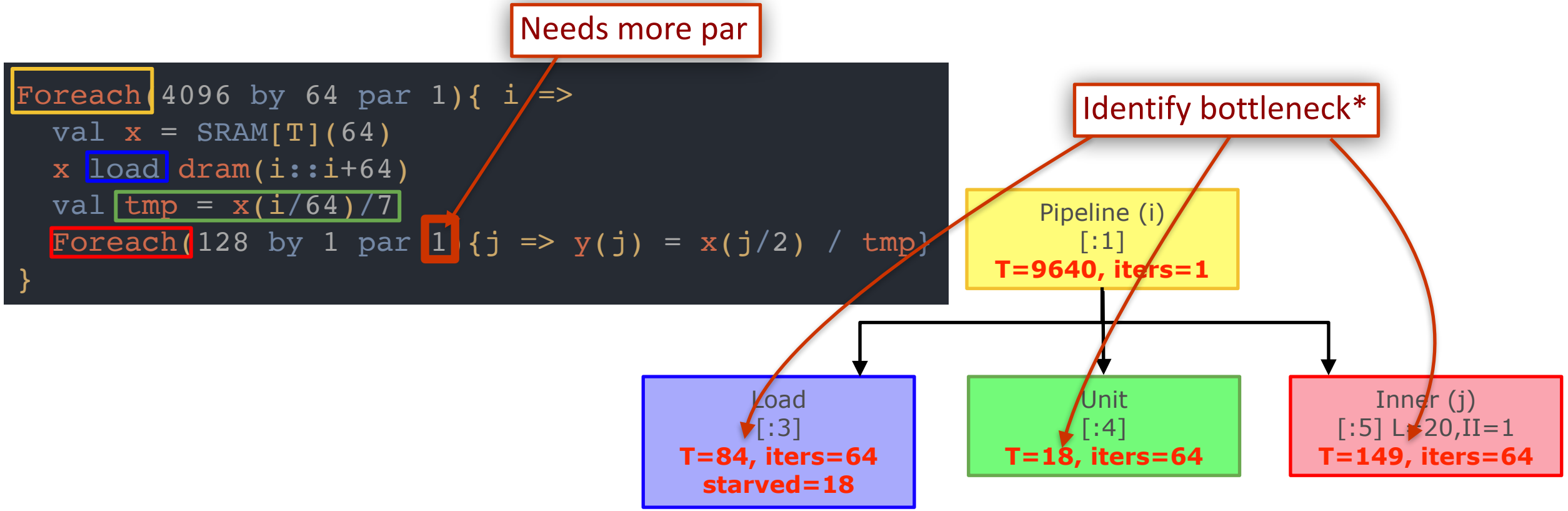
- Suppose someone wrote this snippet. Let's apply the procedure to see how terribly the code actually maps to hardware…

```
Foreach(4096 by 64 par 1){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  val tmp = x(i/64)/7
  Foreach(128 by 1 par 1){j => y(j) = x(j/2) / tmp}
}
```



Bottleneck Analysis

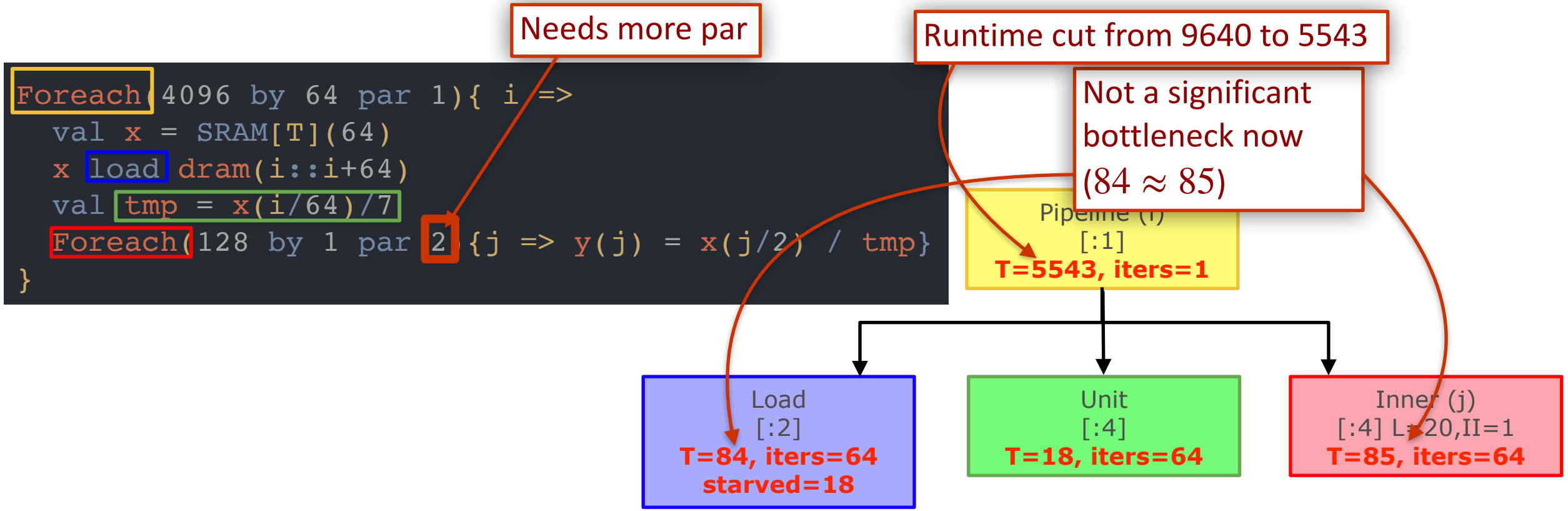
- 1) Assess the bottleneck



* Bottleneck here is significant because 3-stage pipeline runs for 64 iterations, meaning it is in steady-state for a while

Bottleneck Analysis

- 1) Assess the bottleneck



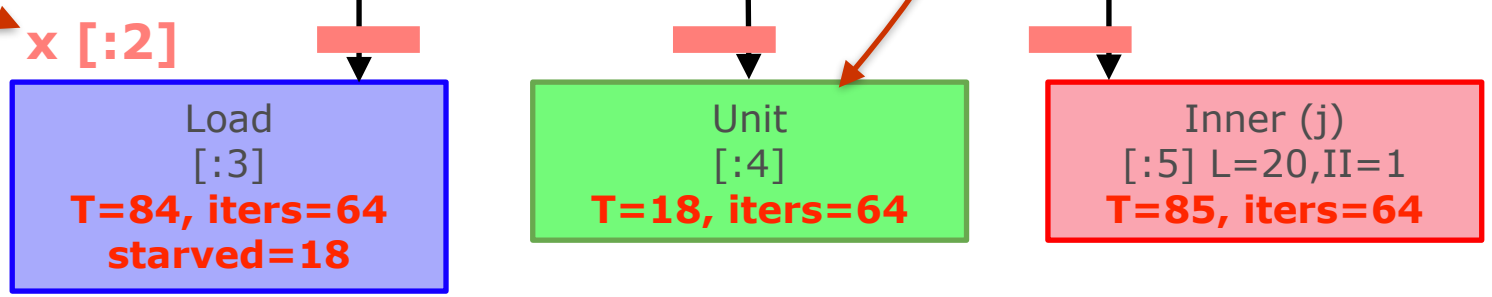
Bottleneck Analysis

- 2) Adjust the hierarchy

```
Foreach(4096 by 64 par 1){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  val tmp = x(i/64)/7
  Foreach(128 by 1 par 2){j => y(j) = x(j/2) / tmp}
}
```

Code can be restructured to eliminate stage by computing this in inner controller

Is triple-buffer necessary?



