18-742:
Computer Architecture & Systems

Pipestitch: An Energy-Minimal Dataflow Architecture with Lightweight Threads

Mitchell Fream

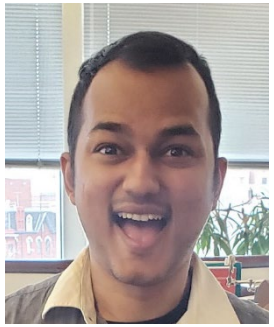
Spring 2025, Lecture 13

"Pipestitch: An Energy-Minimal Dataflow Architecture with Lightweight Threads"

Nathan Serafin, Souradip Ghosh, Harsh Desai,
Nathan Beckmann, Brandon Lucia 2023

- **Nathan:** CMU PhD
- **Souradip:** CMU PhD
- **Harsh:** CMU PhD
  - Defended 2 weeks ago!
- **Nathan:** CMU prof
  - Sloan Fellow 2024
- **Brandon:** CMU prof
  - Young Architect Award 2019





Executive summary

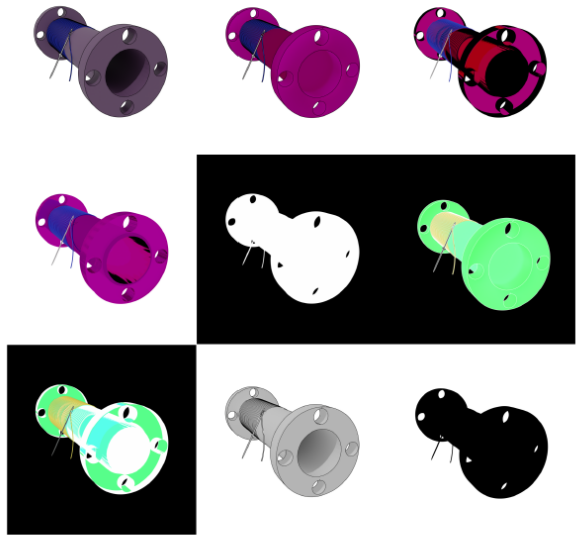
Sparse computation is important at the edge

Existing edge processors can't exploit parallelism in sparse linear algebra

Pipestitch: parallelism via *lightweight dataflow threads*

Fully implemented from compiler → RTL

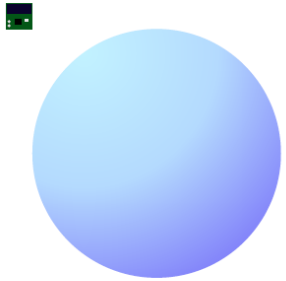
Pipestitch is 3.49x faster than prior state-of-the-art



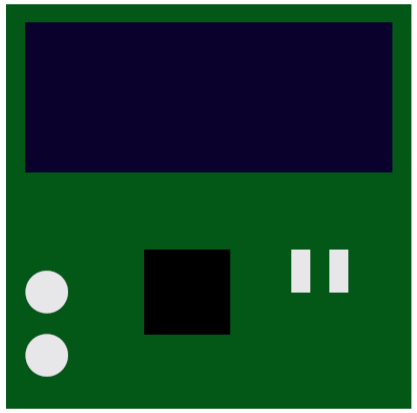
Extreme-edge computing is ubiquitous

Infrastructure monitoring

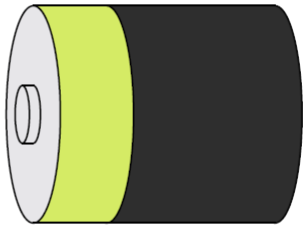
Chip-scale satellites



Chip-scale satellites
< 10mW harvested power



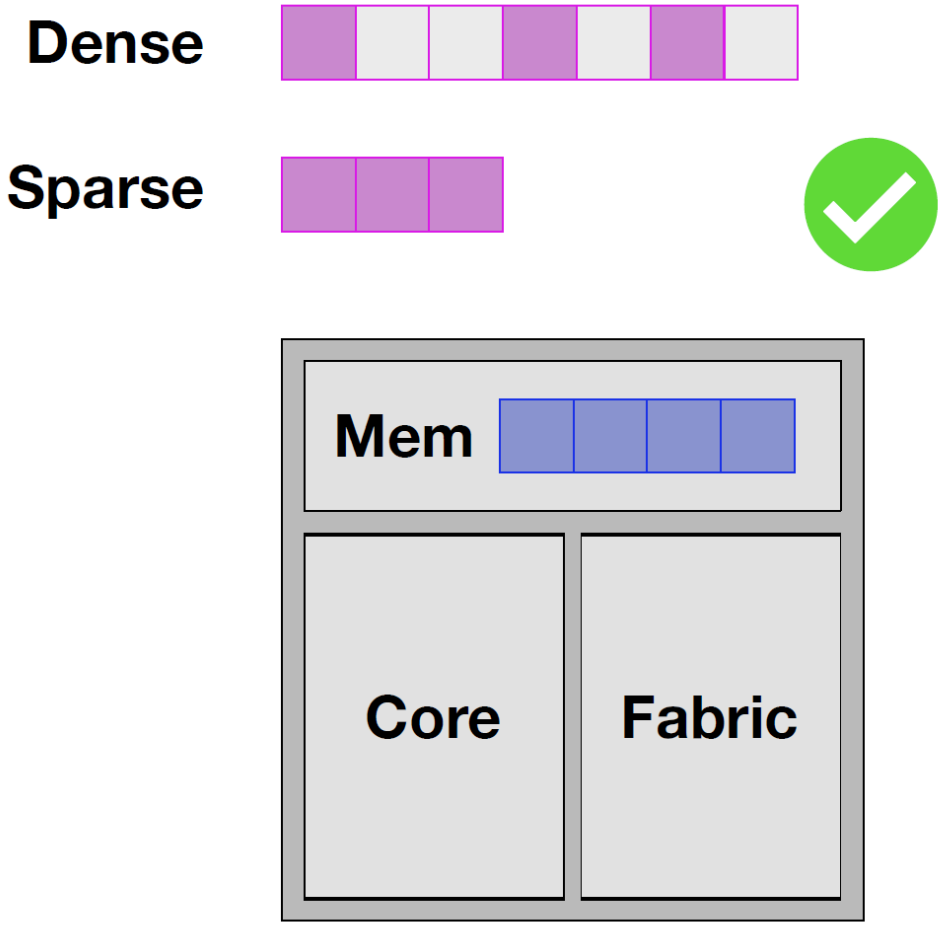
Common batteries
~1mW over 5 years



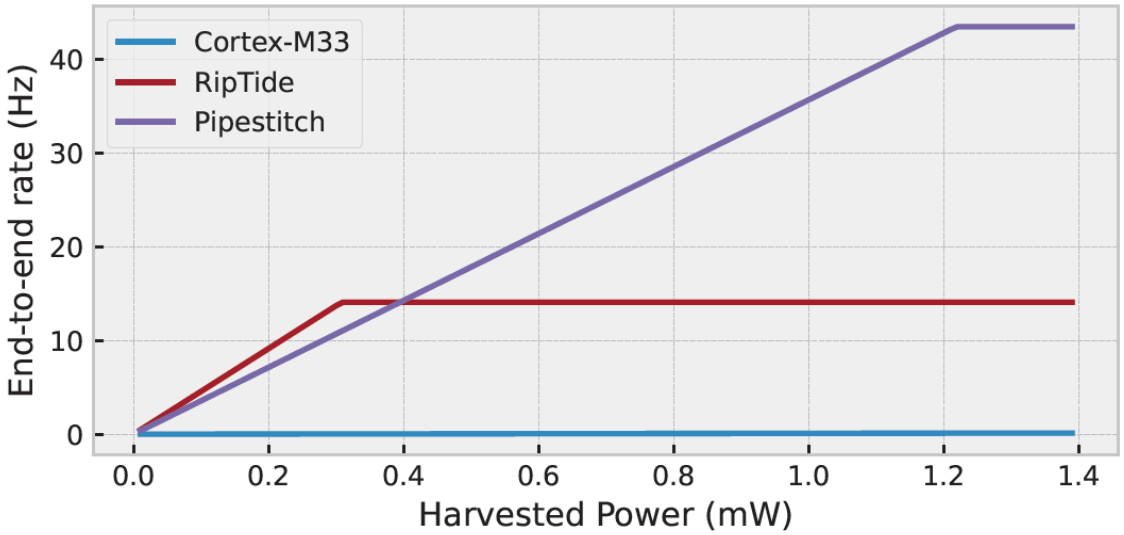
Health sensing

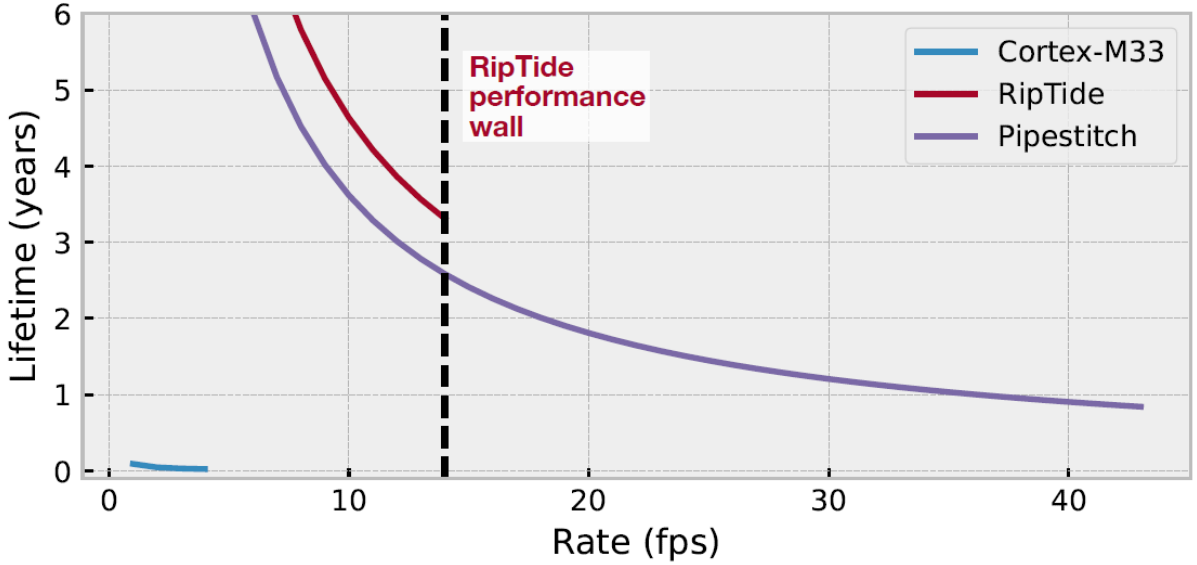
System memory ~100kB

Sparsity lets tiny devices run large apps

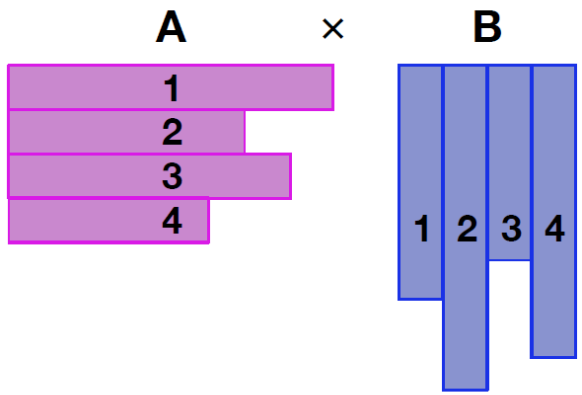


Sparse DNN Inference



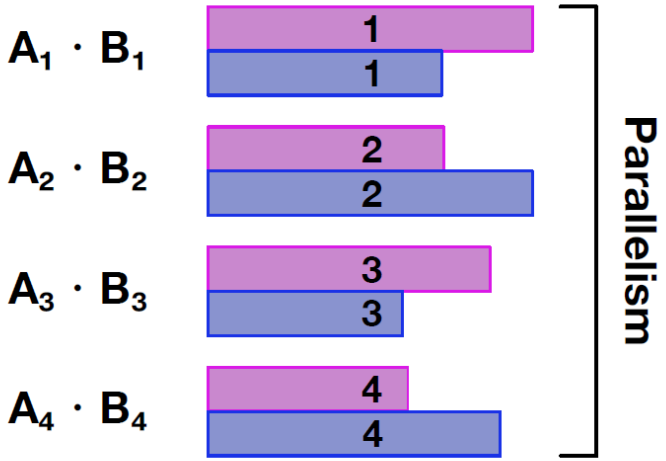


Sparse linalg has abundant parallelism




```
parallel for row = 0..R:
  acc = 0
  ...
  while iA < ARE && iB < BCE
    if a_col == b_col:...
    if a_col <= b_col:...
    if a_col >= b_col:...
    acc = new_acc
    iA = new_iA
    iB = new_iB
```

