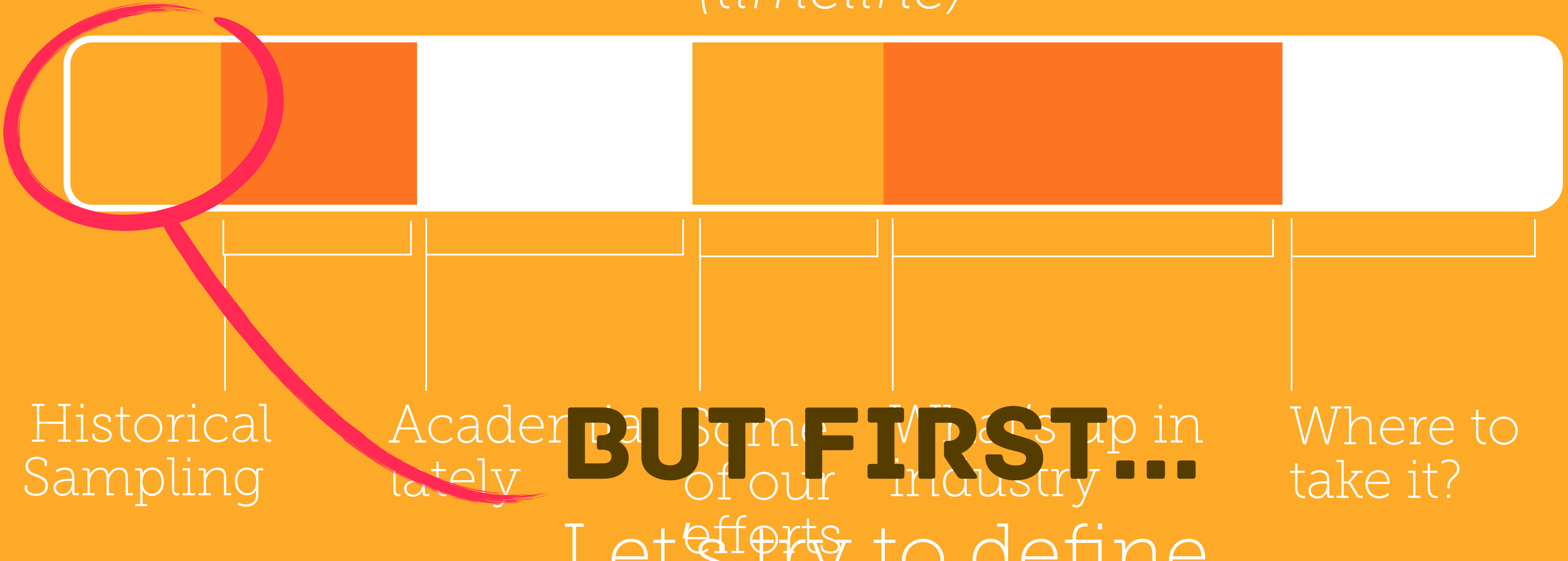
OPEN ISSUES *in* DATAFLOW

HEATHER MILLER
PHILIPP HALLER

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

This talk *(timeline)*



BUT FIRST...

Let's try to define "**Dataflow**"

DEFINING Dataflow

Seems easy enough, right?

Actually, **not really**

Creator of "flow-based programming" paradigm, when asked about relationship with dataflow:

> "It's just that, over the last several decades, so many different approaches all described themselves as data flow, that my feeling was that the term had become so broad as to become almost meaningless. You will find that much of the early work was done using this title, or phrases that included it.
>
> *Paul Morrison, 2010*

So, maybe it's best to agree that we'll probably never agree on an exact definition of "**dataflow**"

DEFINING *Dataflow*

So let's roll with something that most people can agree with.

(Thoughout this talk, I'll be tightening and loosening this definition)

DEFINING Dataflow (Loosely)

First pass: *(let's contrast with control flow)*

“ **In a control flow language,** you have a stream of instructions which operate on external data. Conditional execution, jumps and procedure calls change the instruction stream to be executed. This could be seen as instructions flowing through data

“ **In a dataflow language,** you have a stream of data which is passed from instruction to instruction to be processed. Conditional execution, jumps and procedure calls route the data to different instructions. This could be seen as data flowing through otherwise static instructions like how electrical signals flow through circuits or water flows through pipes.

http://stackoverflow.com/questions/461796/dataflow-programming-languages/949771#949771

More precisely...

DATAFLOW ALWAYS:

- Program represented by a directed graph.
- Nodes of the graph represent operations.
- The edges between the nodes represent **data dependencies**. (FIFO)
- Conceptually, data flows along the edges.

More precisely…

DATAFLOW USUALLY:

- Deterministic
- Based on single-assignment values/collections
- Lightweight concurrency
- Parallelism implicit, thanks to data dependencies
- Extension of functional programming

Concurrent. Declarative.

FOCUS: CONCURRENT/PARALLEL

- FP extended with (**lightweight**) threads and dataflow v **nt**)
- ***Determin*** ecution always gives the same results (tions don't terminate normally)

OZ-LIKE

LIMITED: CAN'T MODEL CLIENT/SERVER

ADVANTAGES:

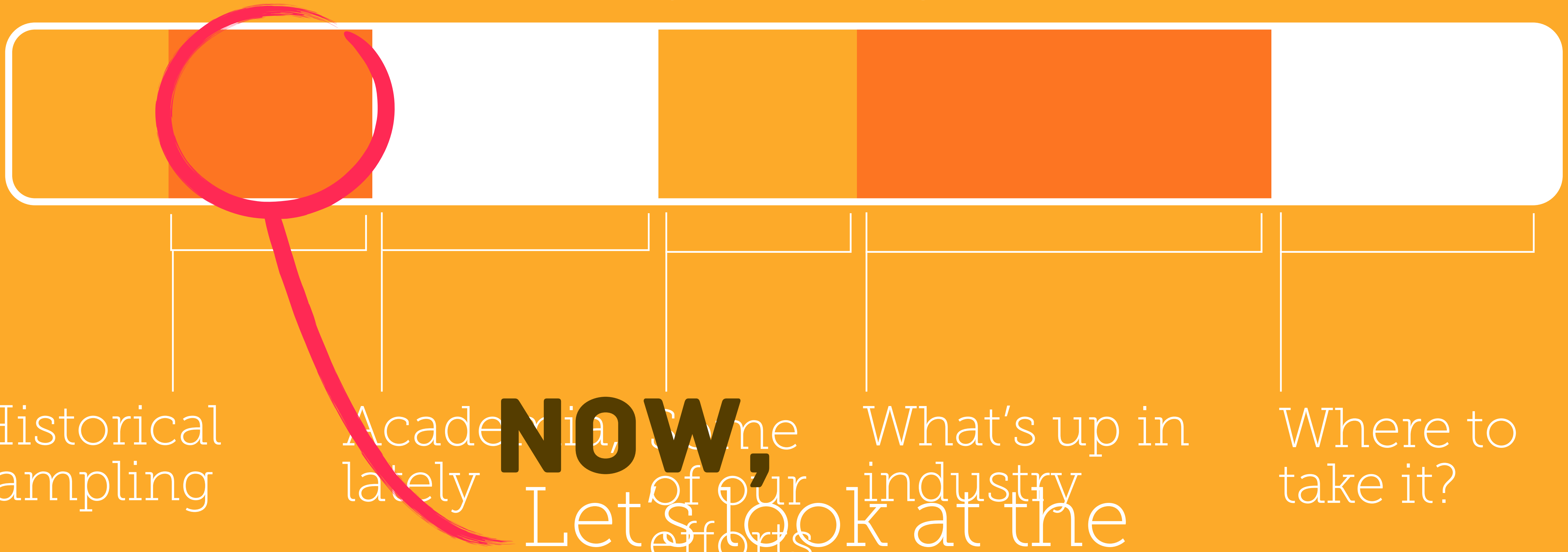
- Race conditions impossible
- Implicit parallelism for FP code