Bottleneck Analysis

- 2) Adjust the hierarchy

```
Foreach(4096 by 64 par 1){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

Reduced to double buffer

x [:2]

Tiny reduction in runtime
(5543 -> 5524)

Pipeline (i)
[:1]
T=5524, iters=1

Load
[:3]
T=84, iters=64
starved=18

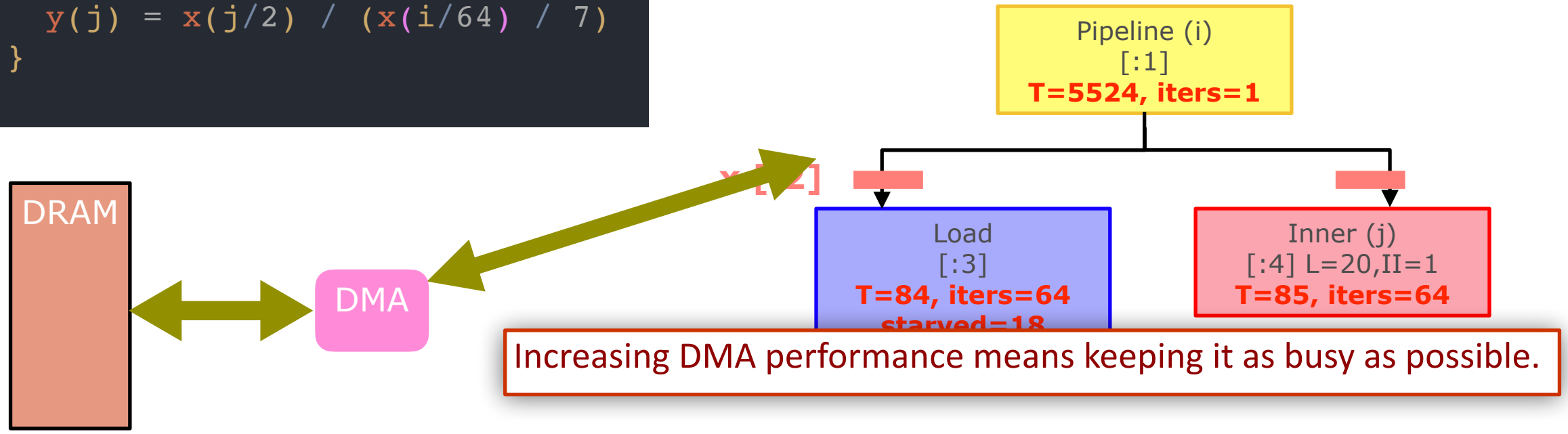
Inner (j)
[:4] L=20,II=1
T=85, iters=64

Note this change is "free" in hardware the divide-by-7 circuitry is stamped out either way, and its latency can be hidden by the existing latency of the inner controller

Bottleneck Analysis

- 3) Maximize DMA efficiency

```
Foreach(4096 by 64 par 1){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