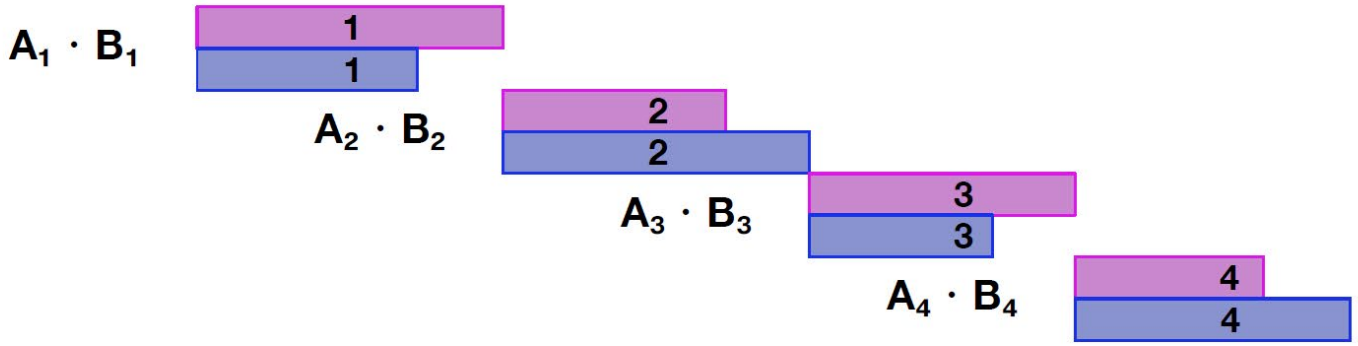
Sparse dot



State-of-the-art fails to exploit parallel work

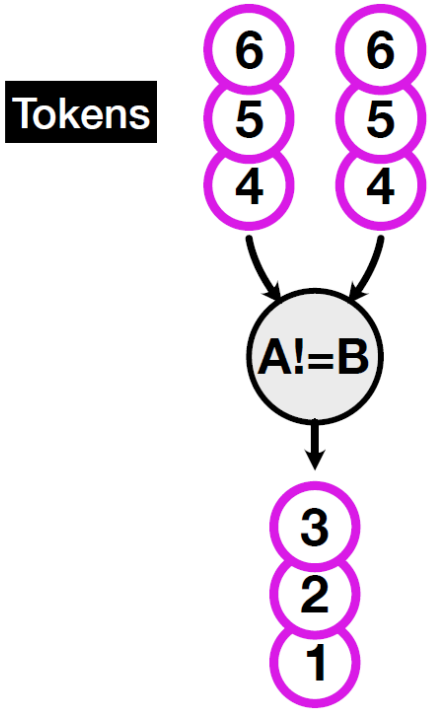
```
for row = 0..R:
  acc = 0
  ...
  while iA < ARE && iB < BCE
    if a_col == b_col:...
    if a_col <= b_col:...
    if a_col >= b_col:...
    acc = new_acc
    iA = new_iA
    iB = new_iB
```

Sparse dot

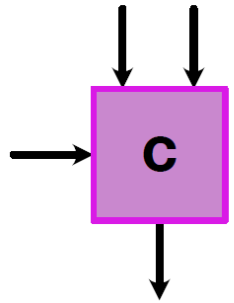


Conservative execution loses parallelism

Ordered dataflow saves energy

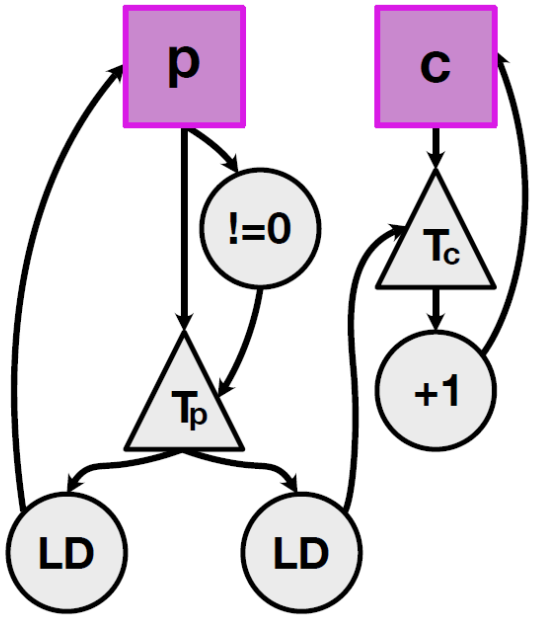


Carry gate sequentializes to preserve ordering

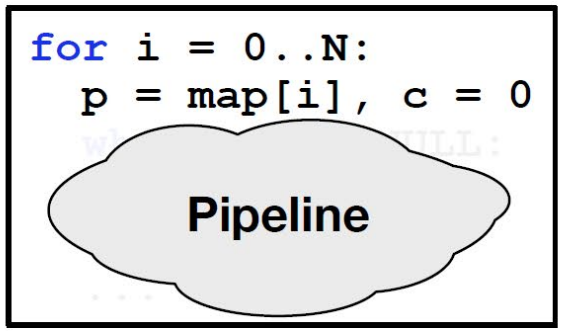


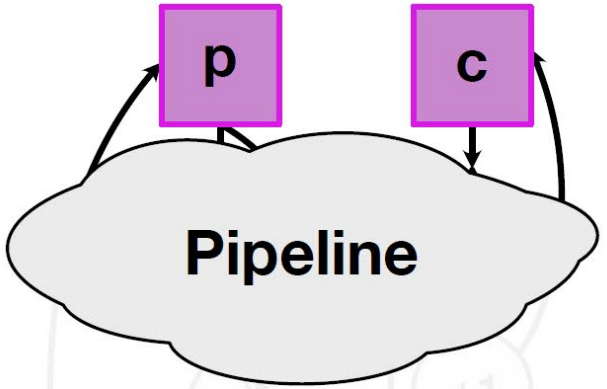
Carry gates manage loops

```
for i = 0..N:
  p = map[i], c = 0
  while p != NULL:
    if p.val: c++
    p = p->next
  ...
```

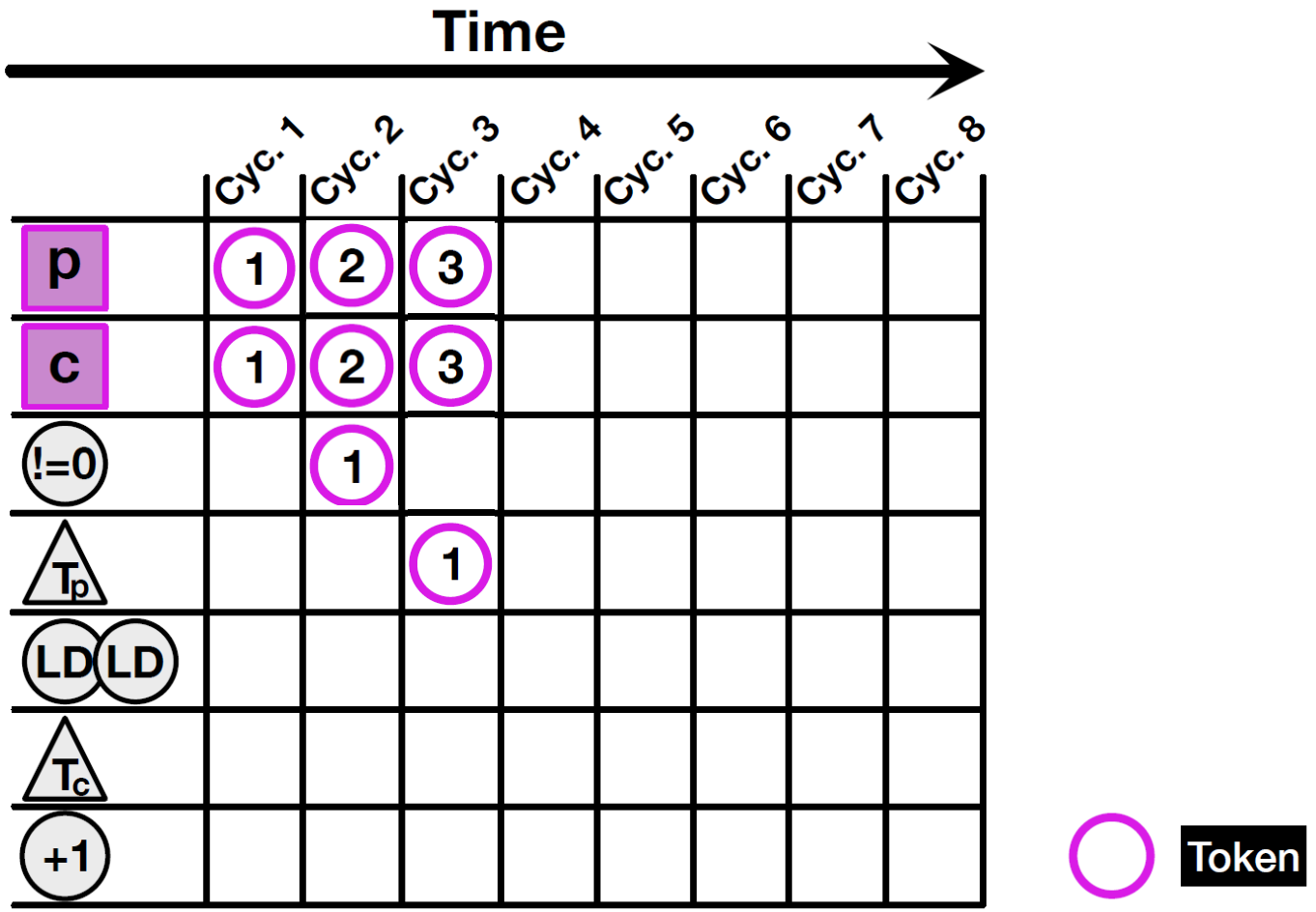


Carry gates manage loops

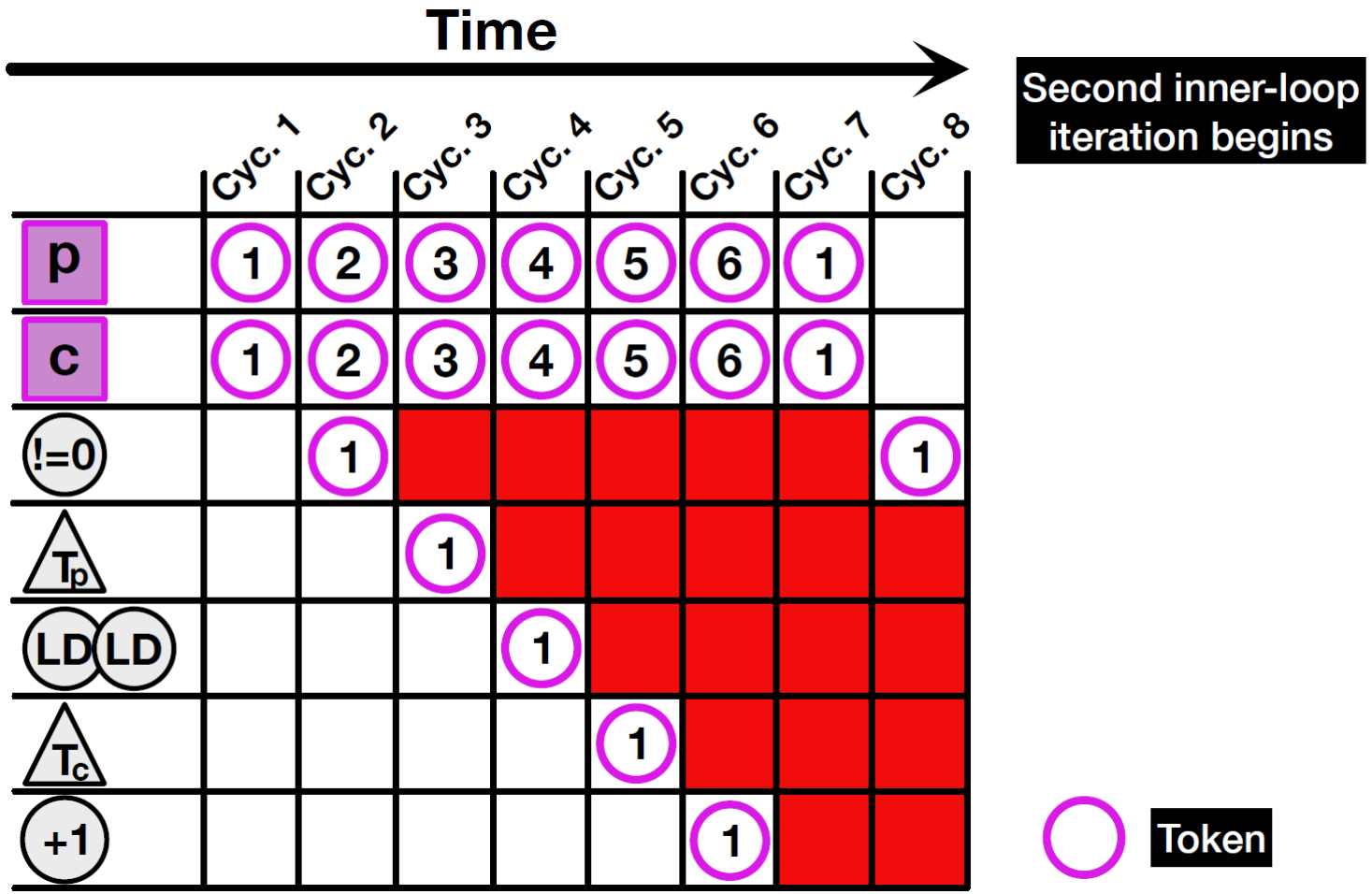




Carry sequentialization kills utilization

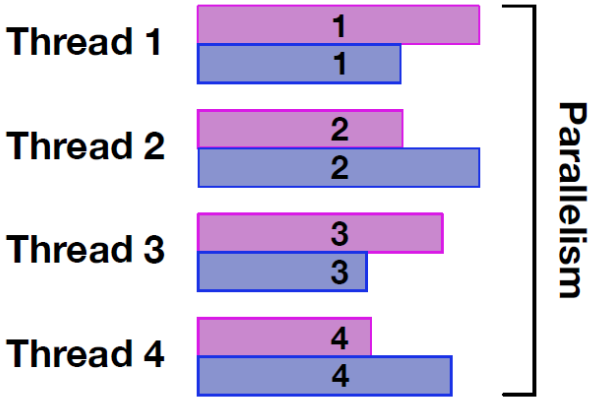


Carry sequentialization kills utilization



Pipestitch overview

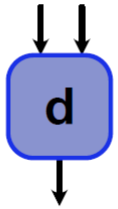
Lightweight dataflow threads



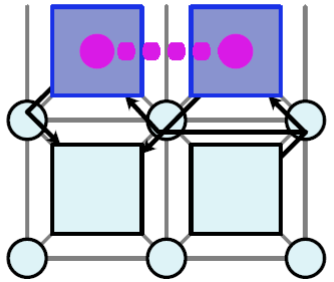
Expressed with parallel for

```
parallel for row = 0..R:
  acc = 0
  ...
  while iA < ARE && iB < BCE
    if a_col == b_col...
    if a_col <= b_col...
    if a_col >= b_col...
    acc = new_acc
    iA = new_iA
    iB = new_iB
```