EXAMPLE Ozma

```
val x = future(1)
val y = future(2)
val z = future(x + y)
println(z)
```

- The type of `x` is `Int`, not `Future[Int]`
- Futures are lightweight tasks, not OS threads
- Instead of blocking, post/register continuation with future's remaining job to dataflow variable

This talk *(timeline)*



NOW,

Let's look at the motivation behind Dataflow Research

GLIMPSE INTO Dataflow History

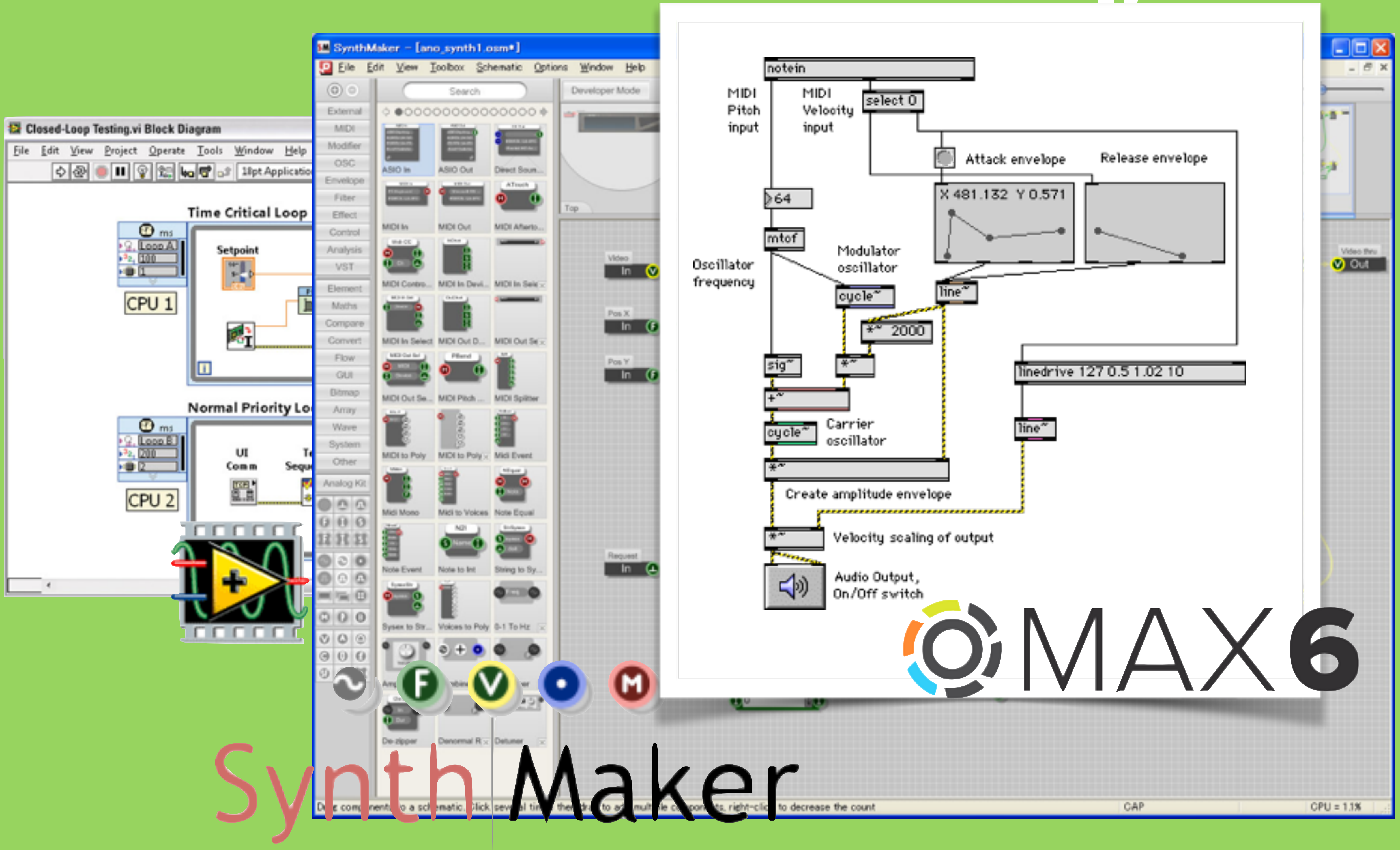
- 70-80s: dataflow computer architectures. Lead to need for new dataflow languages.
- Due to required properties of dataflow languages, the choice of paradigm was **functional.**

  (freedom from side effects, effect locality, single assignment)

GOAL THEN: EXPLOIT PARALLELISM IN A NATURAL TO PROGRAM WAY

Similar to today, right? But then, special dataflow architectures were required, and parallel architectures were far from ubiquitous.

