The WaveScope Project

Functional Programming in the Wild

Ryan Newton
Greg Morrisett
Sam Madden

MIT/CSAIL

Also with:
Lewis Girod, Mike Allen, Kyle Jamieson, Yuan Mei,
Stanislav Rost, Arvind Thiagarajan,
Hari Balakrishnan

http://wavescope.csail.mit.edu/

Applications:
Stream + Signal Processing

Applications:
Stream + Signal Processing

- Pipeline leak detection and localization

Are there anomalies in the frequency response to an introduced pulse?

http://wavescope.csail.mit.edu/

Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG

Are there anomalies in the frequency response to an introduced pulse?

Is a seizure imminent given signals from various brain regions?

http://wavescope.csail.mit.edu/

Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG
- *In situ* animal behavior studies

Are there anomalies in the frequency response to an introduced pulse?

Is a seizure imminent given signals from various brain regions?

What time ranges contained marmot calls?

http://wavescope.csail.mit.edu/

Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG
- *In situ* animal behavior studies

Are there anomalies in the frequency response to an introduced pulse?

Is a seizure imminent given signals from various brain regions?

What time ranges contained marmot calls?

http://wavescope.csail.mit.edu/

Application Features

Application Features

High data-rate

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes

Application Features

High data-rate

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes

Embedded, low-power devices