Threads launched by new dispatch gate

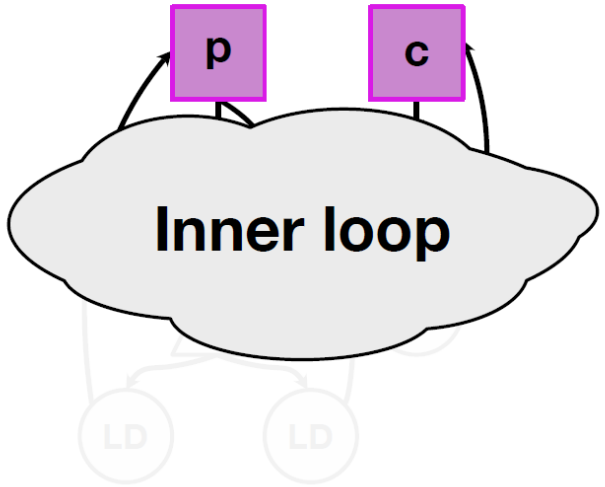


Sync network orders threads

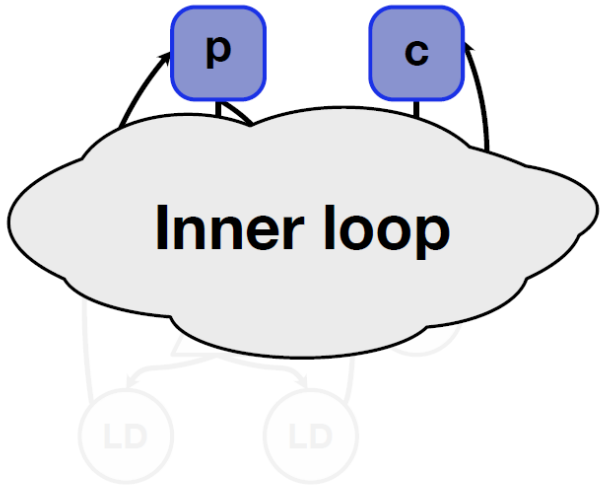


Dispatch replaces carry when threading

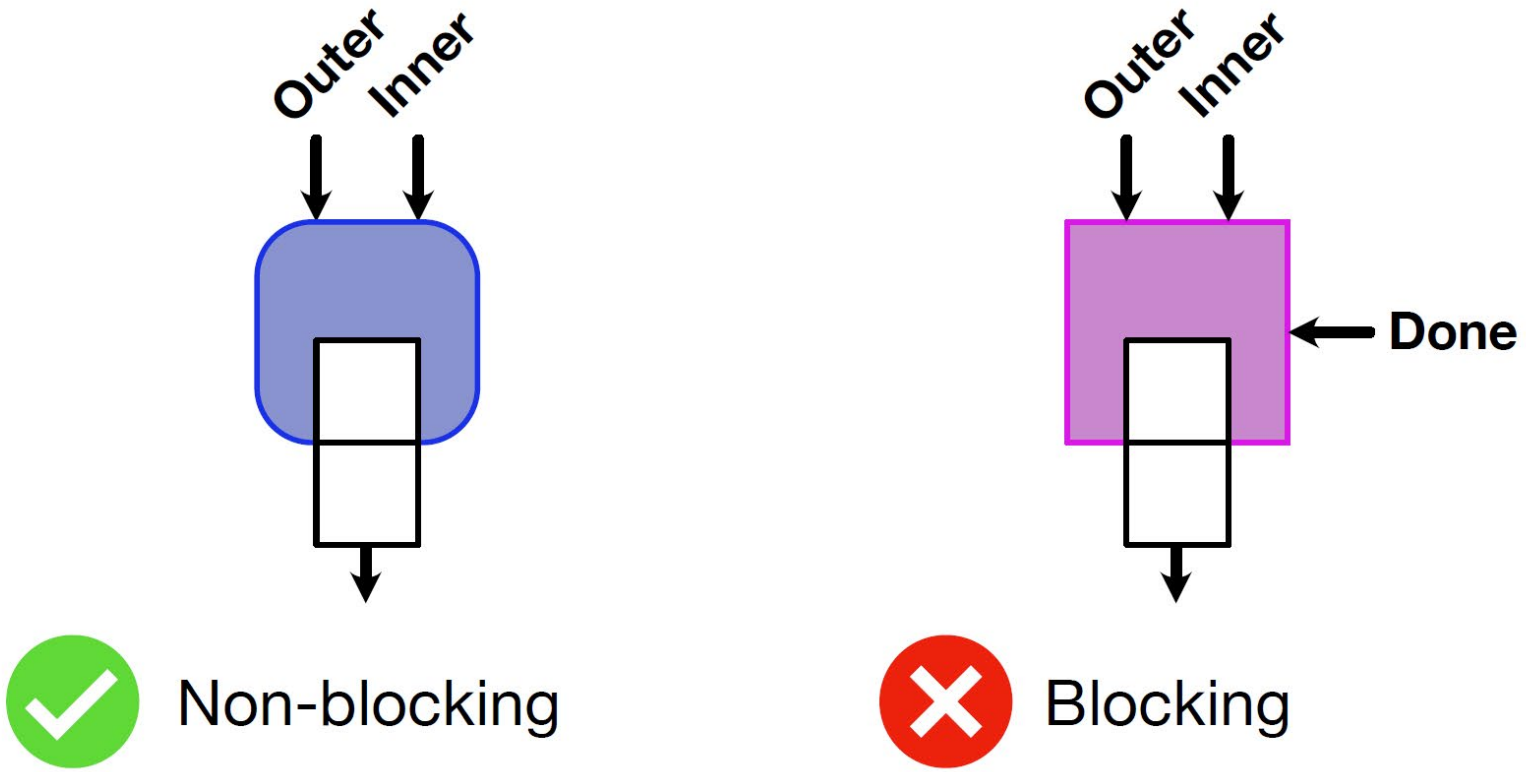
```
for i = 0..N:
  p = map[i], c = 0
  while p != NULL:
    if p.val: c++
    p = p->next
  ...
```



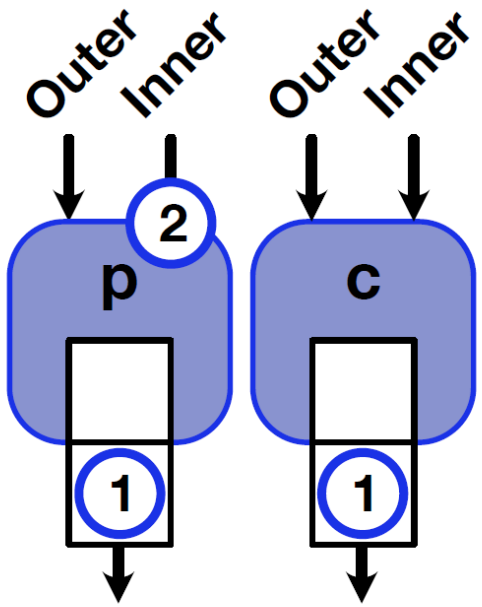

```
parallel for i = 0..N:
  p = map[i], c = 0
  while p != NULL:
    if p.val: c++
    p = p->next
  ...
```