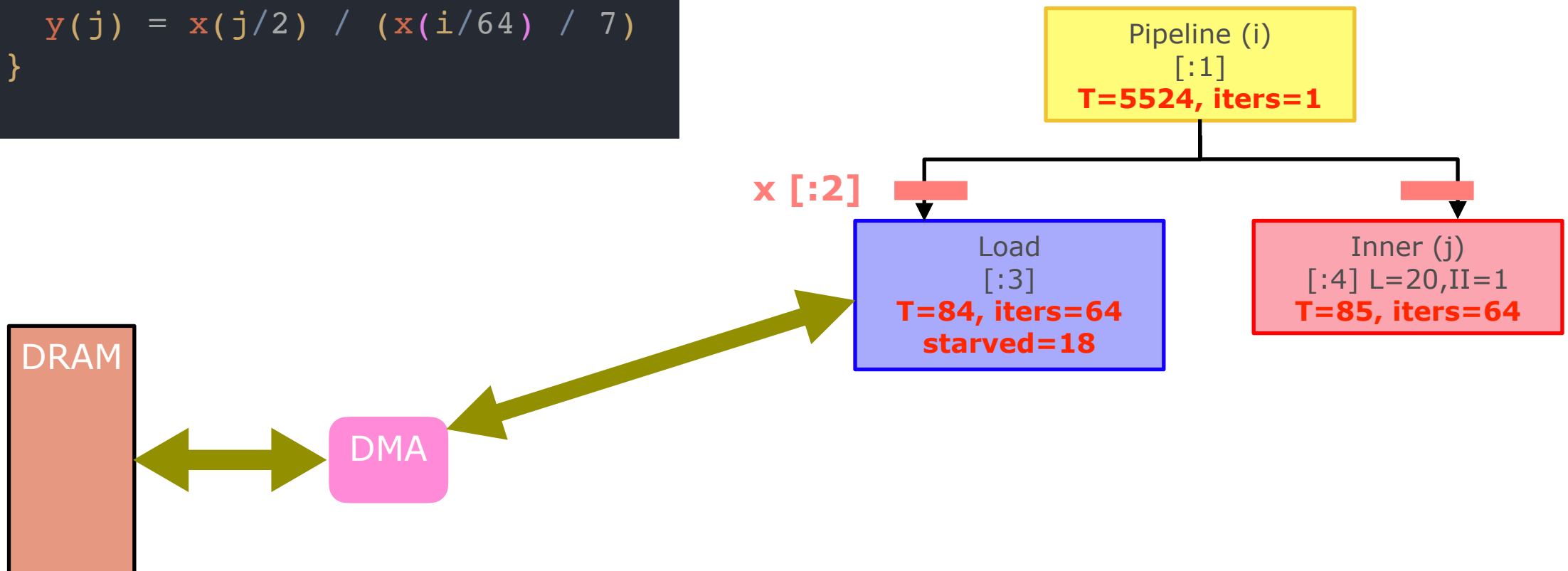


Bottleneck Analysis

- 3) Maximize DMA efficiency

```
Foreach(4096 by 64 par 1){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

Let's focus on increasing parallelization…

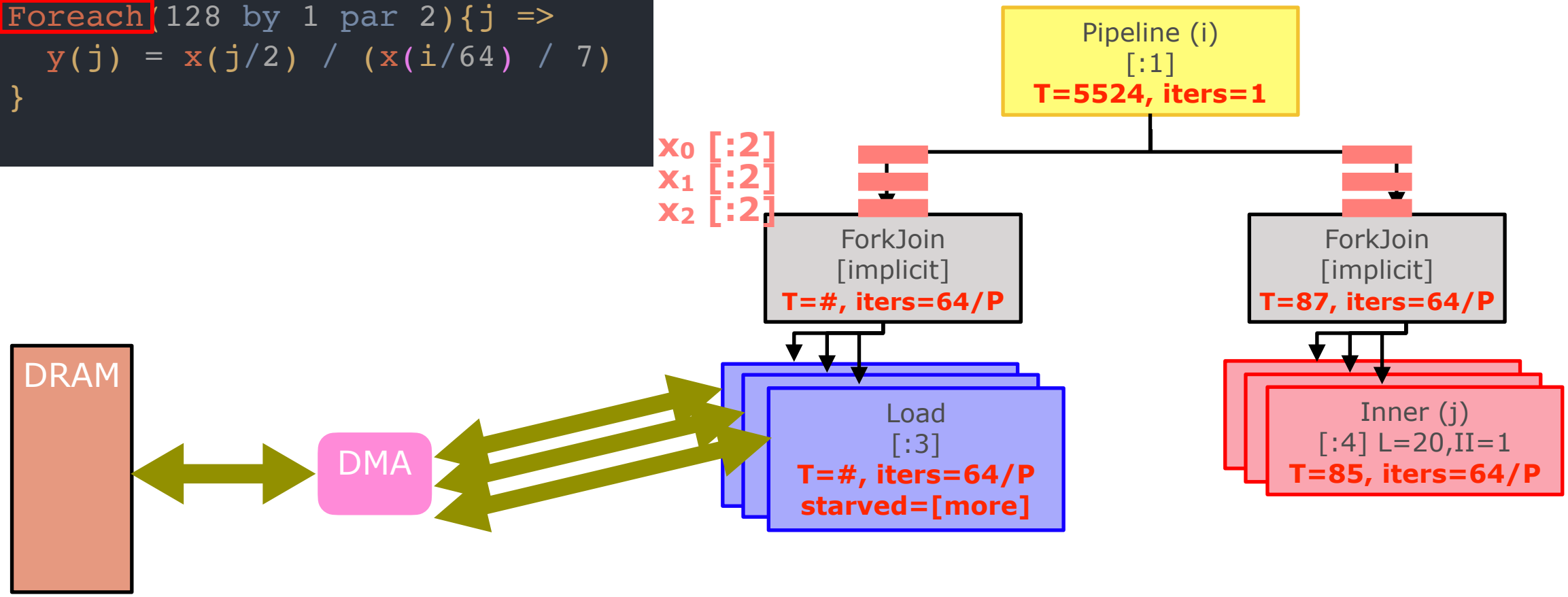


Bottleneck Analysis

- 3) Maximize DMA efficiency

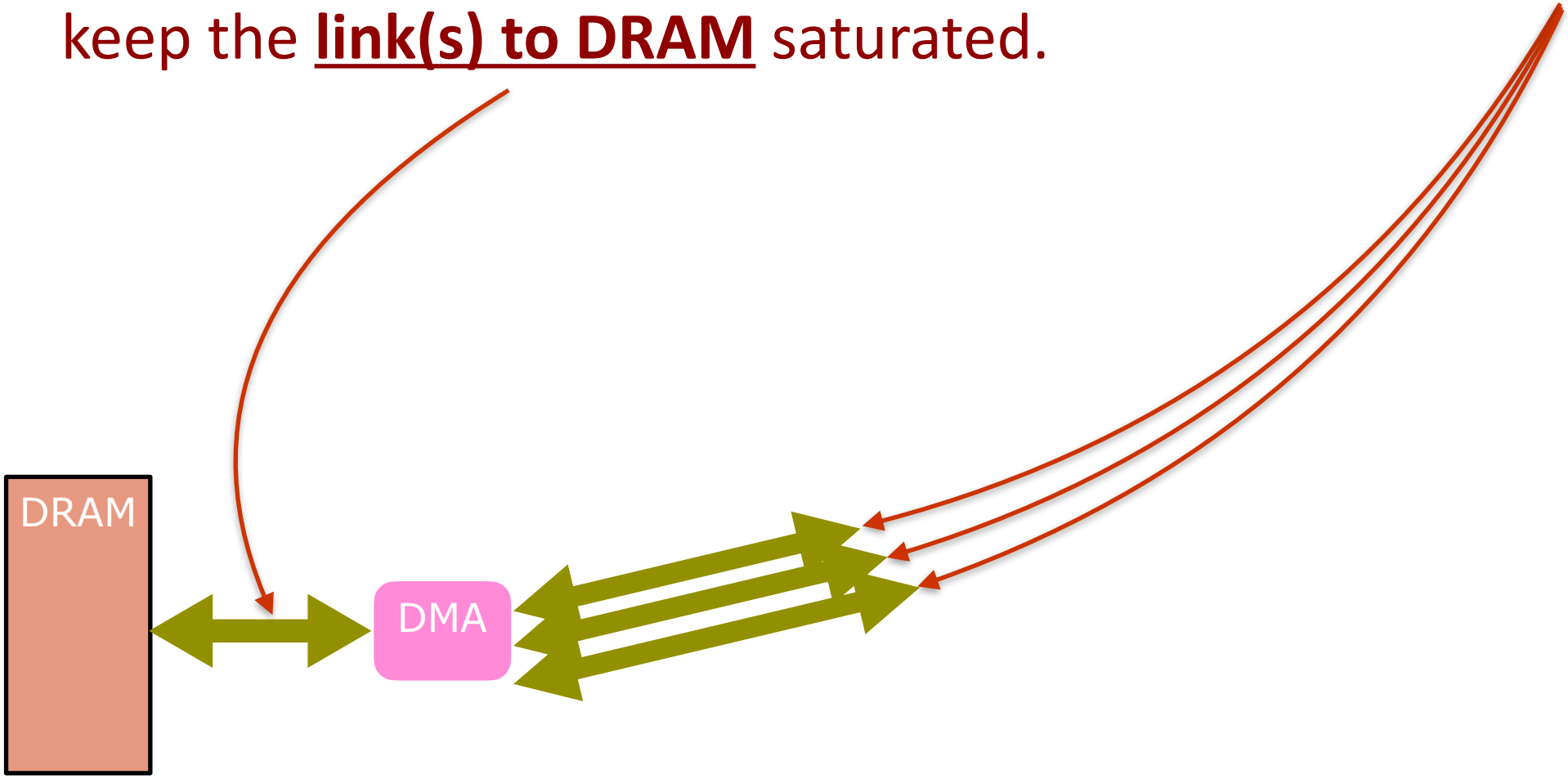
```
Foreach(4096 by 64 par P){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

Let's focus on increasing parallelization…



Bottleneck Analysis

- 3) Maximize DMA efficiency
- The goal is to construct a controller tree whose **links to the DMA** keep the **link(s) to DRAM** saturated.

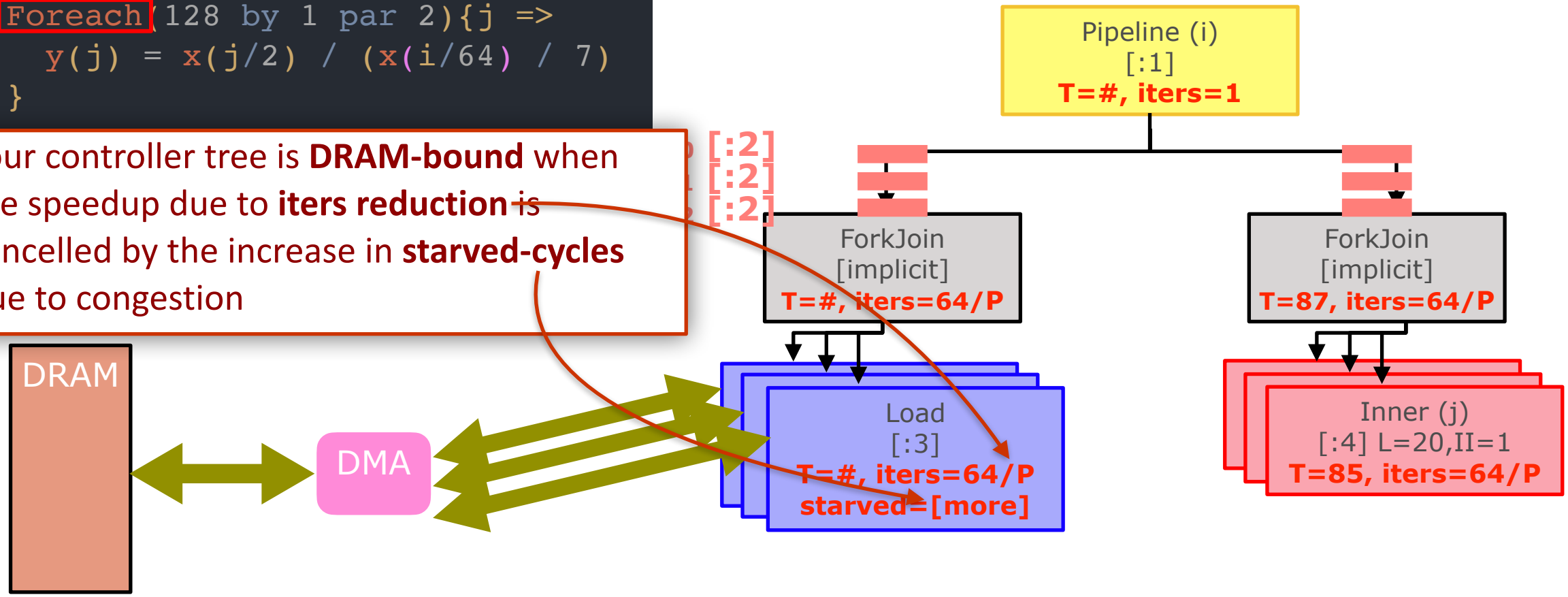


Bottleneck Analysis

- 3) Maximize DMA efficiency

```
Foreach(4096 by 64 par P){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

Your controller tree is **DRAM-bound** when the speedup due to **iters reduction** is cancelled by the increase in **starved-cycles** due to congestion

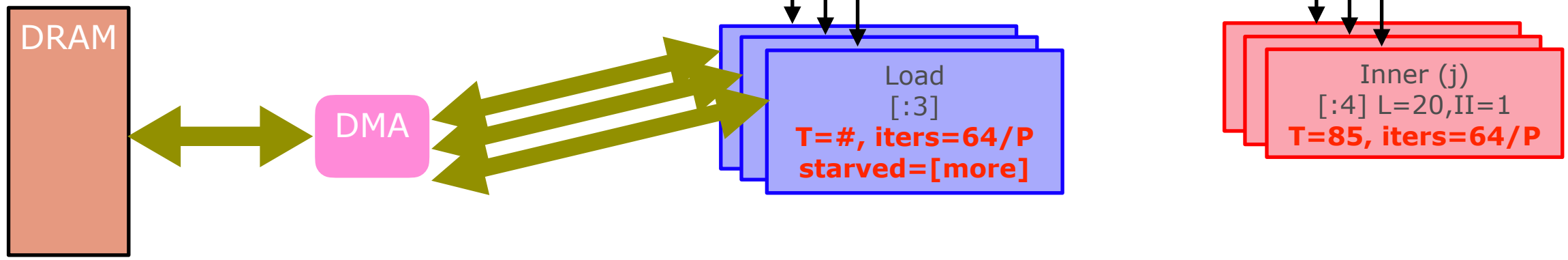


Bottleneck Analysis

- 4) Tweak scheduling