THIS STUFF IS ALSO Dataflow



GLIMPSE INTO *Dataflow History*

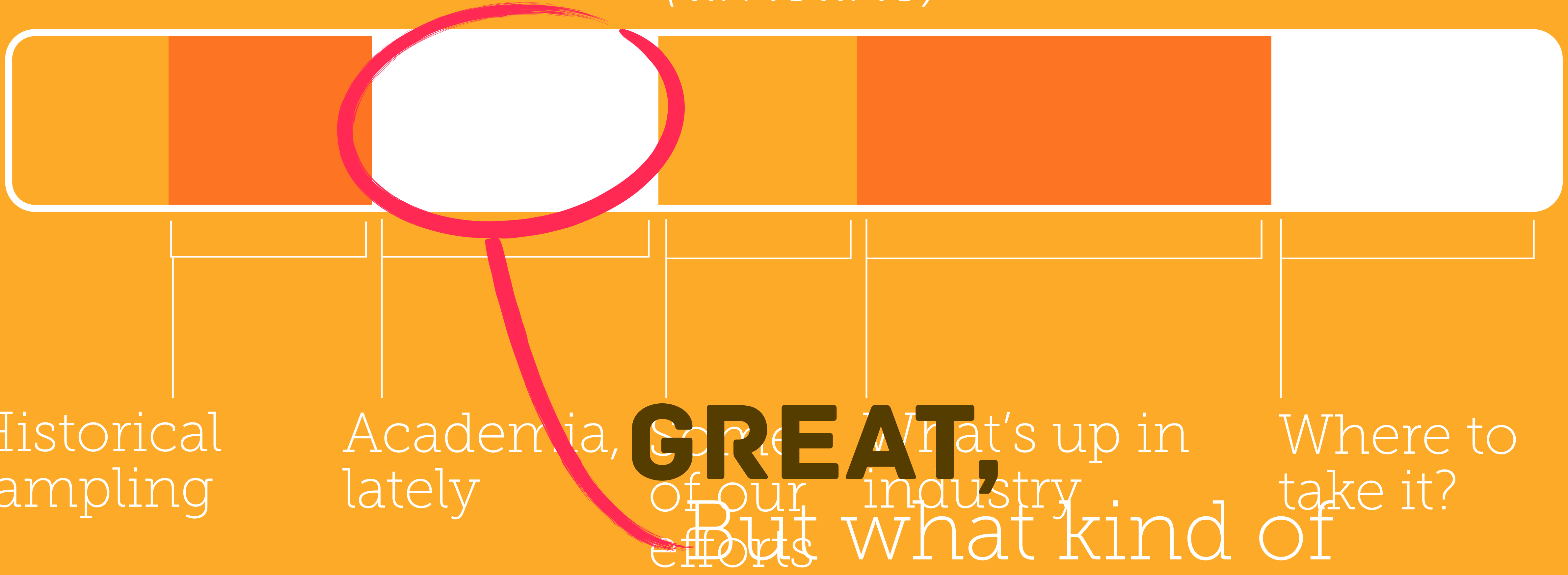
- 90s: cost-effective dataflow hardware did not materialize, so for parallelism, dataflow seemed lost.

Shift to make use of these dataflow ideas in the form of **visual dataflow programming languages**.

BUT NOW: WE STILL WANT TO EXPLOIT PARALLELISM IN A NATURAL TO PROGRAM WAY

- **Today:** attempts to provide dataflow-esque models on **modern general-purpose platforms**, attempts to distribute dataflow

This talk *(timeline)*



GREAT,

But what kind of dataflow research has academia been up to lately?

Why Do We Care?
(About Dataflow Now)

- Potential to simplify parallel programming
  - No race conditions
  - Simple debugging

- Smooth transition from standard FP

GLIMPSE INTO Current Dataflow Work

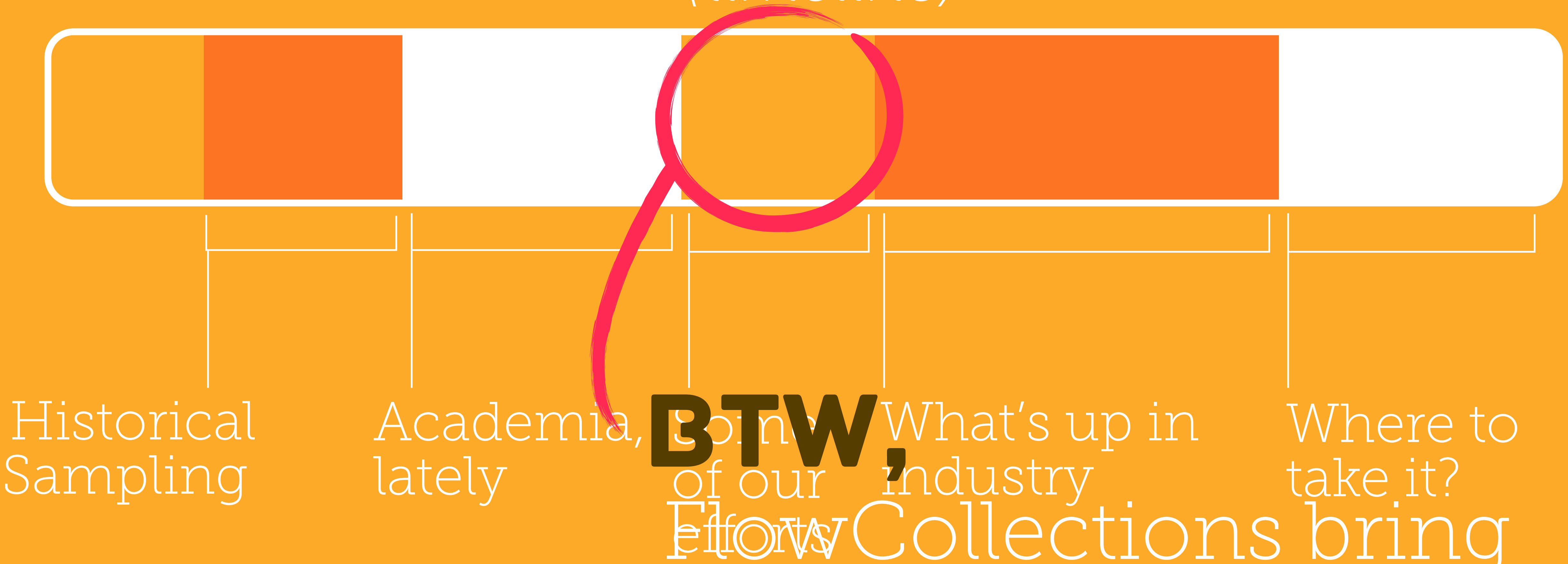
- Provide dataflow programming models in mainstream languages (Java, C++)
- Distribute dataflow (e.g., CnC)

OPEN QUESTION:

- **Can we/should we completely decouple from languages and compilers?**

  (1) DSLs, (2) modern languages good enough?, (3) middle ground, language design

This talk *(timeline)*



BTW,

FlowCollections bring some nice properties to the table

Dataflow Collections

- Collections of dataflow variables
  - *E.g.*, for number crunching
- **Problem:**
  - Creating a dataflow variable per data element prohibitively expensive (allocation + indirection + GC overhead)
- **Idea:** dedicated dataflow collections
  - Deterministic (consistent with classic dataflow)
  - Lock-free

FlowSeqs INTERFACE

- In order to guarantee determinism in our library-based framework, had to introduce the following interface.
  - Append (**<<**), concurrent insert
  - `foreach`, register callbacks (that is, take a function and apply it to all elements). Returns a `Future[Int]`, completed with the # elements processed
  - `aggregate`, like fold, includes operator which combines aggregations and returns a Future[] representing the final aggregation
  - `seal`, disallows further appends, discards registered foreach operations, allows aggregate to complete.