Dispatch unblocks threads

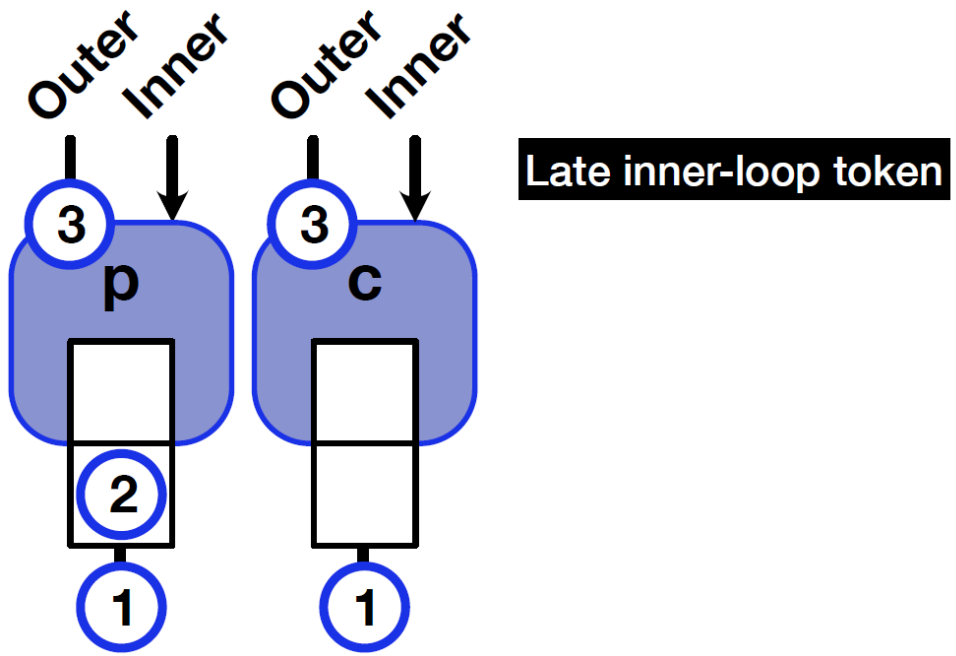


Ordering violated without synchronization

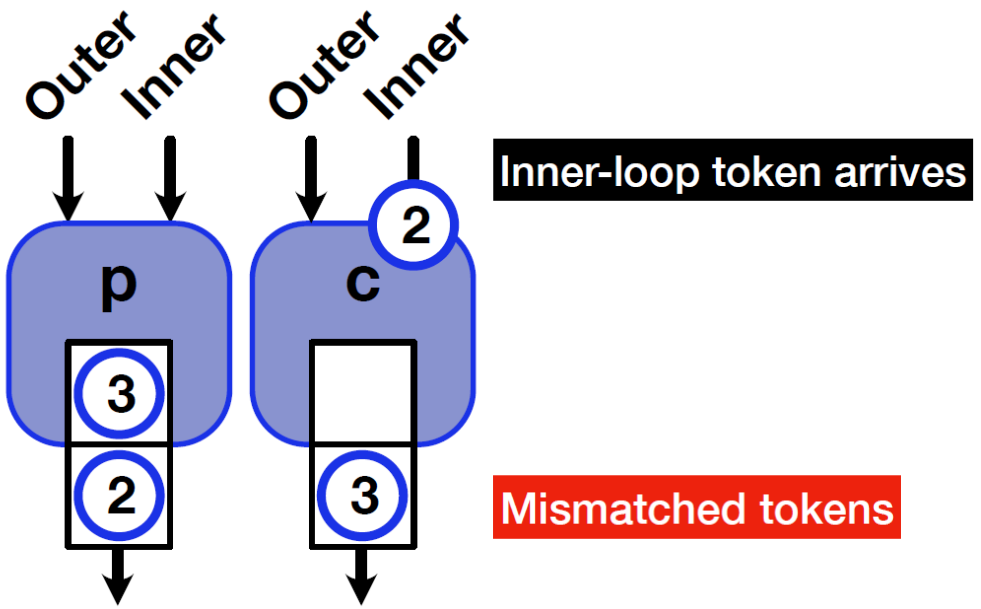


Late inner-loop token

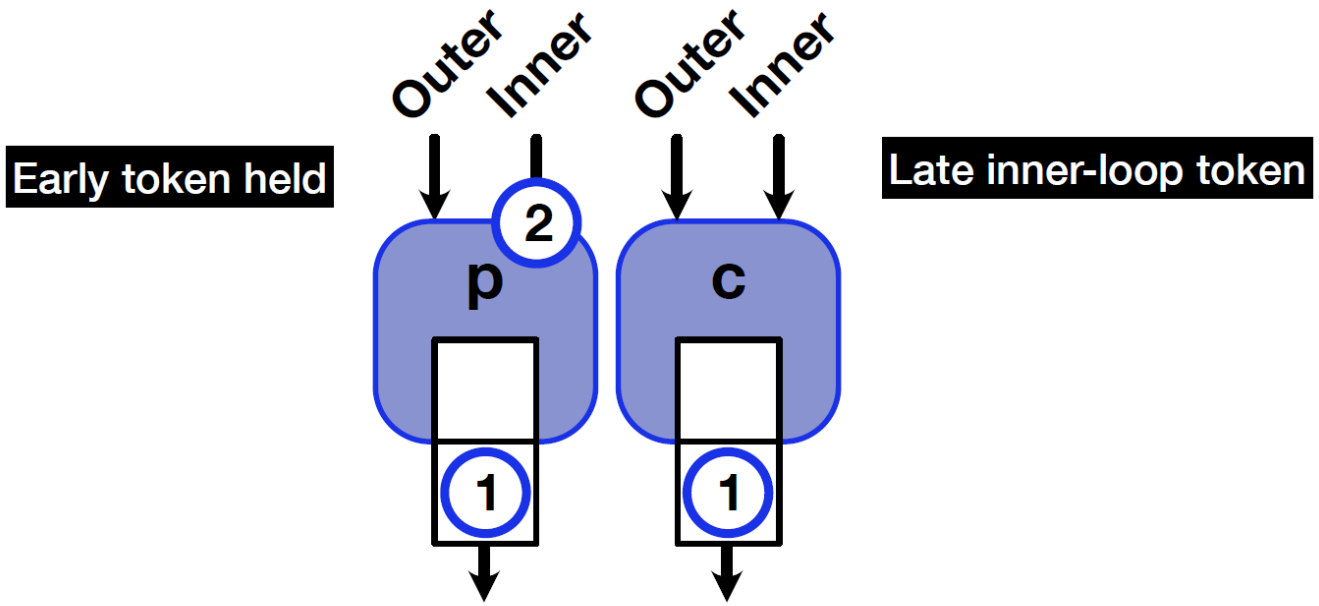
Ordering violated without synchronization



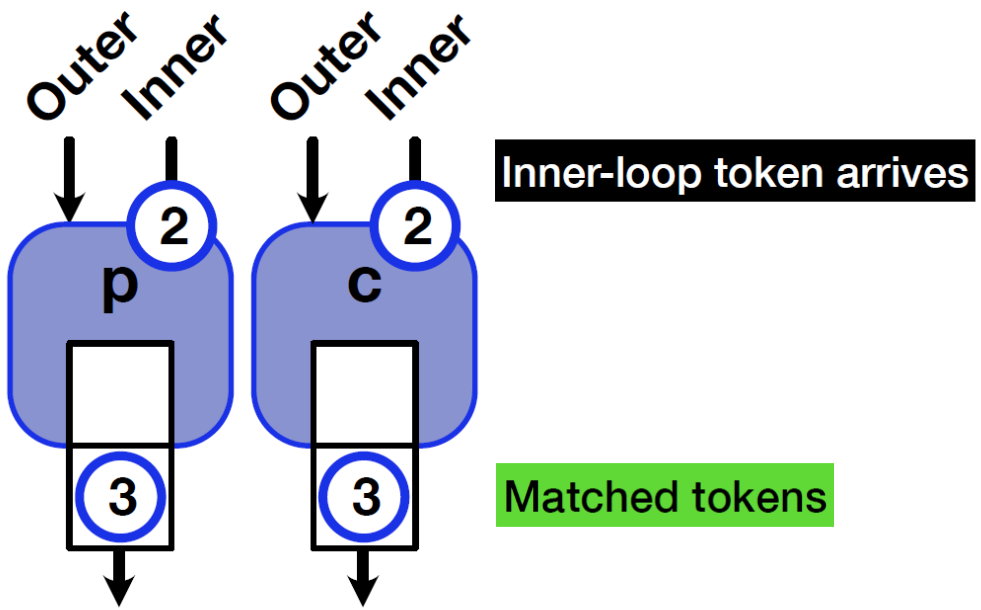
Ordering violated without synchronization



Synchronization fixes ordering violations

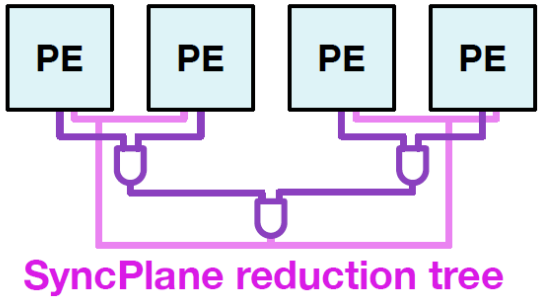


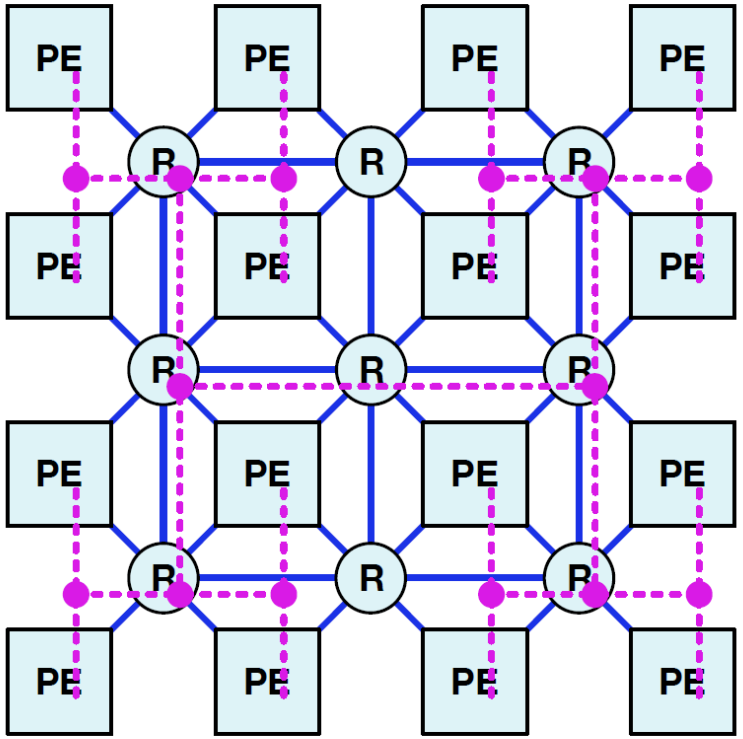
Synchronization fixes ordering violations