```
Foreach(4096 by 64 par P){ i =>
  val x = SRAM[T](64)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    y(j) = x(j/2) / (x(i/64) / 7)
  }
}
```

Rewriting the parent as a `Stream` controller could help cut resource utilization and runtime



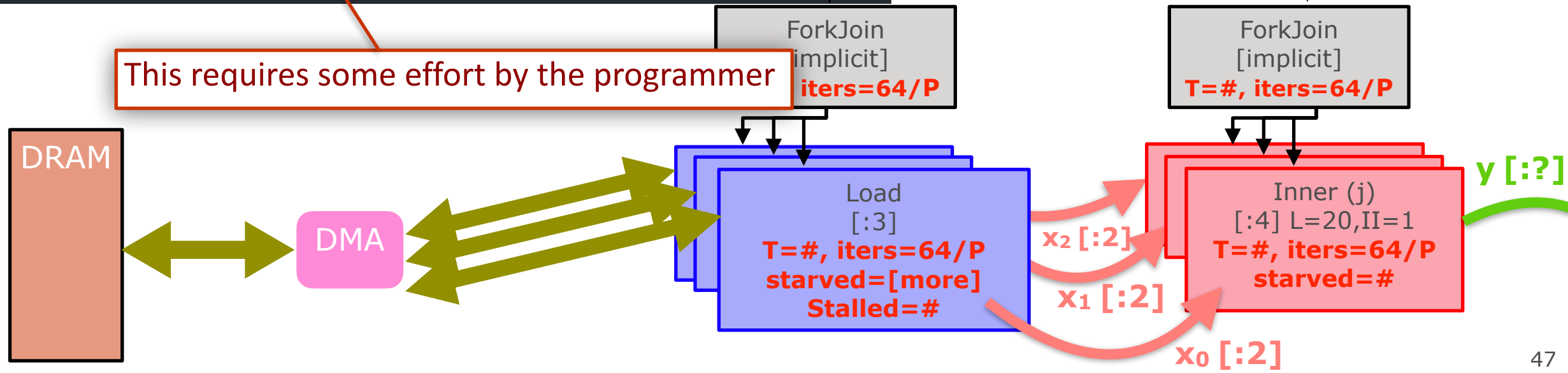
Bottleneck Analysis

- 4) Tweak scheduling