FlowSeqs

Prokopec et al., *A Generic Parallel Collection Framework*, EuroPar'11

- Ordered sequences with parallel bulk operations
- **Related**: Scala's parallel collections

```
val res: ParSeq[Int] = myList.par.map(transform)
```

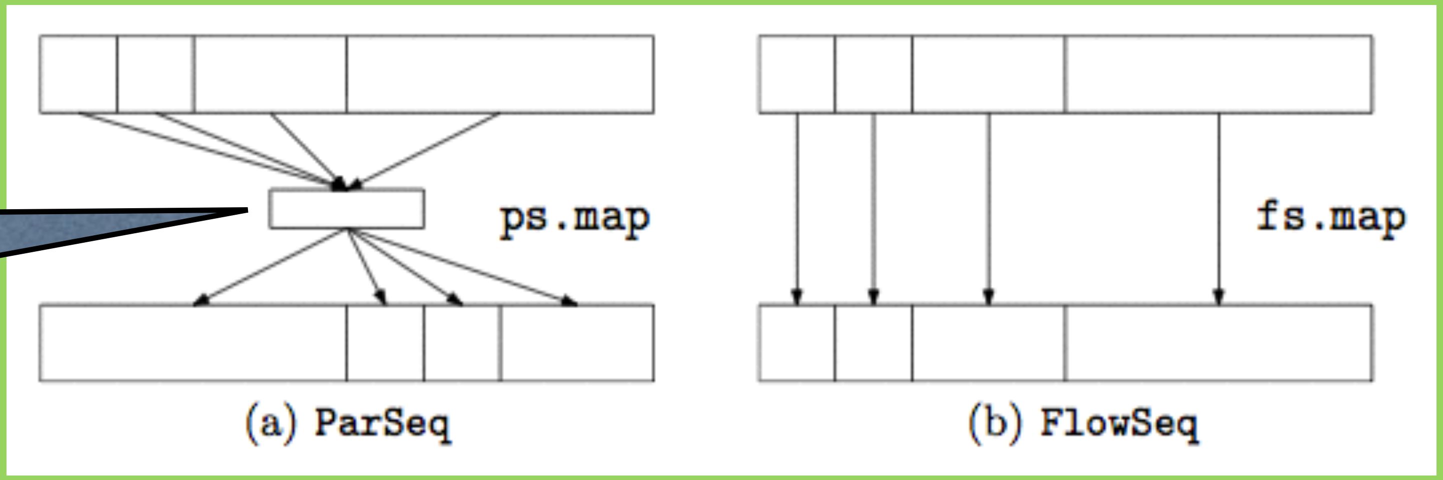
- **Main difference**: no barriers after bulk ops

```
val res: FlowSeq[Int] = myFlowSeq.map(transform)
```

- Call to **`map`** returns immediately, yielding a `FlowSeq` whose elements are well-defined, but not yet computed

FlowSeqs: Barrier Freedom

wait for all blocks



```
val res: FlowSeq[Int] = myFlowSeq.map(transform1)
val final = res.map(transform2)
```

- All calls to *map* return immediately
- As soon as an element/block has been transformed using *transform1*, it flows to the next "processing step", *transform2*

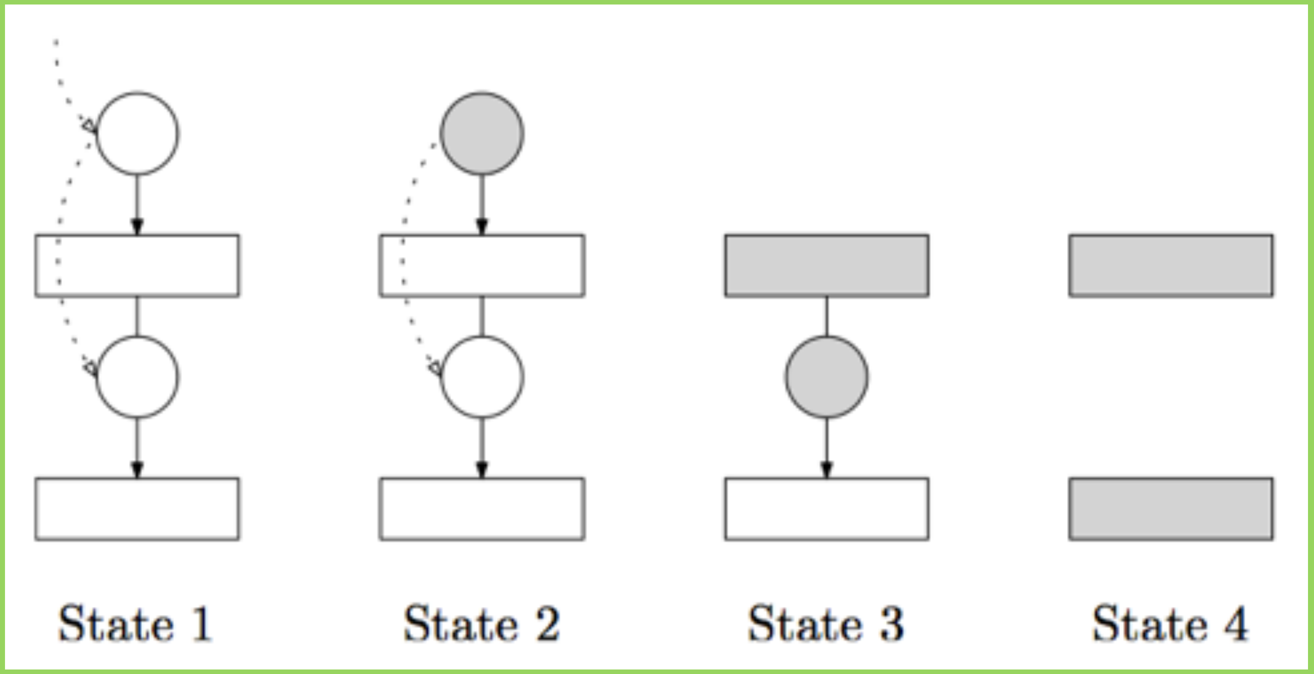
FlowSeqs: Synchronization

- Can insert barriers explicitly
- *blocking* waits until all blocks computed
- Some operations return futures instead

```
val res: FlowSeq[Int] = myFlowSeq.map(transform1)
val final = res.map(transform2)
val nonFlowSeq = final.blocking
```