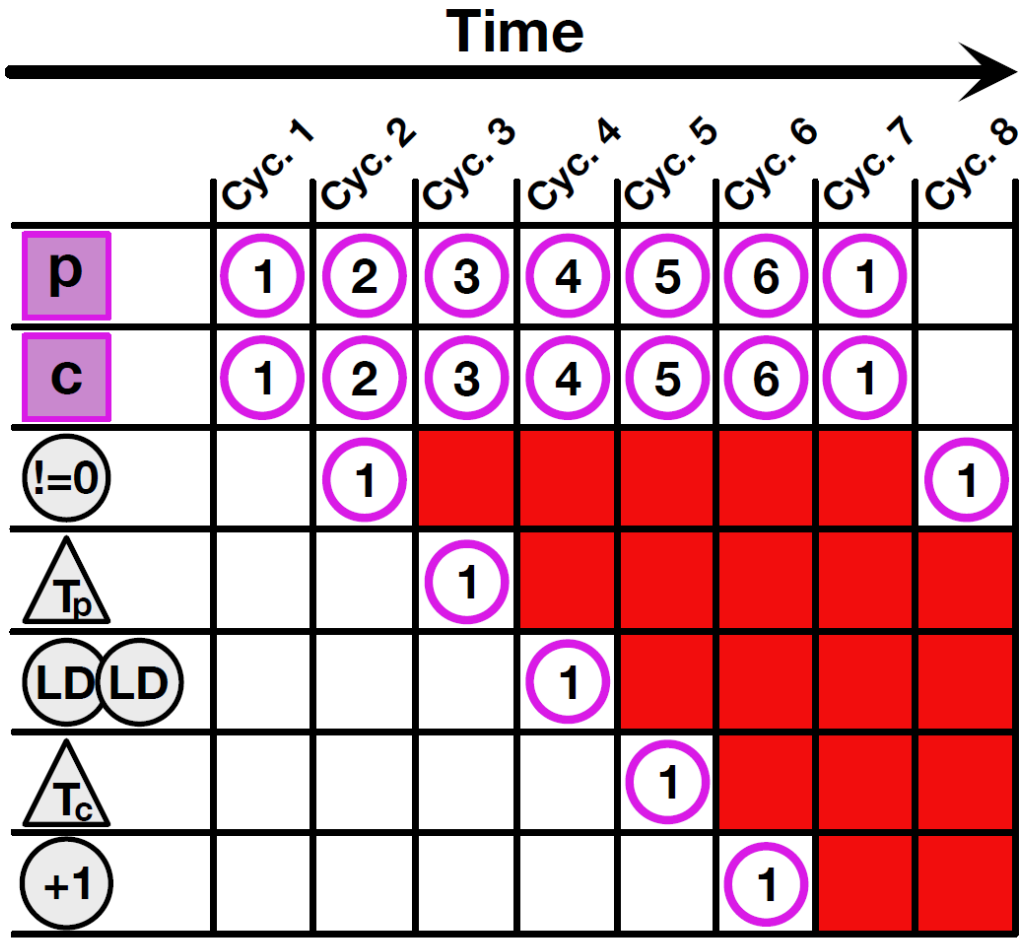
Synchronization accepts token groups

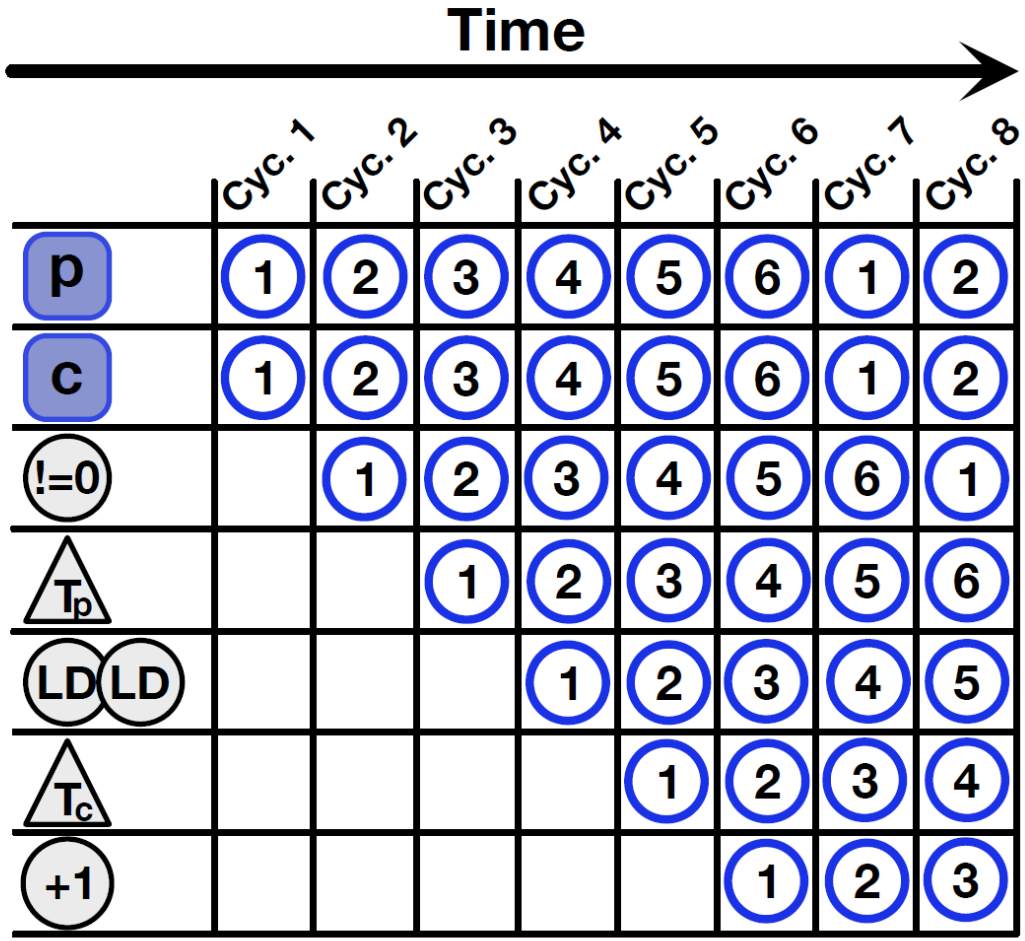
SyncPlane implements synchronization





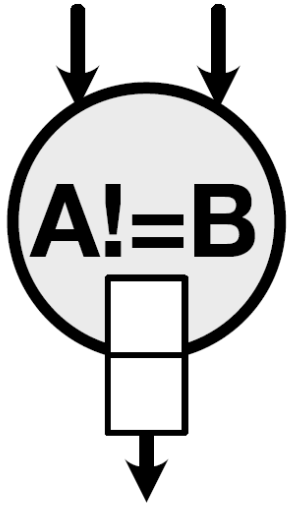
Pipestitch fixes utilization problem



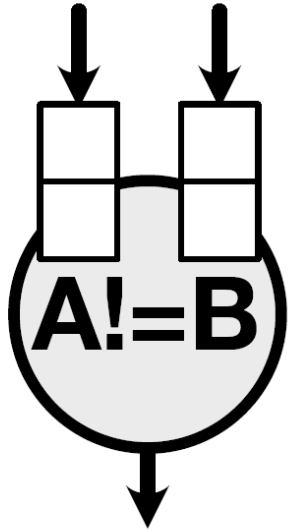


Choice of buffering

Source buffering



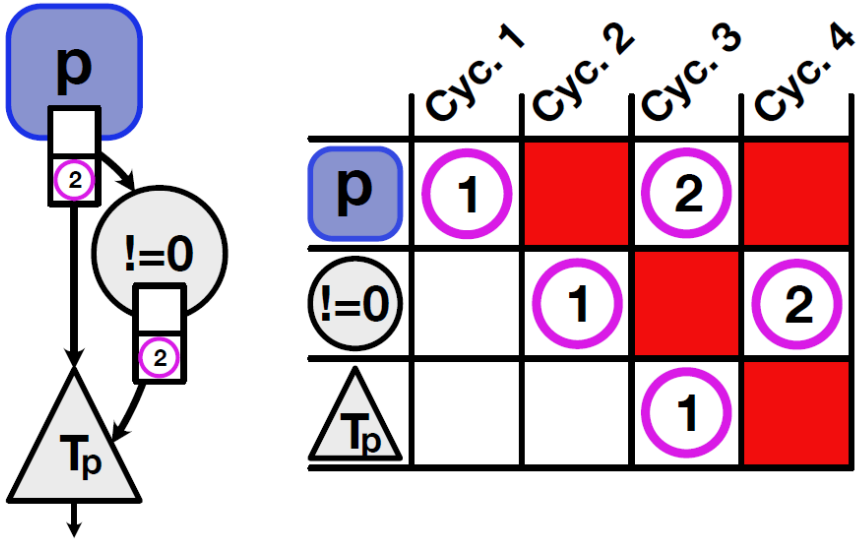
Destination buffering



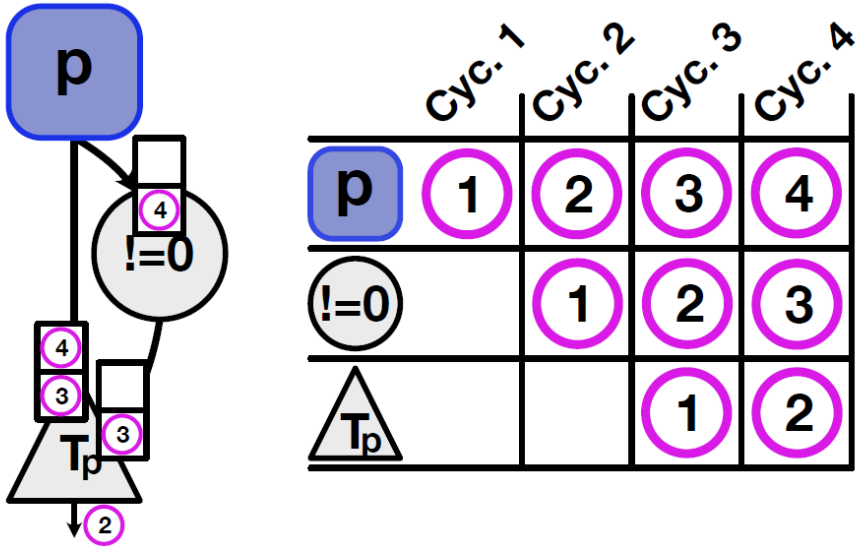
Source buffering saves energy

Destination buffering eliminates bubbles

Source buffering



Destination buffering

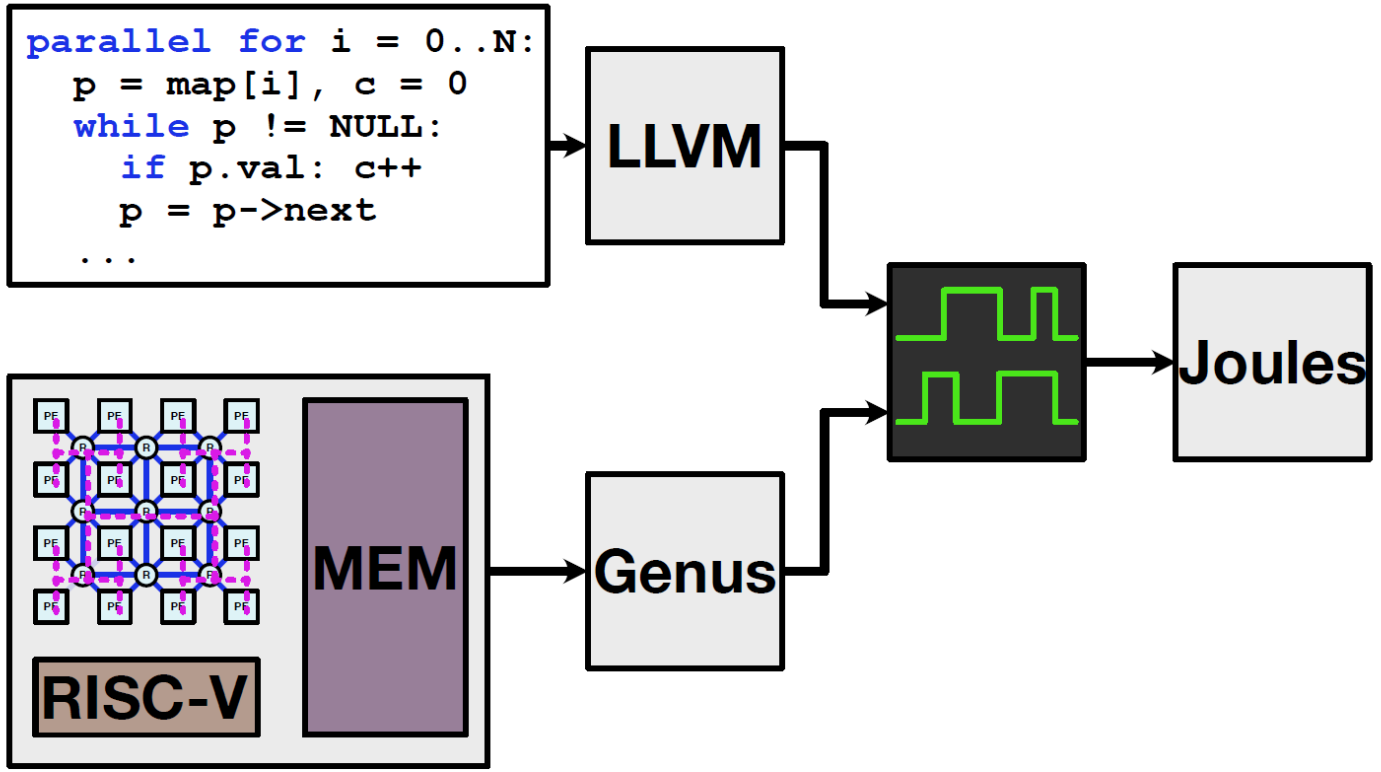


Discussion: Summary Question #1

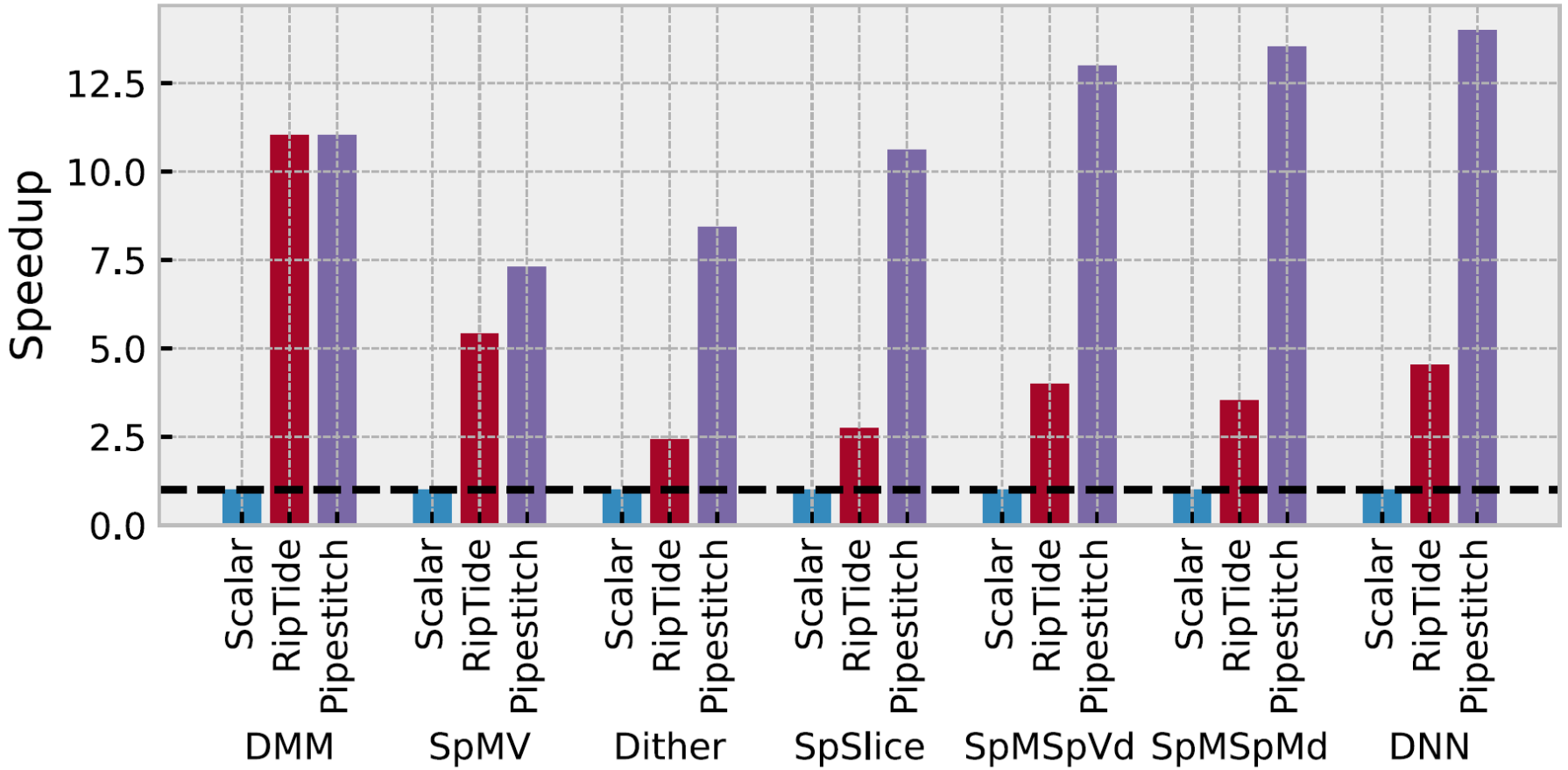
State the 3 most important things the paper says.

These could be some combination of the motivations, observations, interesting parts of the design, or clever parts of the implementation.