```
Stream.Foreach(4096 by 64 par P){ i =>
  val x = FIFO[T](4)
  x load dram(i::i+64)
  Foreach(128 by 1 par 2){j =>
    val num = mux(j%2 == 0, x.deq(), x.peek())
    if (i % 64 == 0) reg := x.peek()
    y.enq(num / (reg / 7))
  }
```

Rewriting the parent as a `Stream` controller could help cut resource utilization and runtime

This requires some effort by the programmer



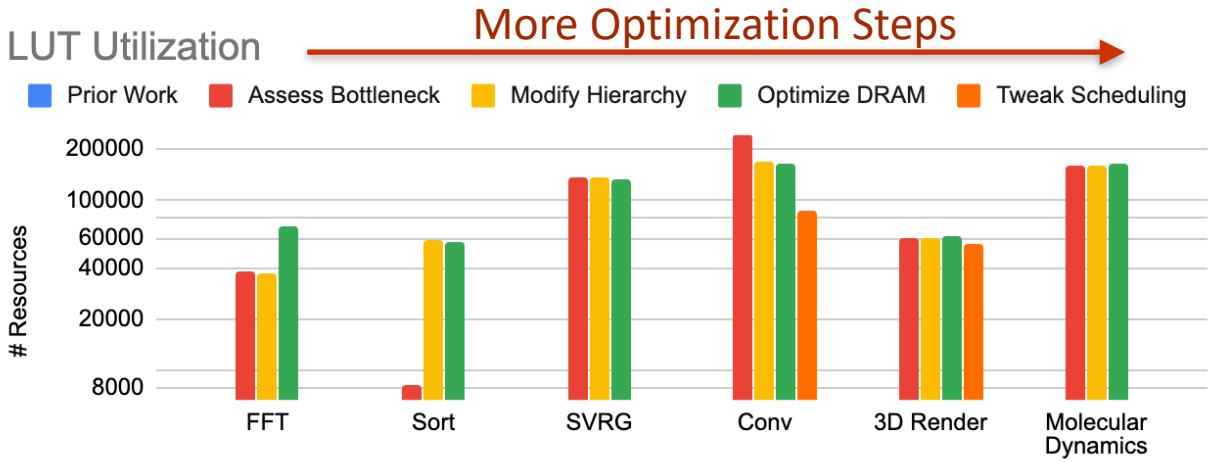
Does Performance Analysis Work?

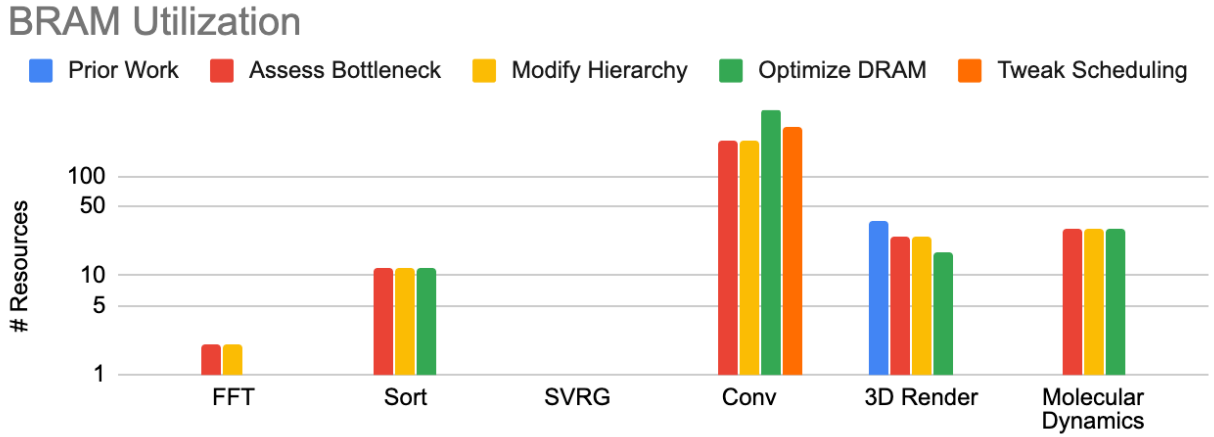
Experimental Setup

- We evaluated six applications on the Xilinx ZCU102 to understand how each step of the procedure helps resource utilization and performance
  - FFT[3]
  - Merge Sort[3]
  - Stochastic Variance-Reduced Gradient Descent (SVRG)[4]
  - CNN Layer[5]
  - 3D Rendering[6]
  - Molecular Dynamics[3]

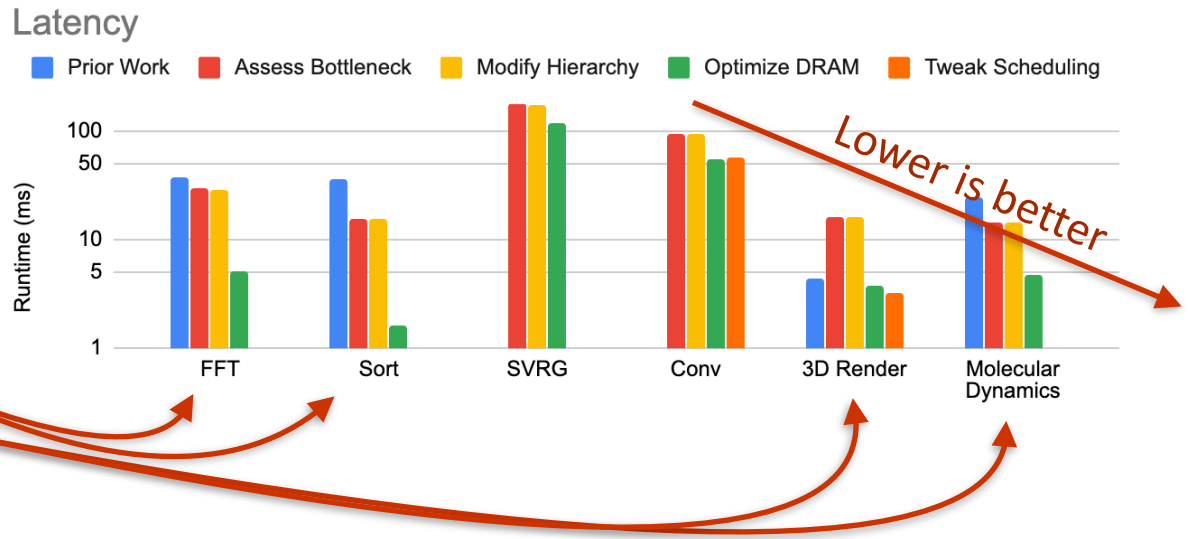
[3] B. Reagen et al "MachSuite: Benchmarks for accelerator design and customized architectures." IISWC 2014
[4] R. Johnson et al. "Accelerating stochastic gradient descent using predictive variance reduction." NIPS 2013
[5] A. Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." NIPS 2012
[6] Y. Zhou et al. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs" FPGA 2018

Results





By following all steps, latency can improve significantly without harming resource utilization.



* Not all levels of optimization apply to all apps, and not all prior work reported utilization numbers

Key Contributions in This Talk

- **Problem**: Expressing hardware at a high level results in sub-optimal designs
- **Solution**: Design a language and compiler that provides
  - the **right software abstractions**
  - A **performance debugging environment** that lends itself to a **set of optimization principles**
  - Hardware-oriented **optimizations for memory partitioning**
- **Validation**: Demonstrate effectiveness on a real-world problem

Memory Banking

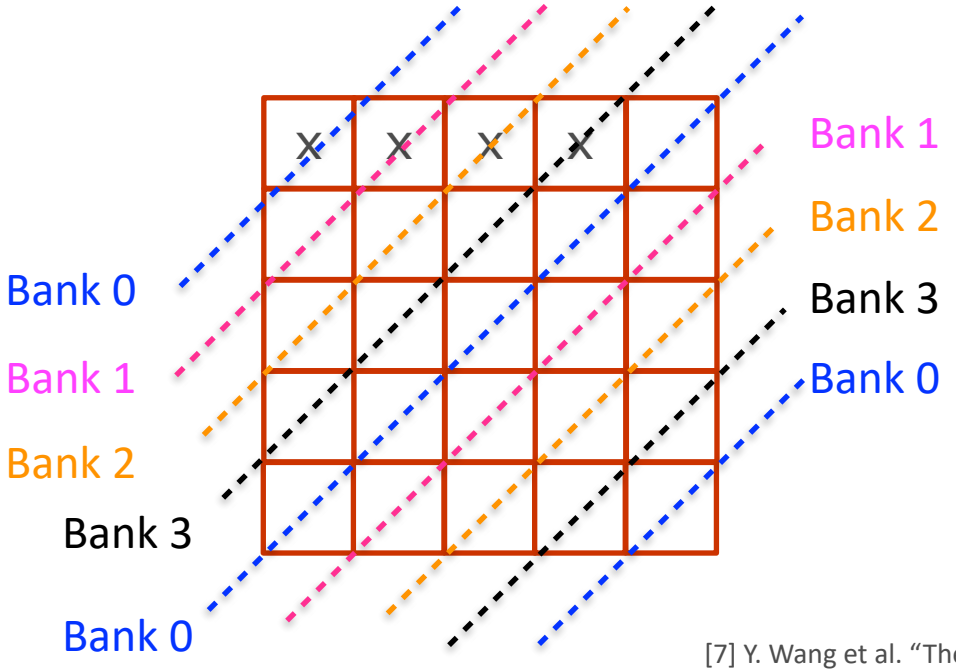
A blackhole for FPGA resources

Memory Banking - Related Work

- The compiler must *implicitly* chain BRAMs together to serve parallel accesses to memory
- There are two formalisms for memory partitioning:

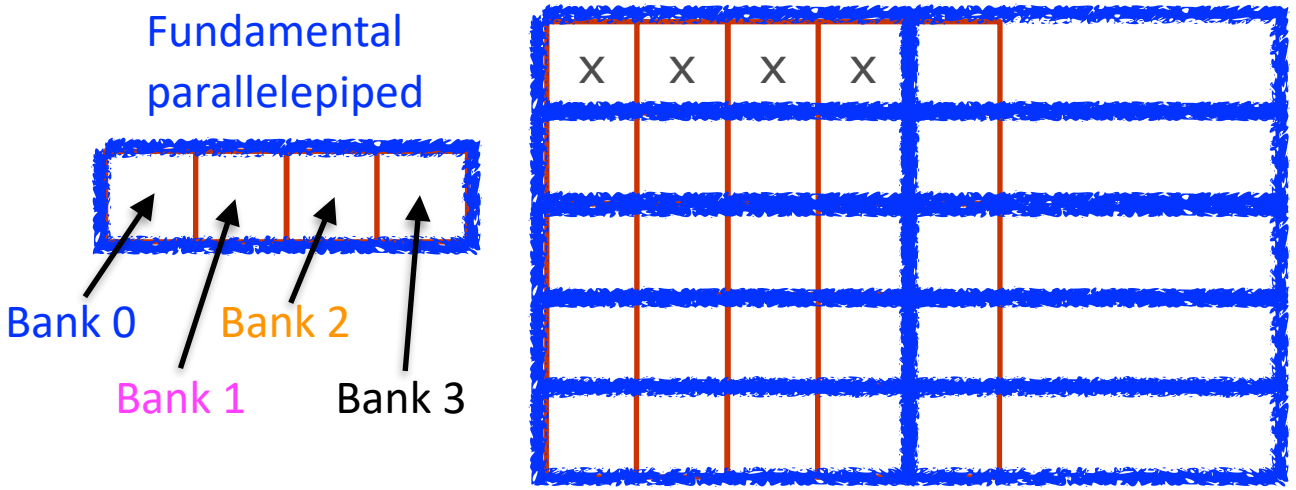
Hyperplane partitioning[7]

separation by parallel hyperplanes



Lattice partitioning[8]

separation by tessellated parallelepipeds



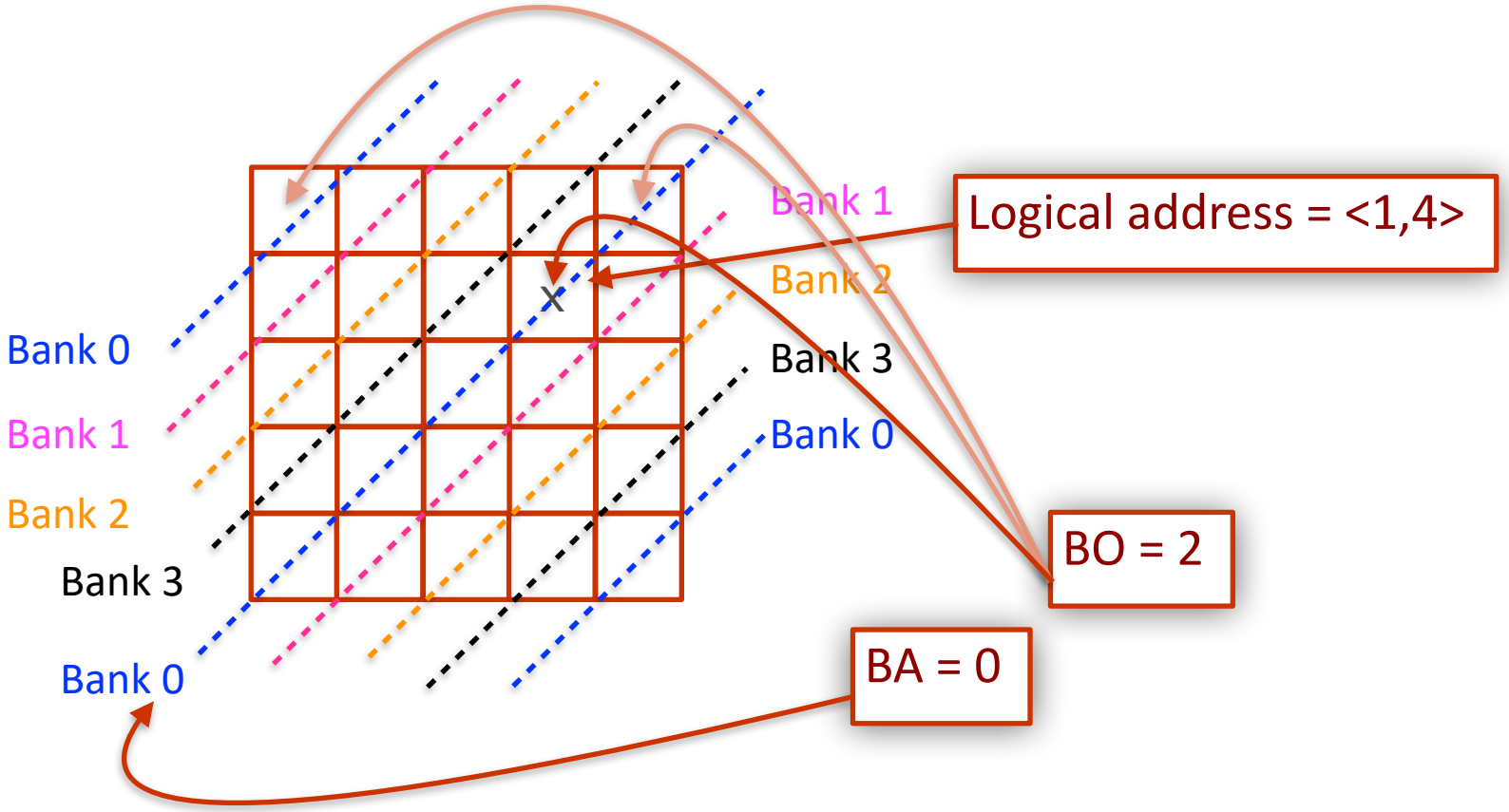
[7] Y. Wang et al. "Theory and algorithm for generalized memory partitioning in high-level synthesis." FPGA 2014
[8] A. Cilardo et al. "Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning." TACO 2015

Memory Banking