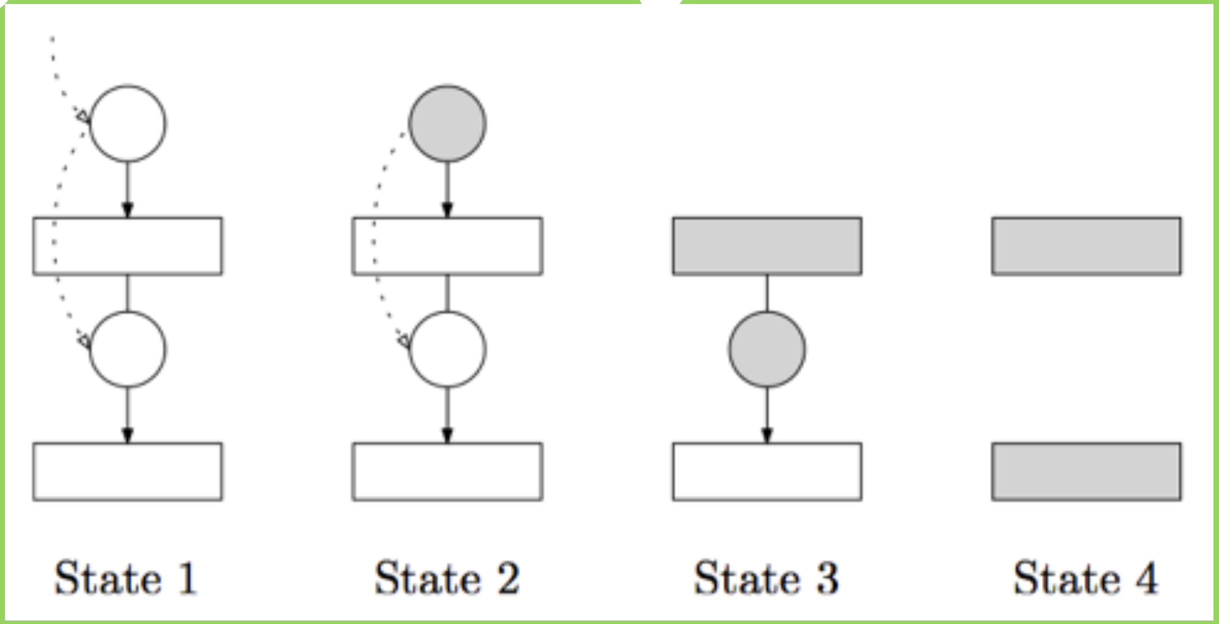
Dependency Tracking

Dependency tracking per block



- Rectangle = block (chunk) of internal data array, computed by single worker thread
- Circles = jobs
  - *gray*: submitted for execution/executing
  - *white*: some required data not yet available

Dependency Tracking



1. Both blocks not yet computed
2. Job for first block scheduled for execution; second job added to first job's dependency queue
3. First block completed; second job scheduled for execution
4. Both blocks completed

Implementation

- Lock-free implementation in Scala
    - Uses JVM intrinsics like CAS via *sun.misc.Unsafe*
- JDK 7 ForkJoinPool as execution environment
- Micro benchmarks comparing to Scala's parallel collections

Benchmarks

Scalar product

```
val x = FlowSeq.tabulate(size)(x => x*x)
val y = FlowSeq.tabulate(size)(x => x*x)

(x zip y).map(x => x._1 * x._2).fold(0)(_ + _).blocking // OR

(x zipMap y)(_ * _).fold(0)(_ + _).blocking // OR

(x zipMapFold y)(_ * _)(0)(_ + _).blocking
```