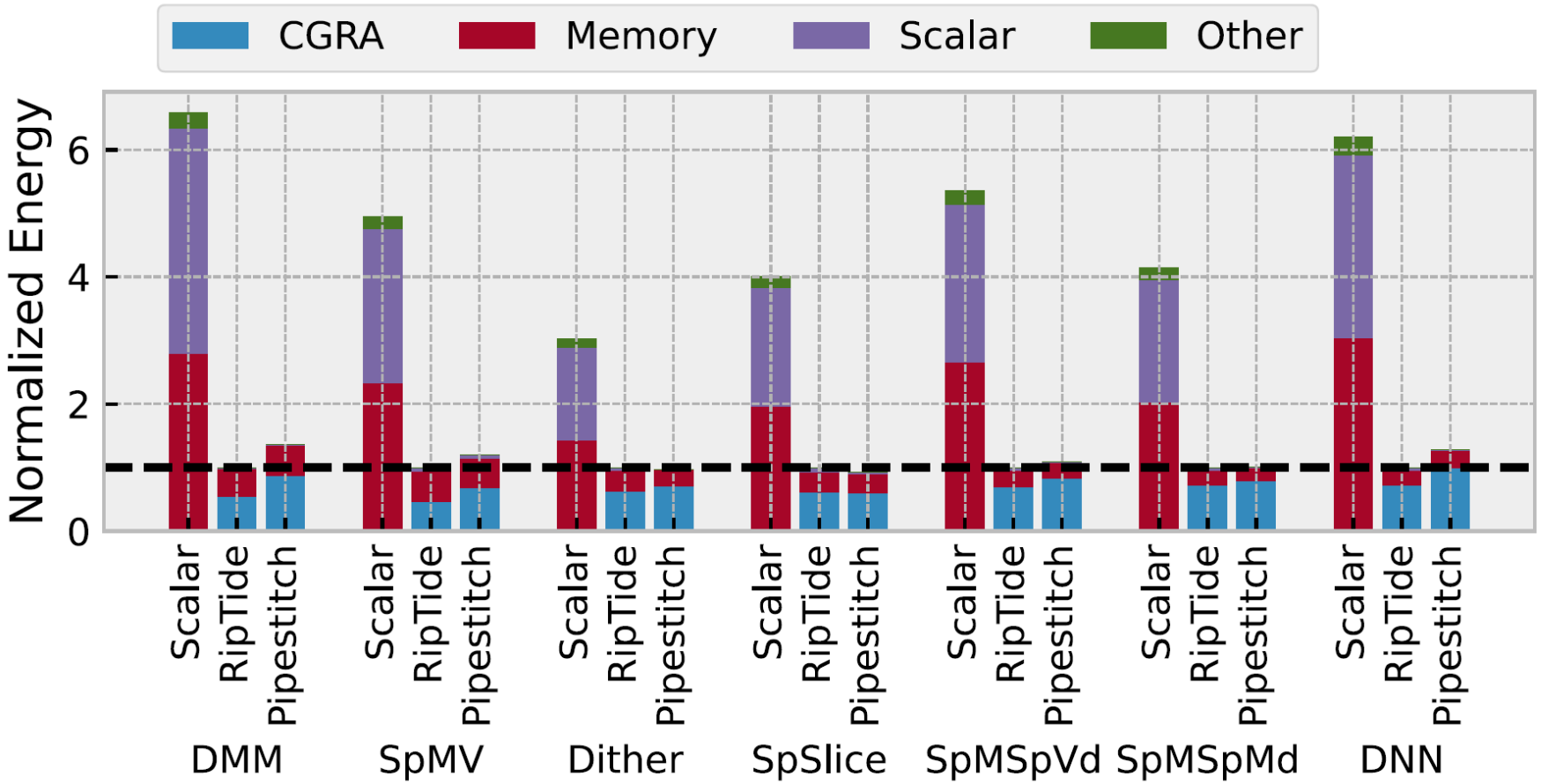
Methodology



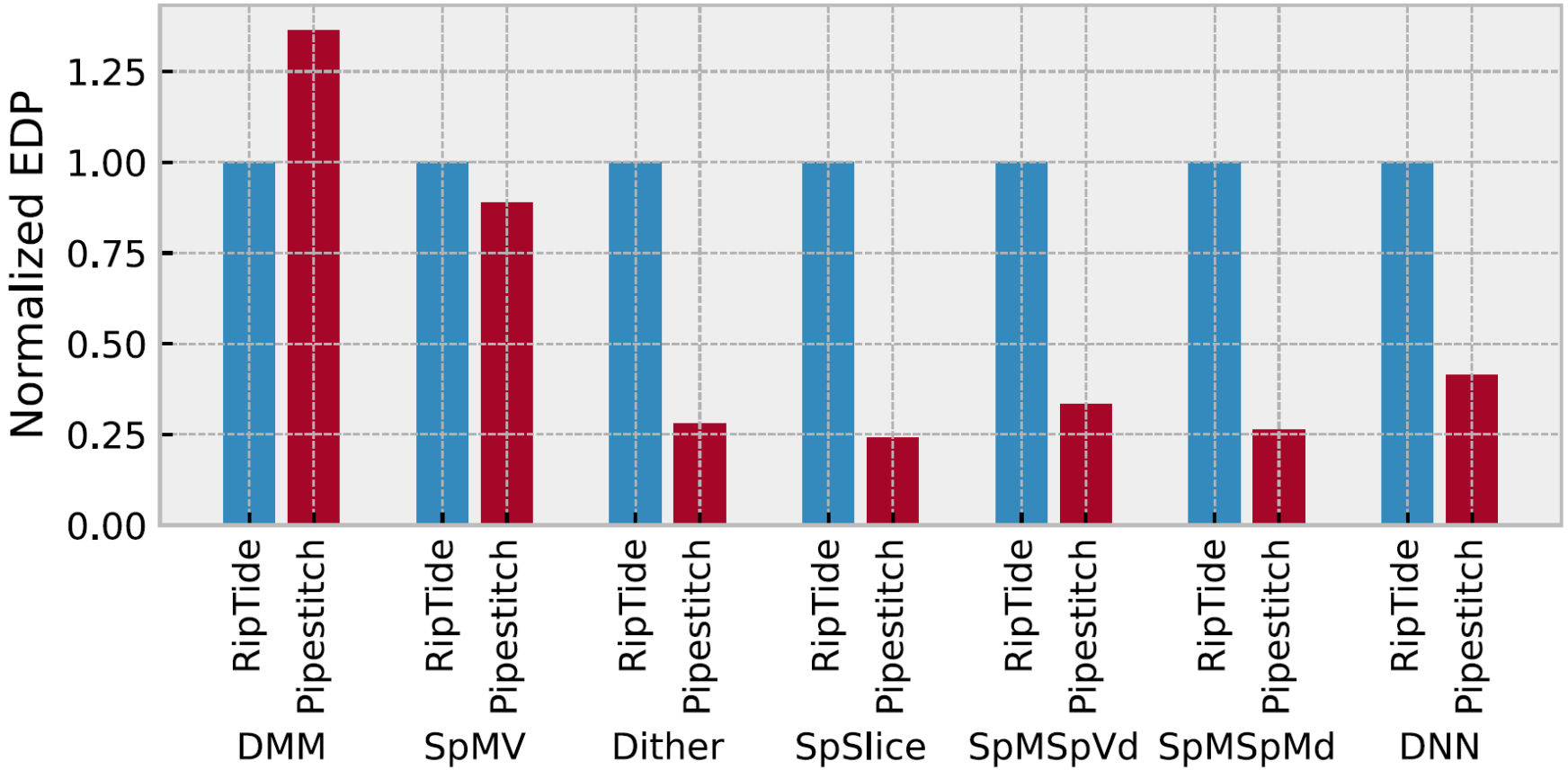
Pipestitch is fast



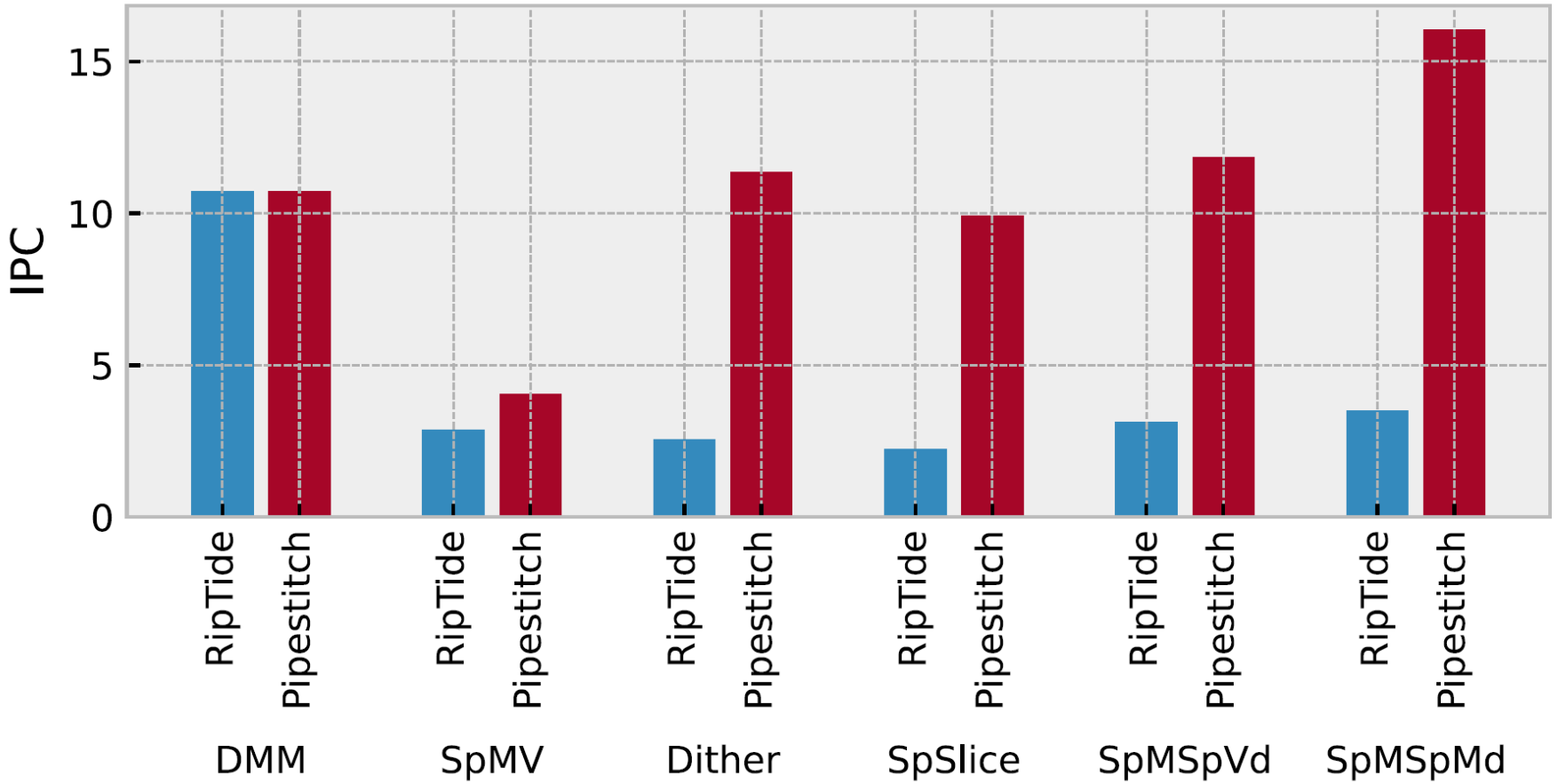
Pipestitch has little energy overhead



Pipestitch improves EDP for sparse



Pipestitch improves IPC



Discussion: Summary Question #2

Describe the paper's single most glaring deficiency.

Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.

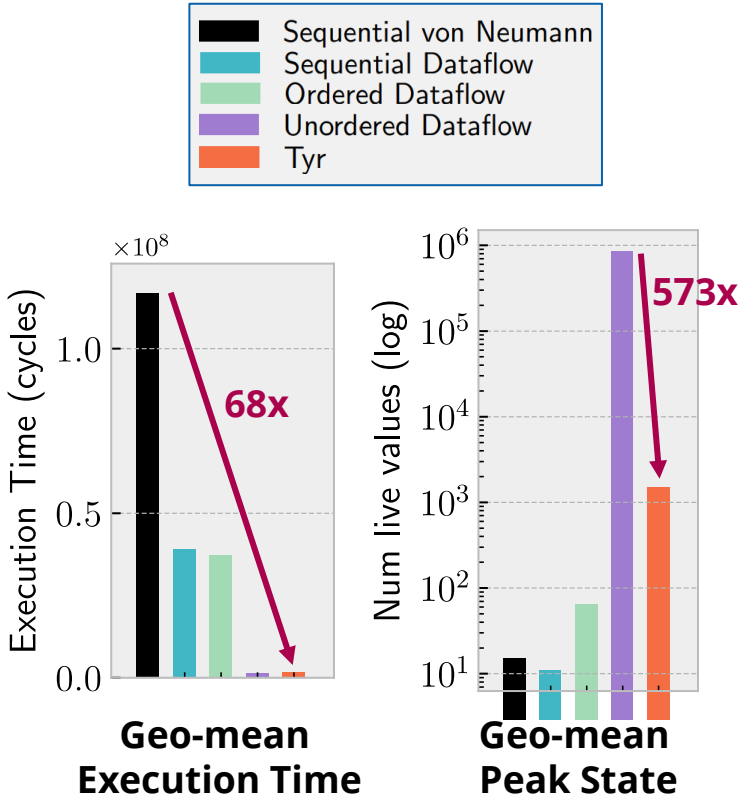
“The TYR Dataflow Architecture: Improving Locality by Taming Parallelism”

Nikhil Agarwal, Mitchell Fream, Souradip Ghosh, Brian C. Schwedock, Nathan Beckmann 2024

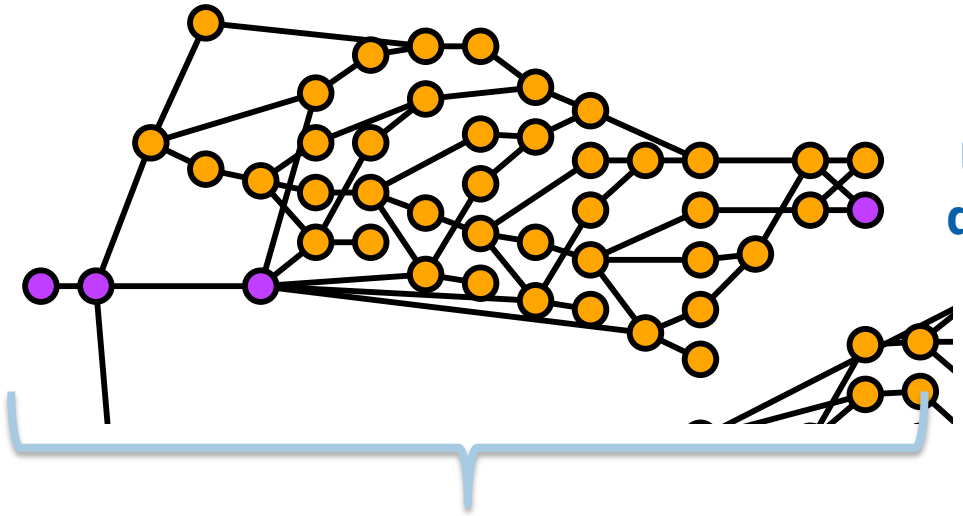
- Tagged dataflow architectures lead to large (unbounded) number of live tokens
- Finite machines can’t make use of unbounded parallelism anyway
- Selectively delaying work can massively decrease peak state without sacrificing parallelism