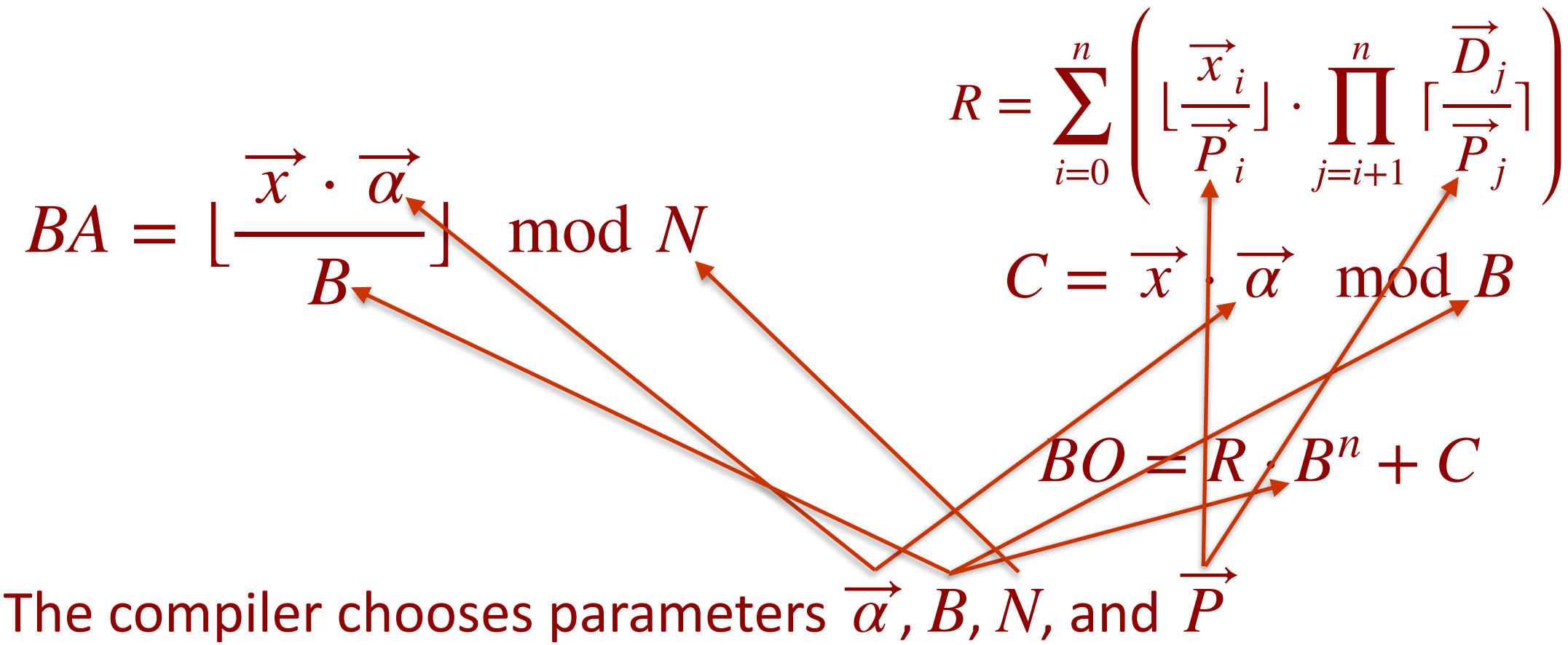
- Spatial relies **modified hyperplane partitioning**, which captures *all* hyperplane geometries and a *subset* of lattice geometries (orthogonal parallelepipeds)

Memory Banking

- How do you compute the bank (BA) and intra-bank offset (BO) of a a logical address, $\vec{x}$, to a memory with dimensions $\vec{D}$?



Banking Equations



- The compiler chooses parameters $\vec{\alpha}$, B, N, and $\vec{P}$

Banking Equations

- **Problem**: Expensive computations are pervasive in these equations…

$$BA = \lfloor \frac{\vec{x} \cdot \vec{\alpha}}{B} \rfloor \mod N$$

$$R = \sum_{i=0}^{n} \left(\lfloor \frac{\vec{x}_i}{\vec{P}_i} \rfloor \cdot \prod_{j=i+1}^{n} \lceil \frac{\vec{D}_j}{\vec{P}_j} \rceil \right)$$

$$C = \vec{x} \cdot \vec{\alpha} \mod B$$

$$BO = R \cdot B^n + C$$

Banking Equations

- **Problem**: Expensive computations are pervasive in these equations…

$$BA = \lfloor \frac{\vec{x} \cdot \vec{\alpha}}{B} \rfloor \mod N$$

$$R = \sum_{i=0}^{n} \left(\lfloor \frac{\vec{x}_i}{\vec{P}_i} \rfloor \cdot \prod_{j=i+1}^{n} \lceil \frac{\vec{D}_j}{\vec{P}_j} \rceil \right)$$

$$C = \vec{x} \cdot \vec{\alpha} \mod B$$

$$BO = R \cdot B^n + C$$

Banking Equations

- **Problem**: Expensive computations are pervasive in these equations…

$$BA = \left\lfloor \frac{\vec{x} \cdot \vec{\alpha}}{B} \right\rfloor \mod N$$

$$R = \sum_{i=0}^{n} \left(\text{Variable} \cdot \text{Constant} \right)$$

$$C = \vec{x} \cdot \vec{\alpha} \mod B$$

$$BO = R \cdot B^n + C$$

Banking Equations

- **Solution**: Choose constants that allow resource-saving transformations
  - **Multiplication Decomposition** (e.g. $12x = 8x + 4x = x \ll 3 + x \ll 2$)
  - **Modulo and Division via Crandall's Algorithm**[9] - Replace div/mod by Mersenne number with bitwise operations and addition
    - Mersenne number: $M = 2^n - 1$

[9] J. Chung et al. "Low-Weight Polynomial Form Integers for Efficient Modular Multiplication." IEEE Transactions on Computers 2007

Banking Equations

- **Solution**: Choose constants that allow resource-saving transformations
  - **Multiplication Decomposition** targets (up to second-order)
    - 1 2 3 4 5 6 7 8 9 10 ~~11~~ 12 ~~13~~ 14 15 16 17 18 ~~19~~ 20 ~~21~~
  - **Crandall's Algorithm** targets (up to second-order)
    - 1 2 3 4 5 ~~6~~ ~~7~~ 8 9 ~~10~~ ~~11~~ ~~12~~ ~~13~~ ~~14~~ 15 16 17 ~~18~~ ~~19~~ ~~20~~ 21

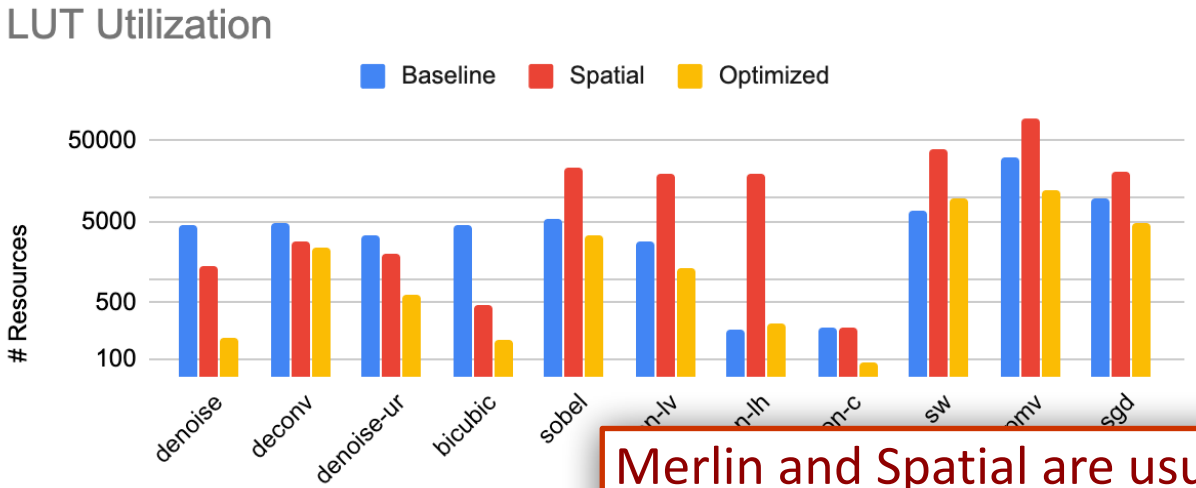
- This flexibility is *very* useful for choosing the best banking parameters

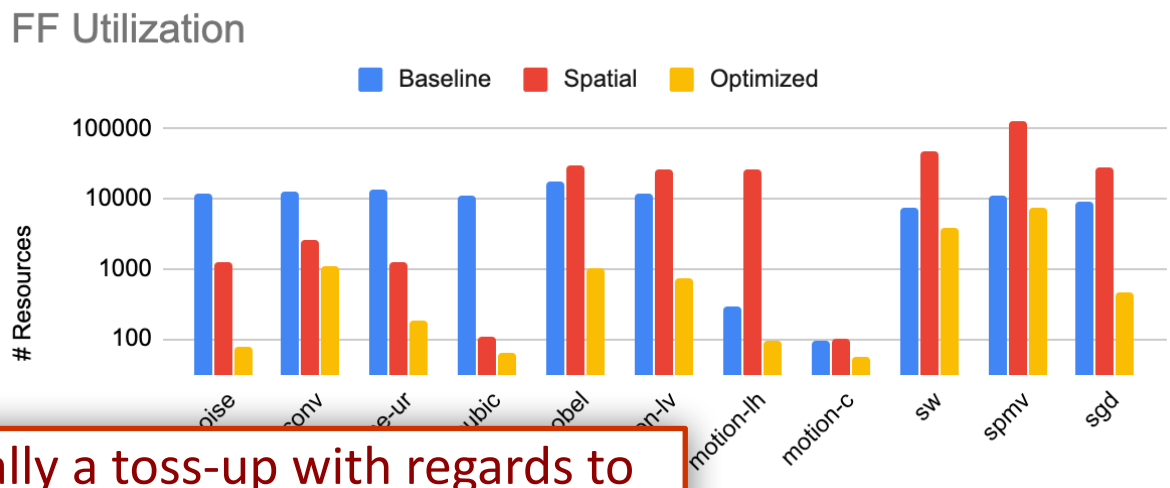
Experimental Setup