Function that takes two parameters

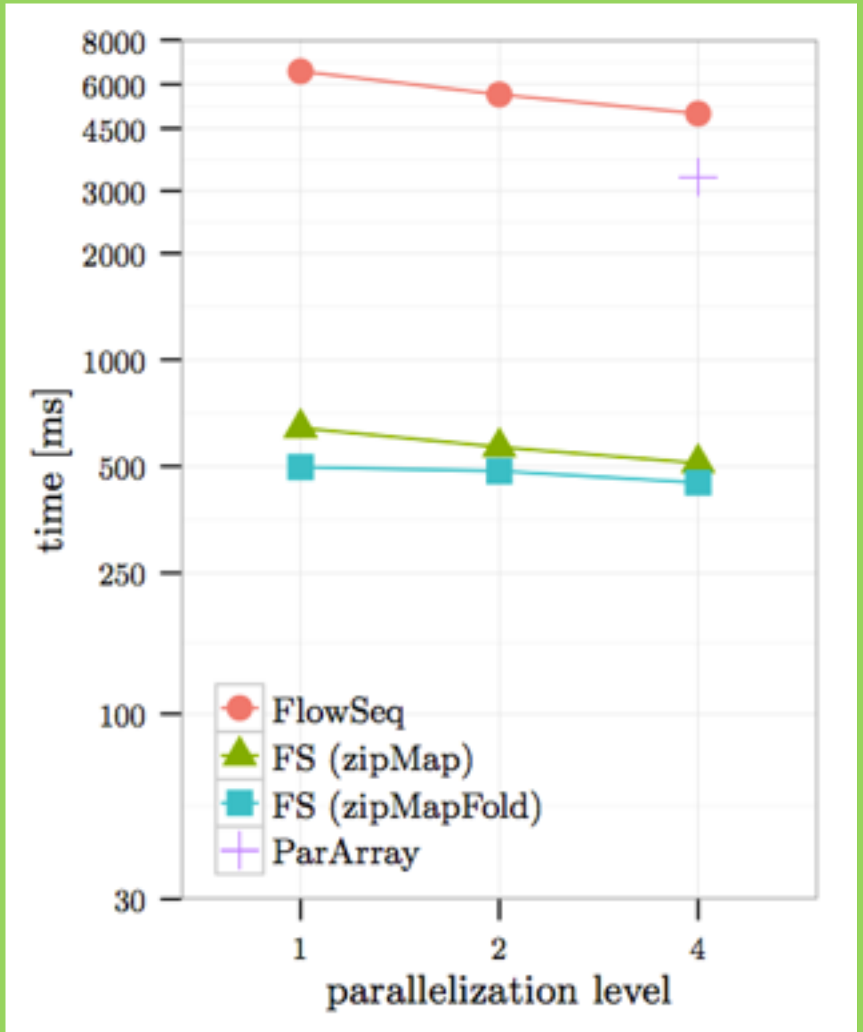
where

Function that takes a tuple as a parameter

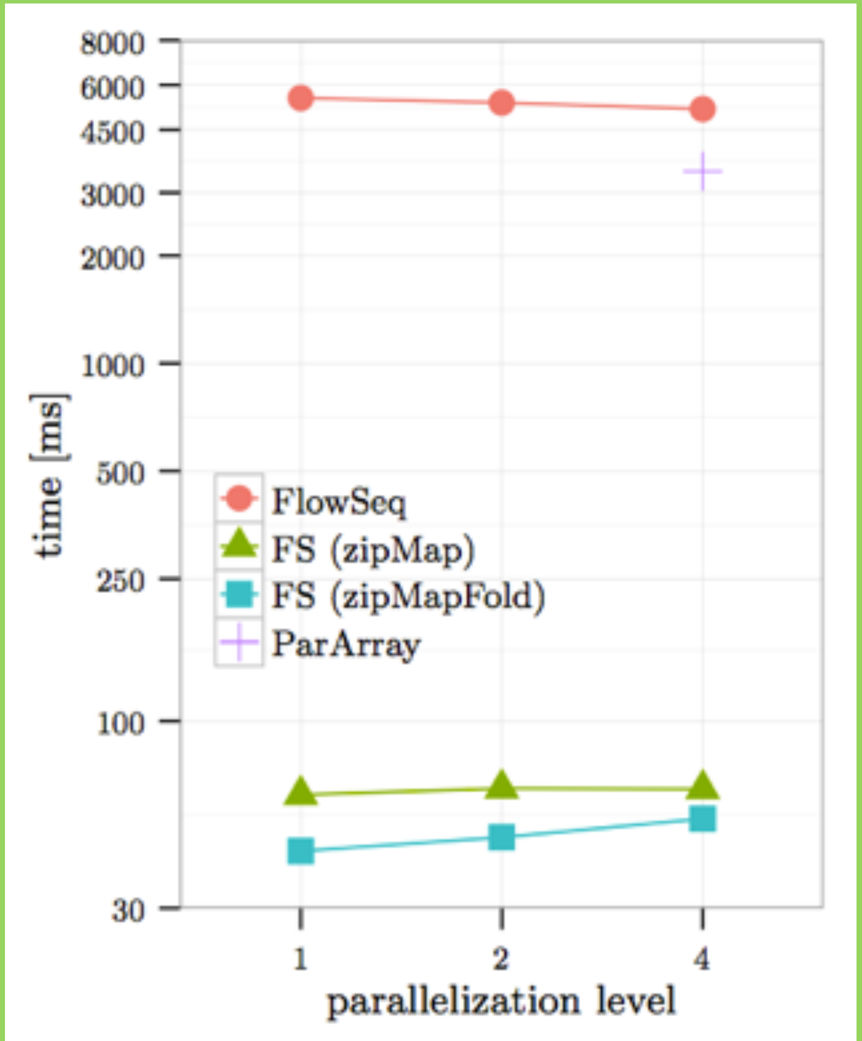
```
x.zipMap(y)(f) <--> x.zip(y).map(f.tupled)
x.zipMapFold(y)(f)(z)(g) <--> x.zip(y).map(f.tupled).fold(z)(g)
```

Benchmark Results

Scalar product (size = 10^7)



(a) Execution Time

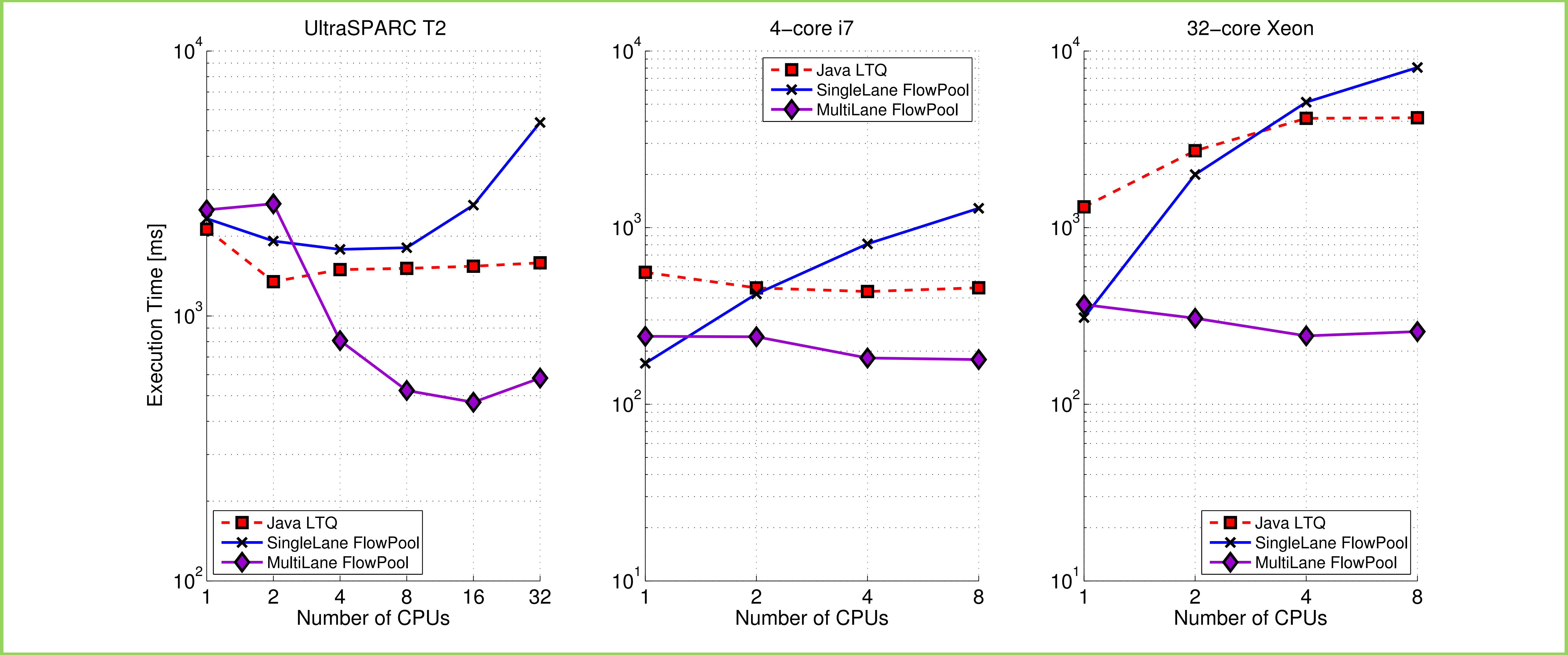


(b) GC Time

Without kernel fusion a majority of time spent in GC!