TYR Overview

- **Fundamental tradeoff: More parallelism ⇔ More state**
- **Dataflow architectures suffer *parallelism explosion***
  - Loss of locality or even deadlock!
- **TYR is the first general-purpose dataflow architecture with high parallelism and provably bounded state**
- **Insight: *Program structure* can be leveraged to break *unnecessary interdependences.***
- **Results: 68x faster vs. vN, 573x less state vs. naïve dataflow**



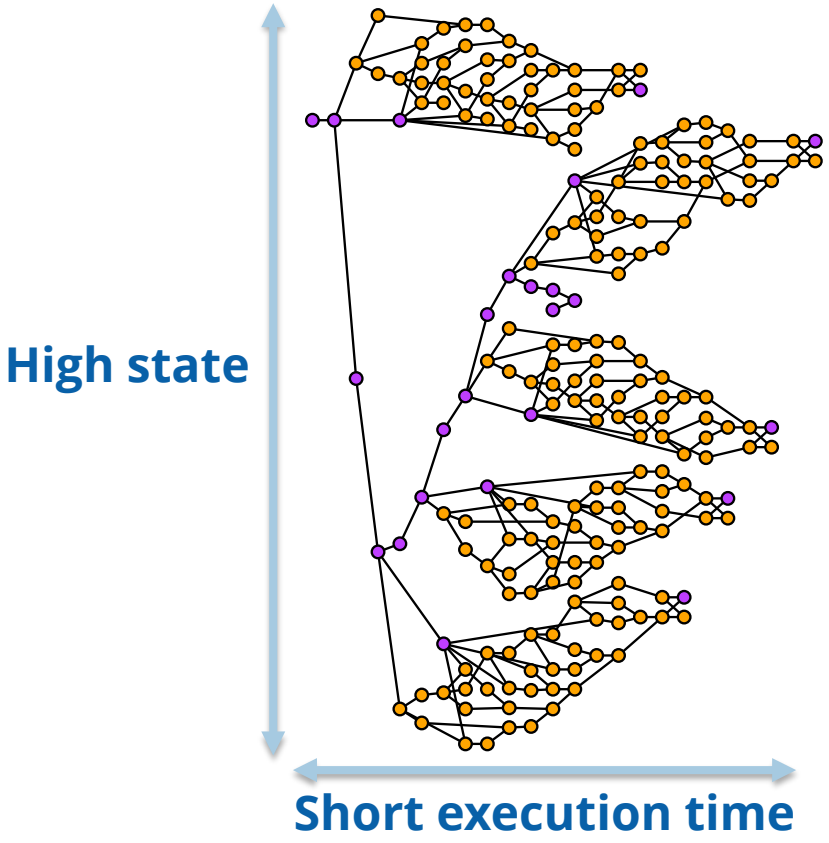
Unordered dataflow is highly parallel



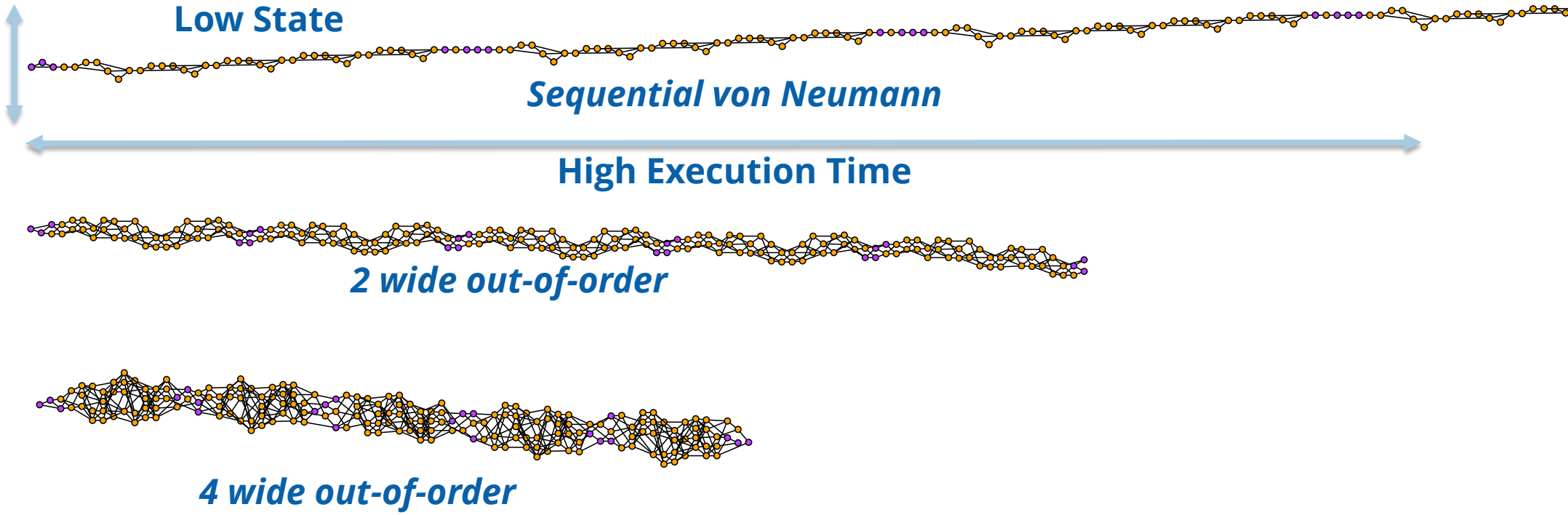
Unordered-tagged dataflow execution trace on 5x5 `dmv`

One outer loop iteration

Unordered dataflow is highly parallel

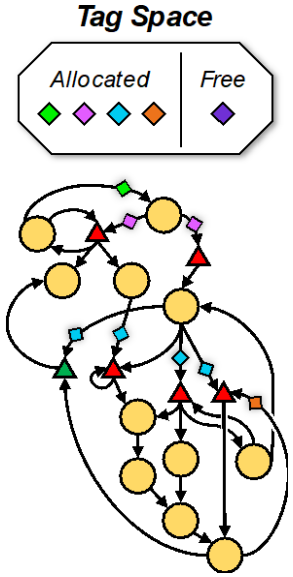


Von Neumann limits parallelism



The problem is over-synchronization

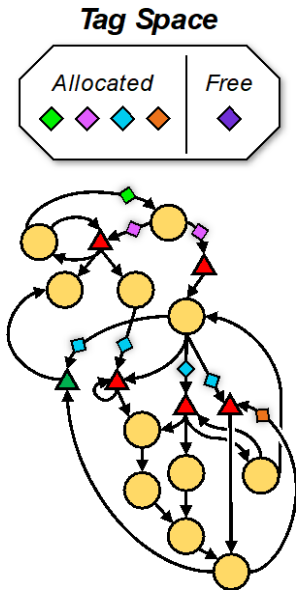
- Insight: A shared global tag space is unnecessary in unordered tagged dataflow.



Unstructured DFG

TYR's local tag spaces avoid over-synchronization

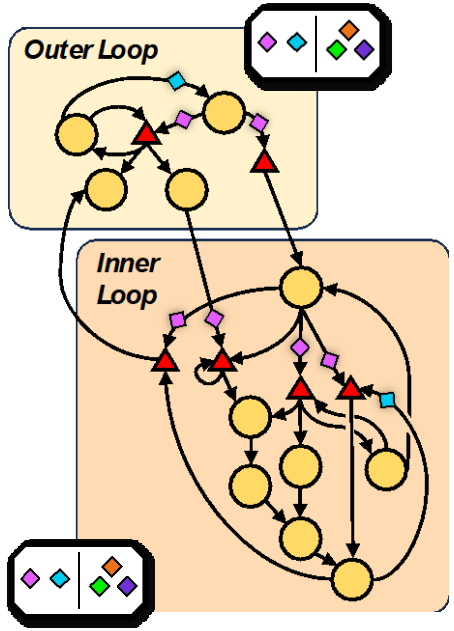
TYR's compiler breaks the program into blocks.



Unstructured DFG

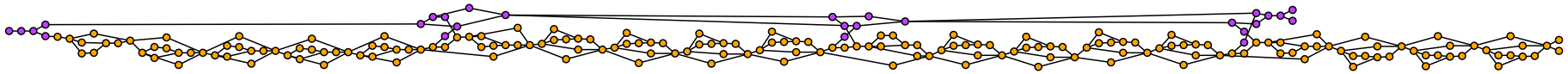
Split into blocks

Each block has its own *local tag space*.