- We compared resource utilization of this optimized analysis on 11 access patterns against two state-of-the-art SDH tools: Merlin Compiler[10] and unmodified Spatial
  - **Stencils**[7]:
    - deconv + deconv-ur
    - denoise
    - bicubic
    - sobel
    - jpeg motion lh + lv + c
  - **Algorithms**:
    - Smith-Waterman genetic alignment[11]
    - sparse matrix-vector multiplication
    - stochastic gradient descent (SGD)[12]

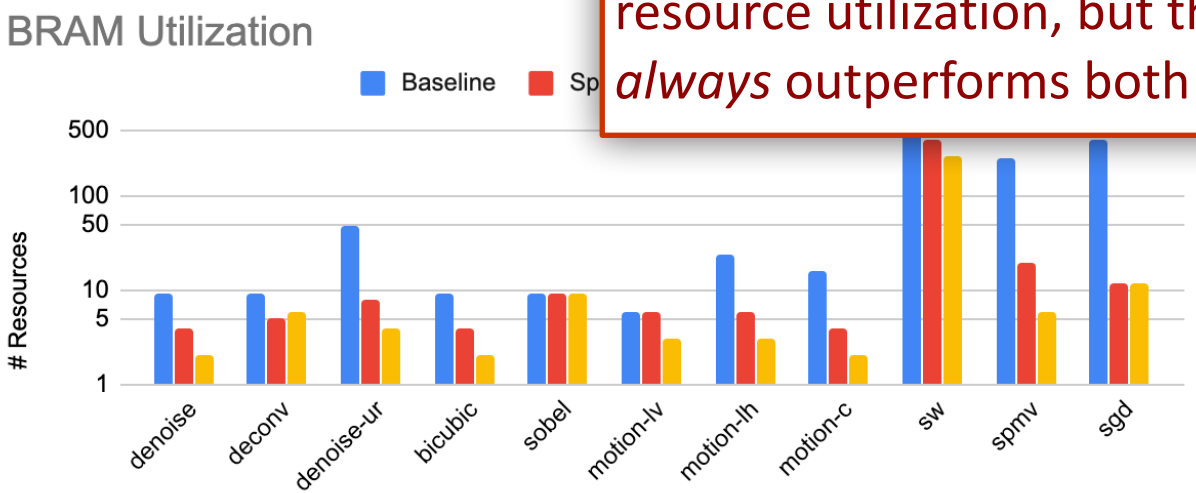
[10] https://www.falconcomputing.com/merlin-fpga-compiler/
[7] Y. Wang et al. "Theory and algorithm for generalized memory partitioning in high-level synthesis." FPGA 2014
[11] Y. Turhakia et al. "Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly." ASPLOS 2018
[12] C. De Sa. et al. "Understanding and optimizing asynchronous low-precision stochastic gradient descent." ISCA 2017

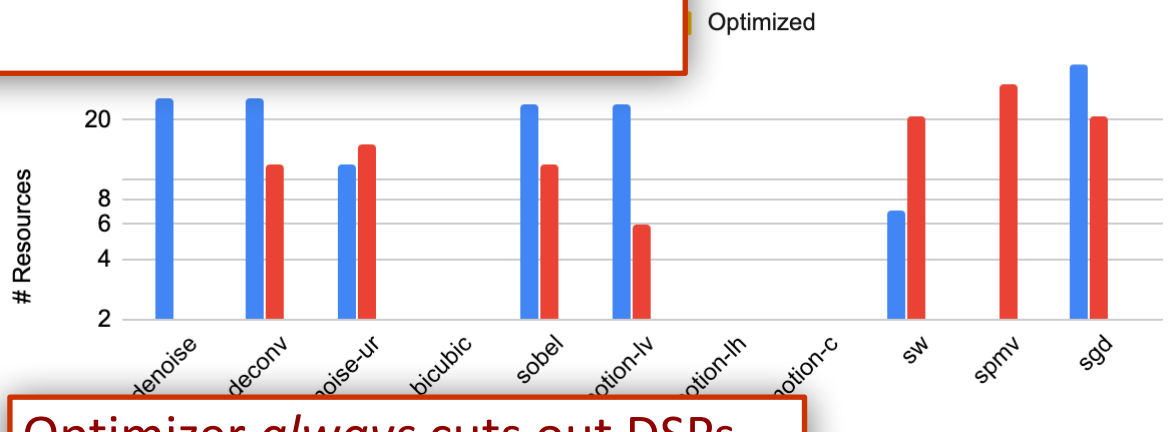
Results





Merlin and Spatial are usually a toss-up with regards to resource utilization, but the optimized banking analysis *always* outperforms both





Optimizer *always* cuts out DSPs

Key Contributions in This Talk

- **Problem**: Expressing hardware at a high level results in sub-optimal designs
- **Solution**: Design a language and compiler that provides
  - the **right software abstractions**
  - A **performance debugging environment** that lends itself to a **set of optimization principles**
  - Hardware-oriented **optimizations for memory partitioning**
- **Validation**: Demonstrate effectiveness on a real-world problem

How Well Does Spatial Work in the Real World?

XFEL Pulse Alignment on LCLS-II

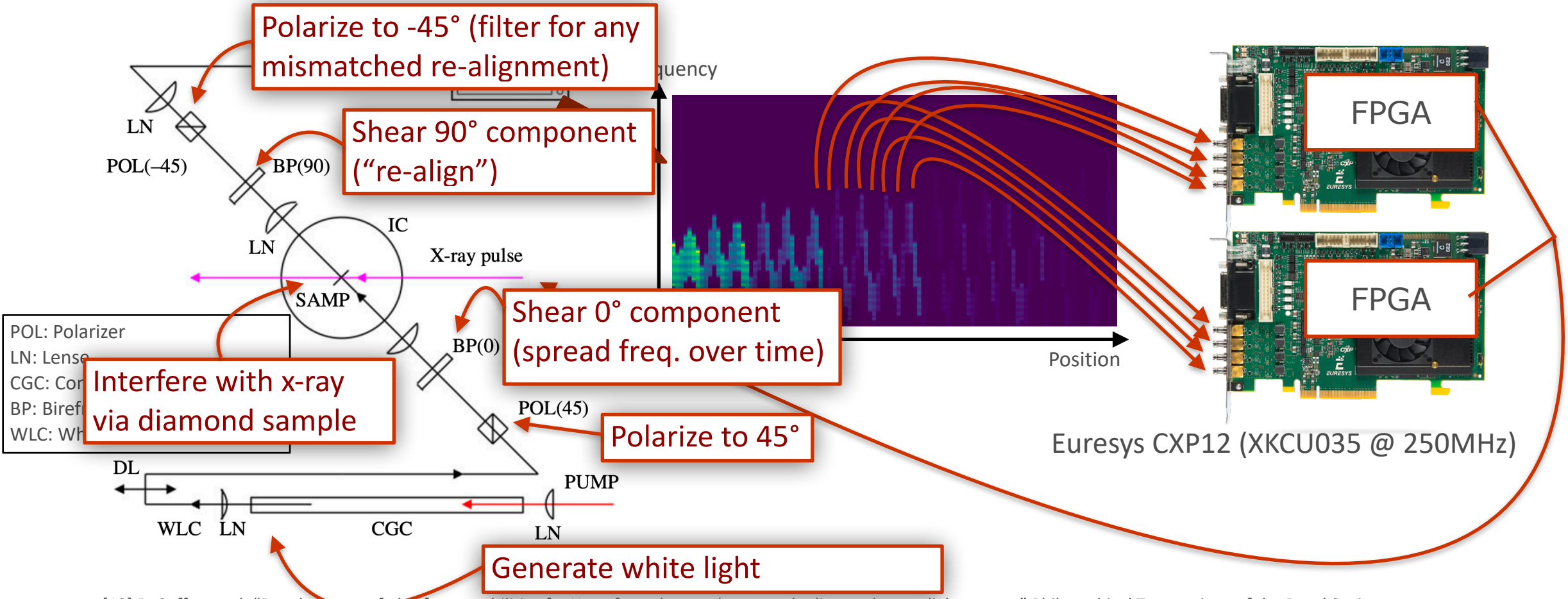
Background

- **SLAC National Accelerator Laboratory** is a Dept. of Energy research lab that uses synchrotron radiation and free-electron lasers for fundamental science
- Annual budget: $383M
- Annual users: 2,700
- The newest system is LCLS-II (Linac Coherent Light Source)
  - Provides scientists with much higher brightness x-rays and faster repetition rates (up to 10,000x LCLS-I)
  - The "**timetool**" is a critical diagnostic component for optical/x-ray pulse synchronization

SLAC NATIONAL ACCELERATOR LABORATORY

The Task

- Predict arrival time of x-ray pulse by measuring its effect on chirped white light[13]



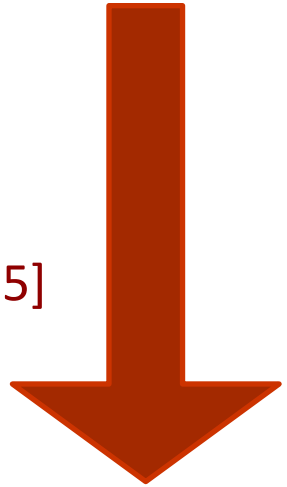
[13] R. Coffee et al. "Development of ultrafast capabilities for X-ray free-electron lasers at the linac coherent light source." Philosophical Transactions of the Royal Society

The Task

- In order to solve this task on the available FPGAs in real-time, we need a way to quickly and easily