The Cost of Ordering

- FlowPools: *unordered* FlowSeqs
- Benchmark: create and map



Experience

FlowPools and FlowSeqs have some things in common with JDK 8's streams (package *java.util.stream*)

- Give up some amount of determinism
- To reduce object creations and GC overhead, Java streams are not data structures, but only *views* that process elements on demand
- Computation only kicked off when a *terminal operation*, such as *sum* or *reduce*, is called

Applying FlowSeqs

- FlowSeqs are useful in the context of another dataflow-esque model: Rx (Reactive Extensions)
- What is Rx?
- Programming model based on *observable data streams*, such as event sources
- Only minimal requirements on host language
    - There are implementations for most mainstream programming languages

Why Rx?

- Principled approach to composing observable data streams
- A very general model for *push-based, high-volume data streams*
- Language-agnostic
- Many industrial applications

Rx Basics

```scala
trait Observable[T] {
  def subscribe(observer: Observer[T]): Disposable
}

trait Observer[T] {
  def onNext(value: T): Unit
  def onError(error: Exception): Unit
  def onCompleted(): Unit
}

trait Disposable {
  def dispose(): Unit
}
```

Rx: Behavioral Assumptions

- Calls to an instance of *Observer[T]* should follow the regular expression onNext(t)* (onCompleted() | onError(e))?
- Implementations of *Observer[T]* can be assumed to be synchronized; conceptually they run under a lock
- Resources associated with an observer should be cleaned up when *onError* or *onCompleted* is called. In particular, the subscription returned by the *subscribe* call of the observer will be disposed of by the observable as soon as the stream completes.

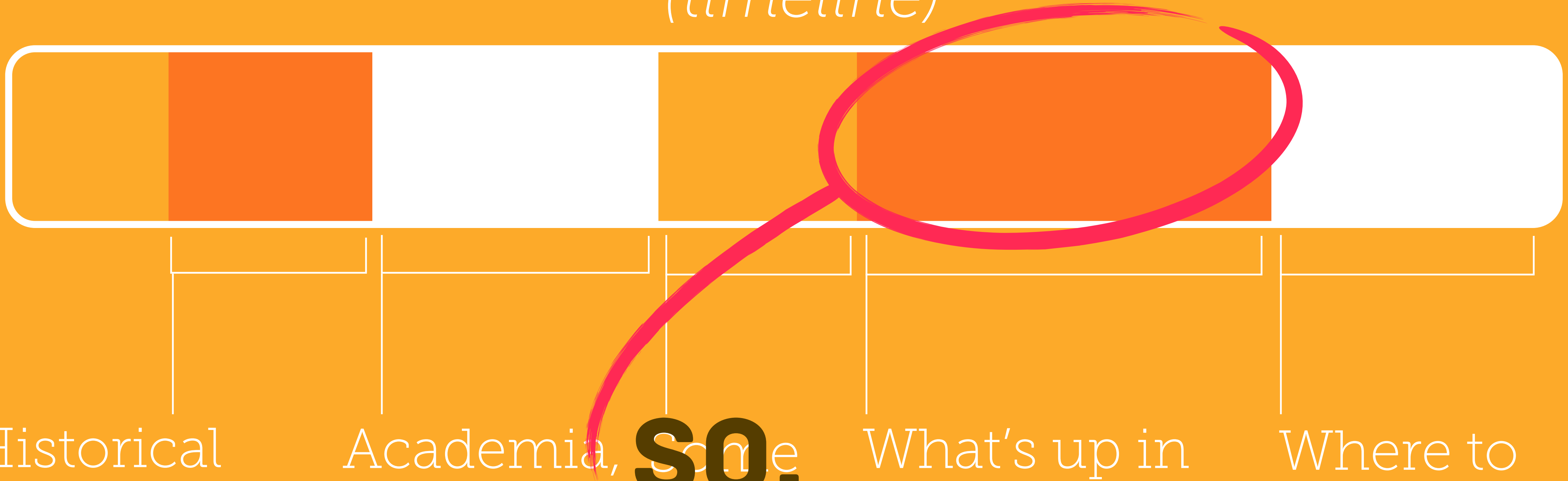
Implementing Observables

- Now we have the interfaces
- Meijer describes a number of combinators to compose observables
- Remaining challenge: efficient implementations of data processing steps
- This is where FlowSeqs come in!

Observable FlowSeqs

- Ongoing work
- Goal: Efficient parallel stream processing integrated with Rx model
- Idea: Turn FlowSeqs into Observables
  - Seal corresponds to completing a stream
- Required machinery already in place, but so far only internal to FlowSeq implementation
- Combinators on the obtained streams can be implemented using *FlowSeq*'s combinators