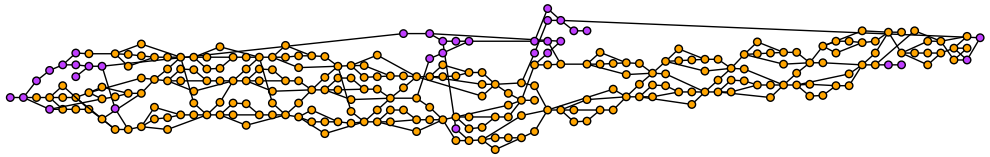


Structured DFG

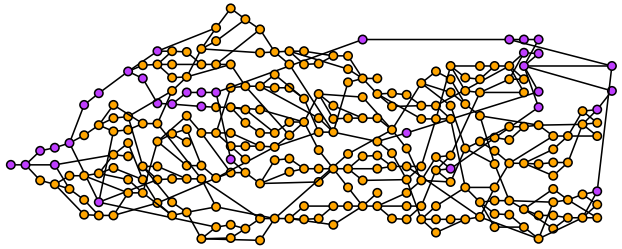
TYR can expand work as needed



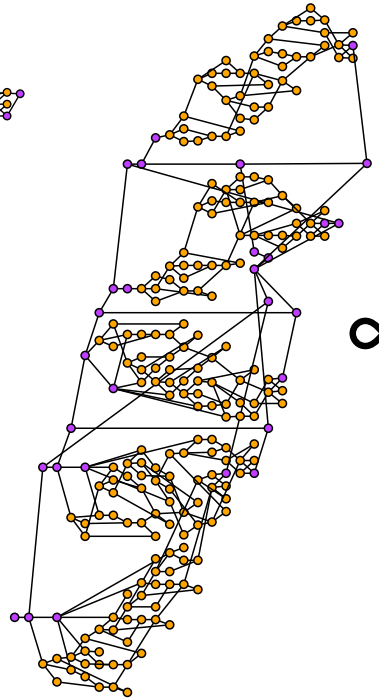
2 tags



4 tags



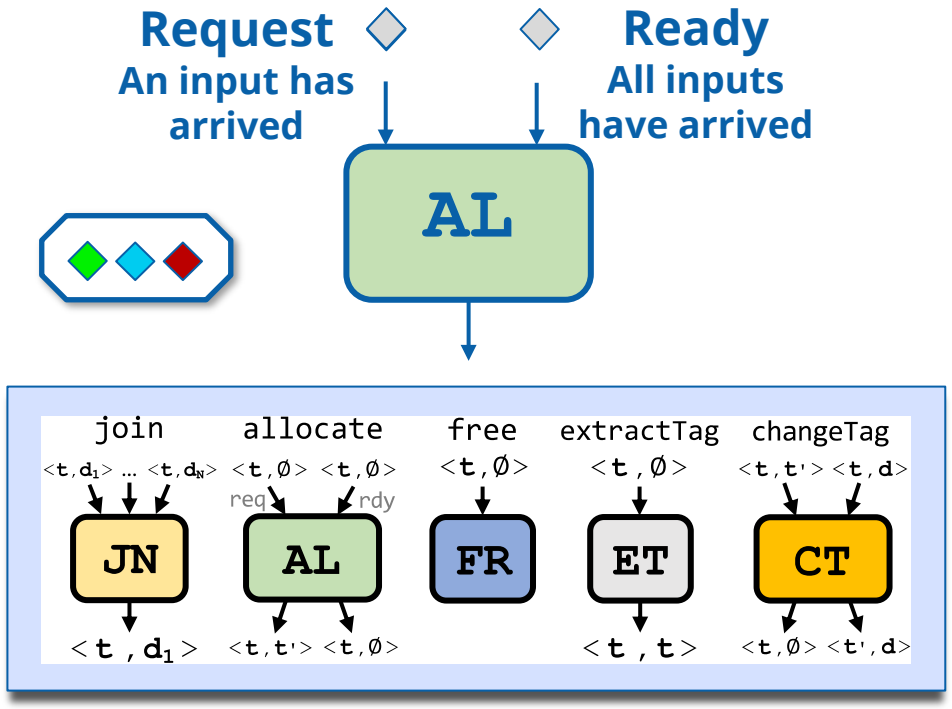
8 tags

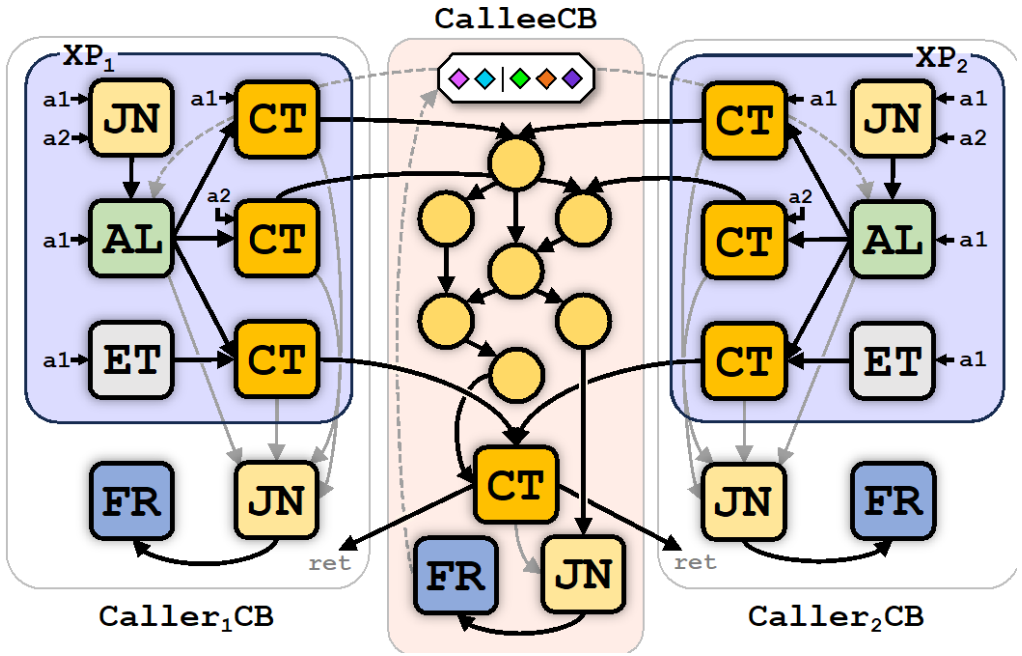


∞ tags

`dmv` *traces on TYR with different tag counts*

TYR's allocation rule





TYR guarantees deadlock freedom and bounded state

A. Proof of deadlock freedom

We first consider a program consisting of a single concurrent block, and then build up towards more complex programs.

Definition. *A leaf block is a concurrent block that does not contain*

B. Proof of bounded state

Prior unordered dataflow architectures

bound state because they are

live tags. TYR

to show that a

a tag. TYR only allocates

to a ready context, so there is alway

tag or (*ii*) a ready context already executing,

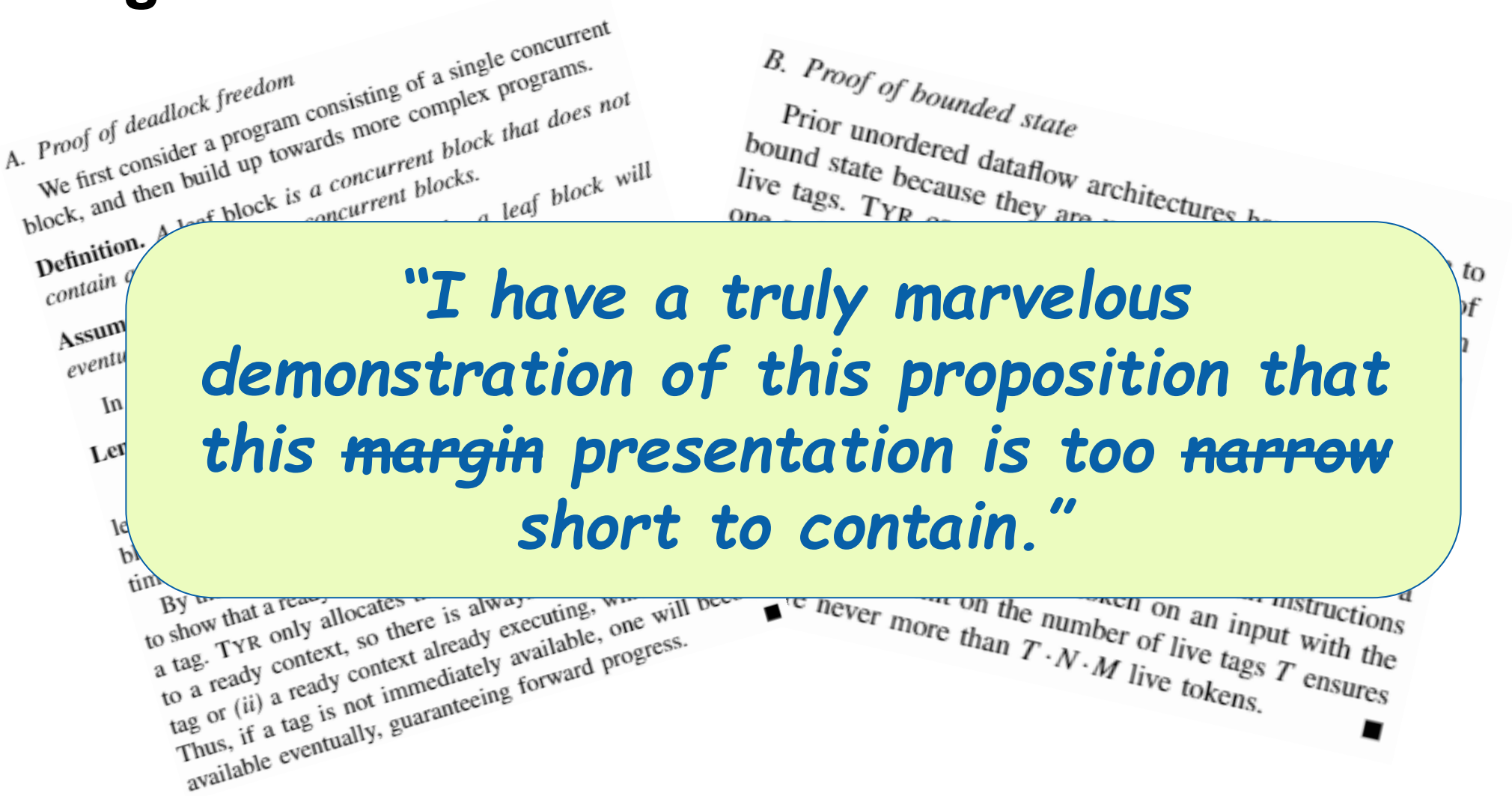
Thus, if a tag is not immediately available, one will be

available eventually, guaranteeing forward progress. ■

on the number of live tags T ensures never more than $T \cdot N \cdot M$ live tokens. ■

> *"I have a truly marvelous demonstration of this proposition that this margin is too narrow to contain."*
> *-Fermat*

TYR guarantees deadlock freedom and bounded state



A. *Proof of deadlock freedom*

We first consider a program consisting of a single concurrent block, and then build up towards more complex programs.

Definition. *A leaf block is a concurrent block that does not contain*

B. *Proof of bounded state*

Prior unordered dataflow architectures

bound state because they are

live tags. TYR

to show that a

a tag. TYR only allocates

to a ready context, so there is alwa

tag or (*ii*) a ready context already executing, wi

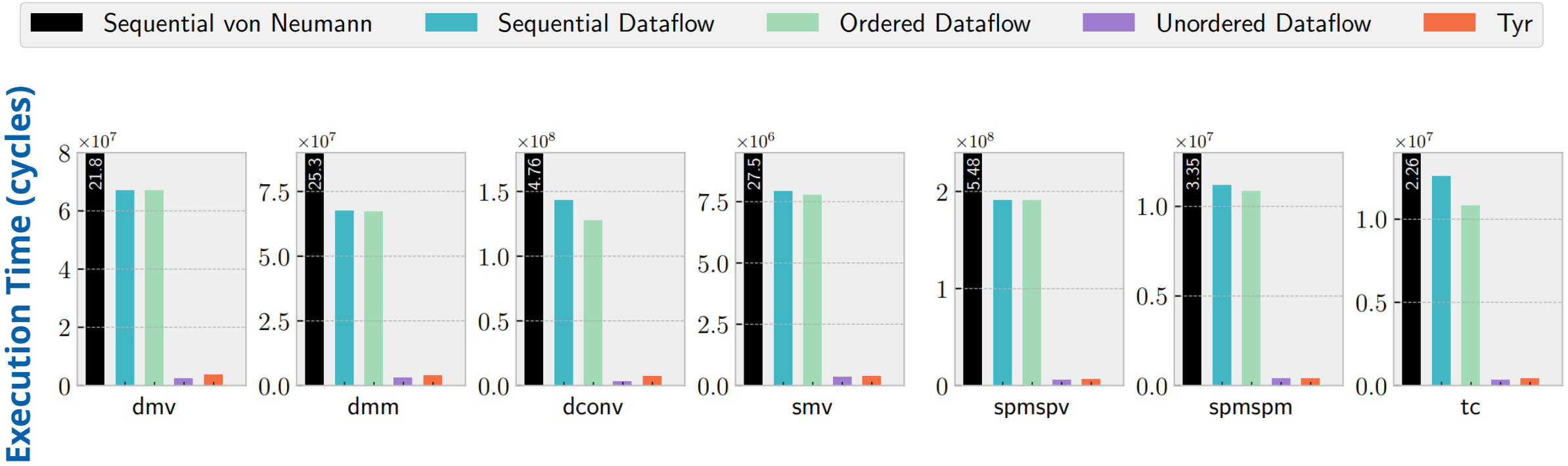
Thus, if a tag is not immediately available, one will bec

available eventually, guaranteeing forward progress. ■

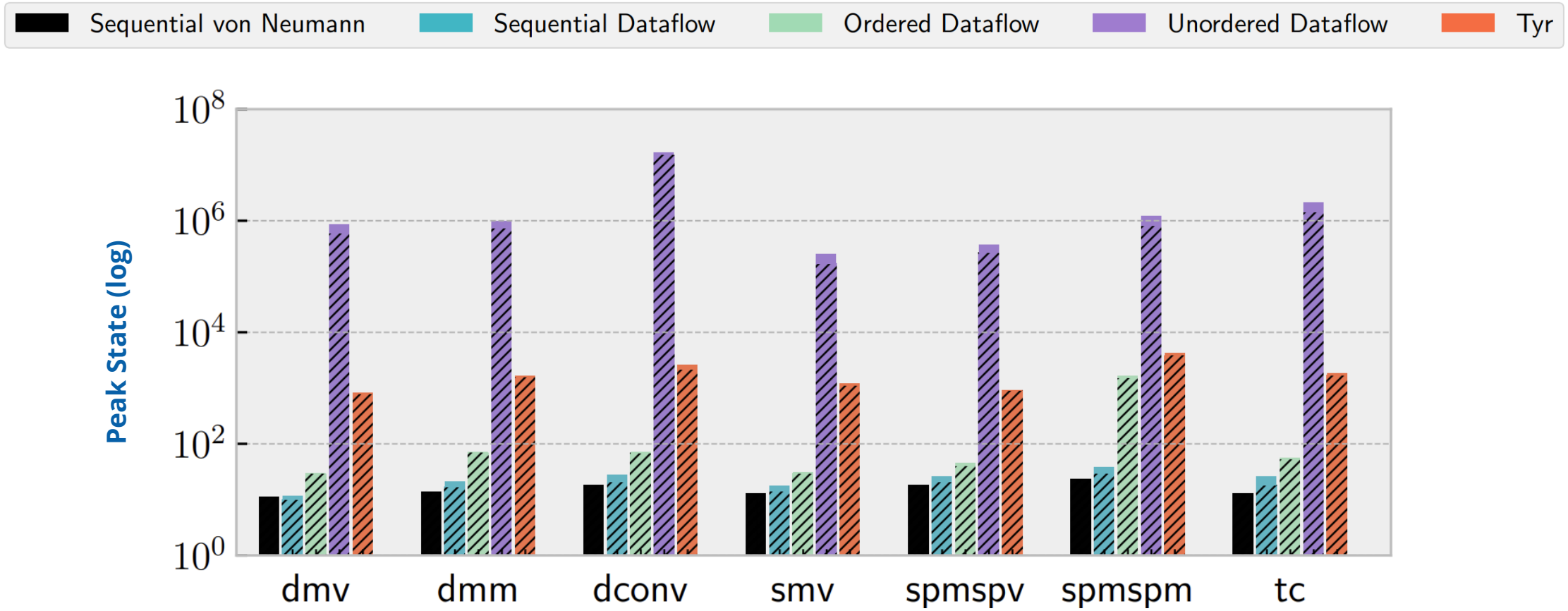
on the number of live tags T ensures never more than $T \cdot N \cdot M$ live tokens. ■

> *"I have a truly marvelous demonstration of this proposition that this ~~margin~~ presentation is too ~~narrow~~ short to contain."*

TYR is fast


Sequential von Neumann
Sequential Dataflow
Ordered Dataflow
Unordered Dataflow
Tyr
Execution Time (cycles)
dmv
dmm
dconv
smv
spmspv
spmspm
tc

TYR bounds live state



Friday

Project Discussion Meetings

Monday

Midterm