  - Design various ML networks
  - Explore the design space of each network
  - Converge on the *best* solution (minimize latency and mean-absolute error) that satisfies the FPGA's resource budget

The Solution

- Spatial makes this easy*
- Assessed four classes of inference networks
  - Polynomial fit
  - Gradient Boosting Trees (XGBoost)[14]
  - Lattice Regression (with Hypercube interpolation)[15]
  - Multilayer Perceptron (MLP)[16]



"Heavier" computation at inference time

But is the improvement in ML metrics worth it?

* Measured in programming time/effort (hours or days) and LoC (~50 per kernel).

[14] T. Chen et al. "XGBoost: A Scalable Tree Boosting System." arXiv Preprint 2016
[15] E. Garcia. "Lattice Regression." NIPS 2009
[16] F. Murtagh. "Multilayer perceptrons for classification and regression" Neurocomputing 1991[

Results

Red cell indicates “limiting” resource if you scale up the model’s hyper parameters. Only best configuration per kernel is shown

Lattice and MLP require too much compute to fit the data

Polyfit cannot fit the data well

	Runtime Performance	Resource Utilization				ML Performance
Network Category	latency (ns)	LUT	FFs	DSP	BRAM	MAE (fs)
Baseline (FFT-based CPU)[13]	~1000	-	-	-	-	~5
Polyfit (3rd order)	104	832	1340	60	0	19.78
XGBoost (depth 7 x 128 trees)	60	155941	77107	0	0	1.39
Lattice (calib 4, edged 5)	92	6215	9425	191	0	4.01
MLP (8x16x4x1)	208	17428	34531	1134	0	2.91

XGBoost is LUT/routing intensive, but overall the best architecture for the task

[13] R. Coffee et al. “Development of ultrafast capabilities for X-ray free-electron lasers at the linac coherent light source.” Philosophical Transactions of the Royal Society

Conclusion

- The **right software abstractions** and **performance debugging environment** allows the programmer to take a **principled approach to hardware optimization**
- Memory partitioning can be be optimized by incorporating **automatic, targeted resource-saving optimizations**
- Domain experts can **rapidly innovate and generate optimal hardware** for complex algorithms

Acknowledgements


Kunle Olukotun


Christopher Ré


Kayvon Fatahalian

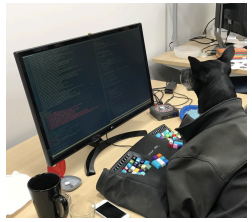

Ryan Coffee


Mykel Kochenderfer


Raghu Prabhakar

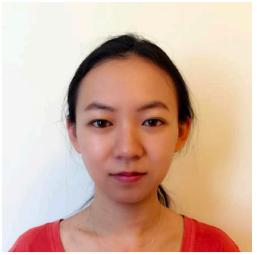

David Koeplinger


Alex Rucker


Matt Vilim


Tian Zhao


Nathan Zhang


Yaqi Zhang


Tushar Swamy


Stefan Shadjis

Olivia Hsu, Rekha Singhal, Muhammad Shahbaz, Ardavan Pedram, Luigi Nardi, and everyone from Kunle's group!

Dan Moreau, Andrea Brand-Sanchez, and all DAWN members!

Friends and Family and **Elyse**!

Thank You!