This talk *(timeline)*



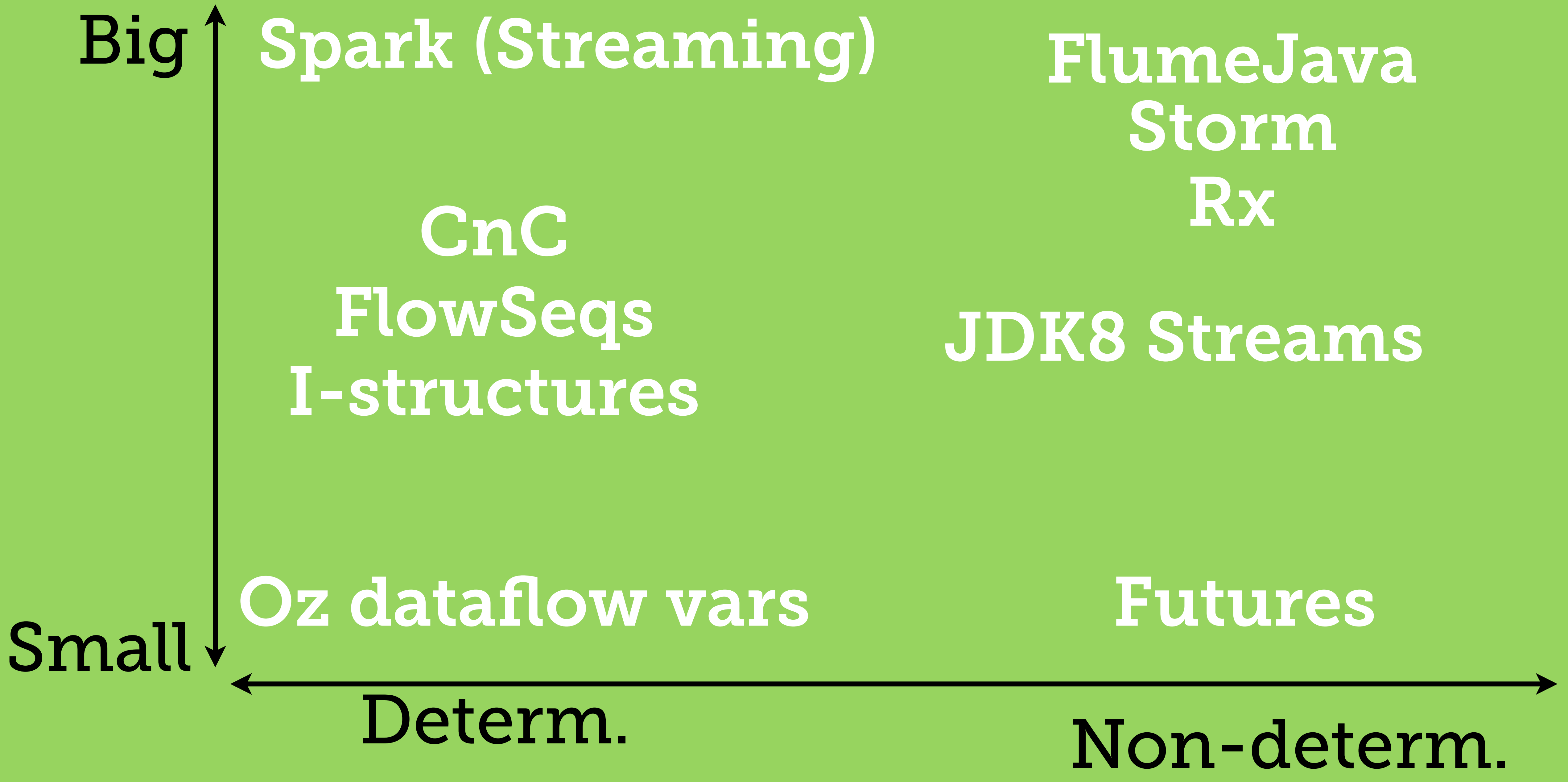
SO,

What's industry's take on all of this?

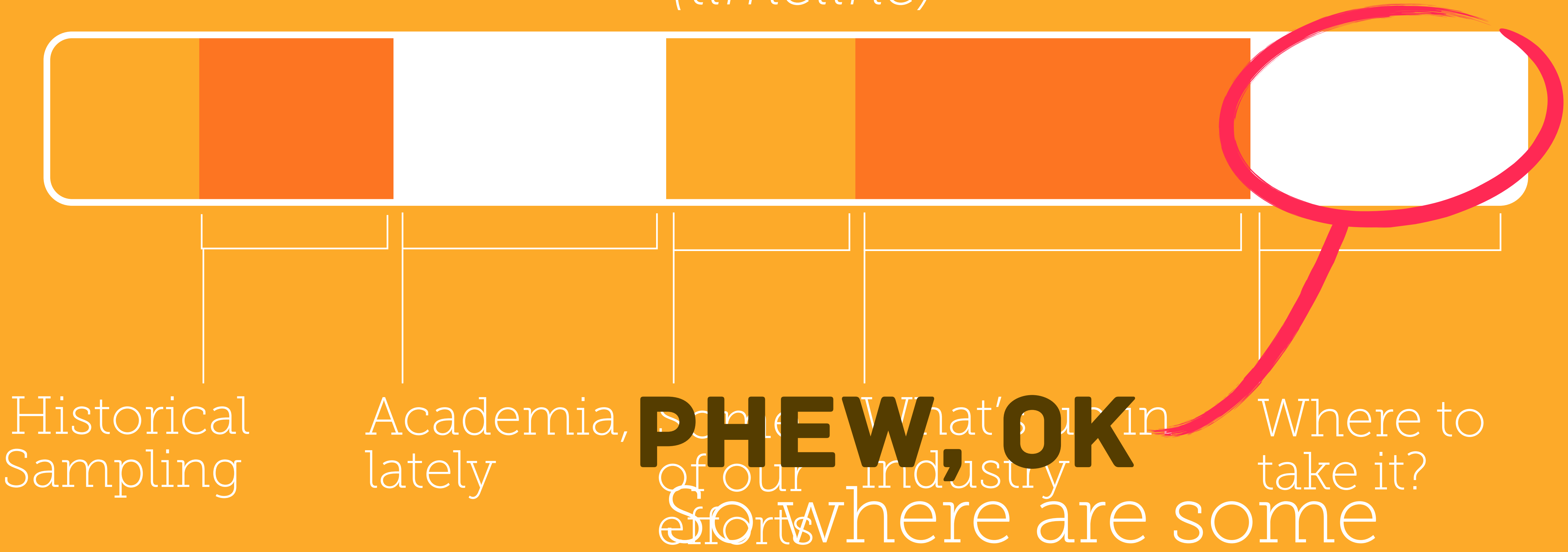
What's Hot in Industry?

- Typically dataflow properties relaxed
- Library implementations
- Try to incorporate ideas into mainstream runtime systems
- Lots of libraries and frameworks that are similar to dataflow programming models
  - Rx, JDK 8 Streams, FlumeJava, Futures/Promises, Storm, Spark, ...

Map


Big
Small
Determ.
Non-determ.
Spark (Streaming)
CnC
FlowSeqs
I-structures
Oz dataflow vars
FlumeJava
Storm
Rx
JDK8 Streams
Futures

This talk *(timeline)*



PHEW, OK

So where are some places we can take Dataflow?

OPEN QUESTIONS:

- How (much) should this map inform research directions?
- Is transitioning from small to big data important?
- Should a system provide controlled non-determinism?

WHY IS DATA FLOW SUCH A POPULAR IDEA IN DISTRIBUTED SYSTEMS?

WHY DON'T WE SEE THE SAME THING HAPPENING IN MULTICORE?

OPEN QUESTIONS:

- Are correct-by-construction programs feasible in a library-based approach to dataflow?
- What kind of static checking is most useful for dataflow programs?
  - Which types? Which effects?
- Which other programming models would be interesting to integrate with dataflow?

QUESTIONS?

Dataflow vs. Stream Processing

Sacrifice flexibility to enable more optimizations

- Stream Processing:
  - Works well for DSP or GPU-type applications (image, video, and digital signal processing)
  - Regular and repeating computations (stream graph often static): task, data, and pipeline parallelism
- Example: StreamIt's optimizations
  - Coarsen: fuse stateless sections of the graph
  - Data parallelize: parallelize stateless filters
  - Software pipeline: parallelize stateful filters

Dataflow vs. Stream Processing

- Dataflow:
  - Flow graph typically dynamically created/changed
  - Flow graph often implicit (example: Oz)
  - Also used for symbolic computations, stream processing focuses on number crunching, filters, etc., instead
- Challenges:
  - Optimization (hybrid static/dynamic?)
  - Language vs. library