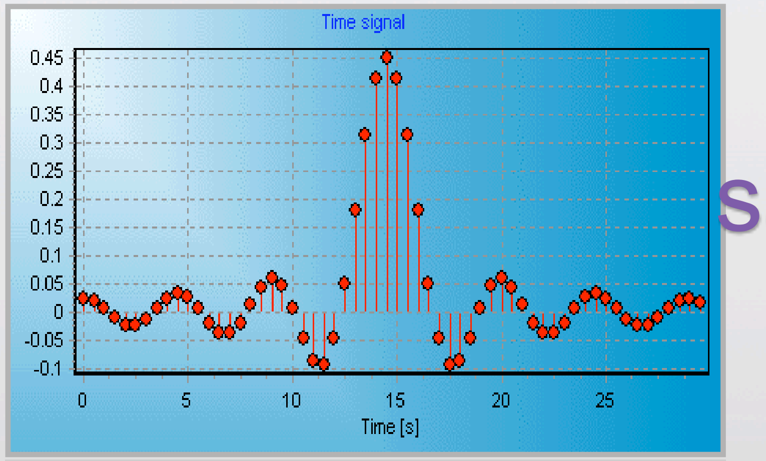
Application Features

High data-rate

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes

Embedded, low-power devices



Regularly sampled signals

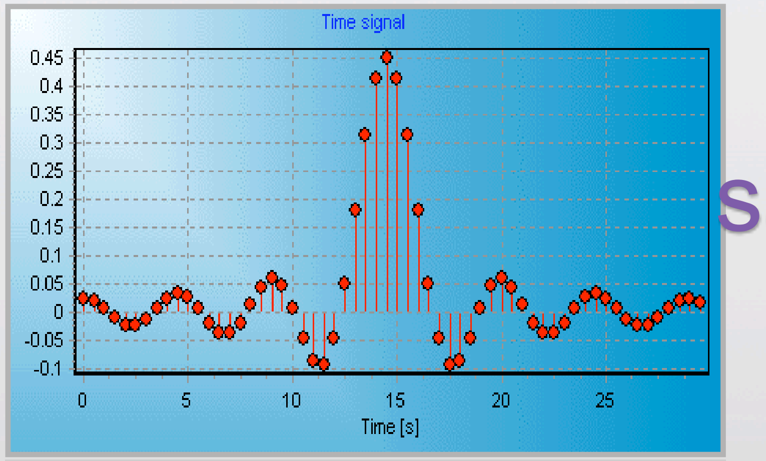
Application Features

High data-rate

Embedded, low-power devices

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes



Regularly sampled signals

- Consistent data rates
- Efficient time-stamping

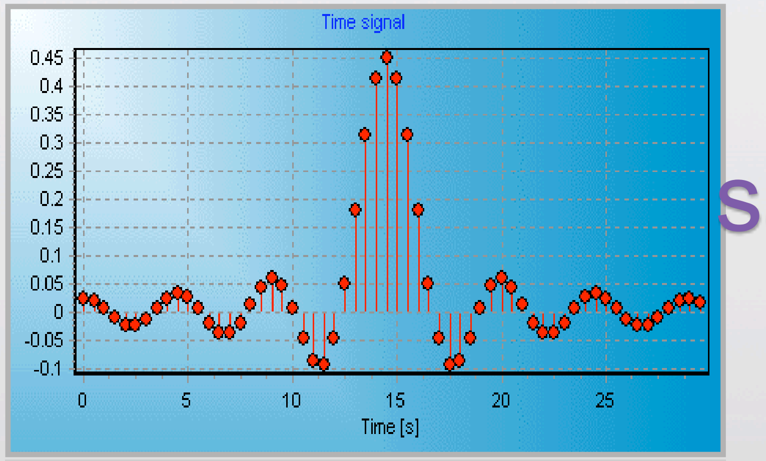
Application Features

High data-rate

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes

Embedded, low-power devices



Regularly sampled signals

- Consistent data rates
- Efficient time-stamping

```
animalcalls = detector(audio)
s2 = map( classify, marmotcalls )
```

Discrete Events

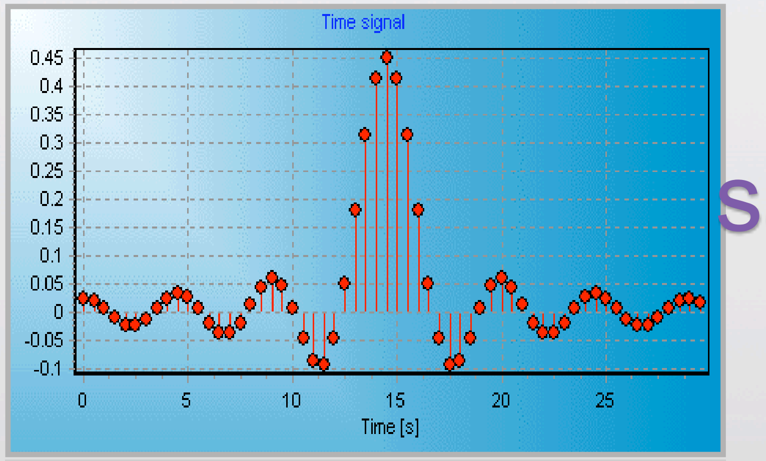
Application Features

High data-rate

Embedded, low-power devices

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes



Regularly sampled signals

- Consistent data rates
- Efficient time-stamping

```
animalcalls = detector(audio)
s2 = map( classify, marmotcalls )
```

Discrete Events

- *First class streams*
- Higher order stream combinators (map, etc)

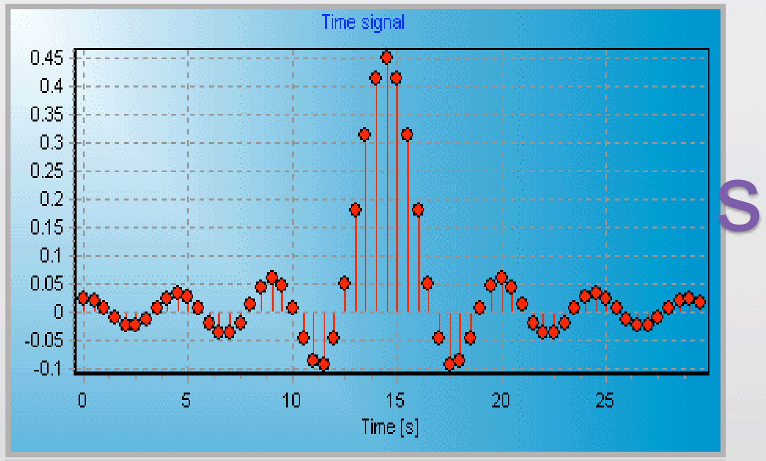
Application Features

Embedded, low-power devices

High data-rate

4 channels
x 48 khz
= 400,000 bytes/sec (per node)

x 10-



Regularly sampled signals

- Consistent data rates
- Efficient time-stamping

```
animalcalls = detector(audio)
s2 = map( classify, marmotcalls )
```

Discrete Events

- *First class streams*
- Higher order stream combinators (map, etc)

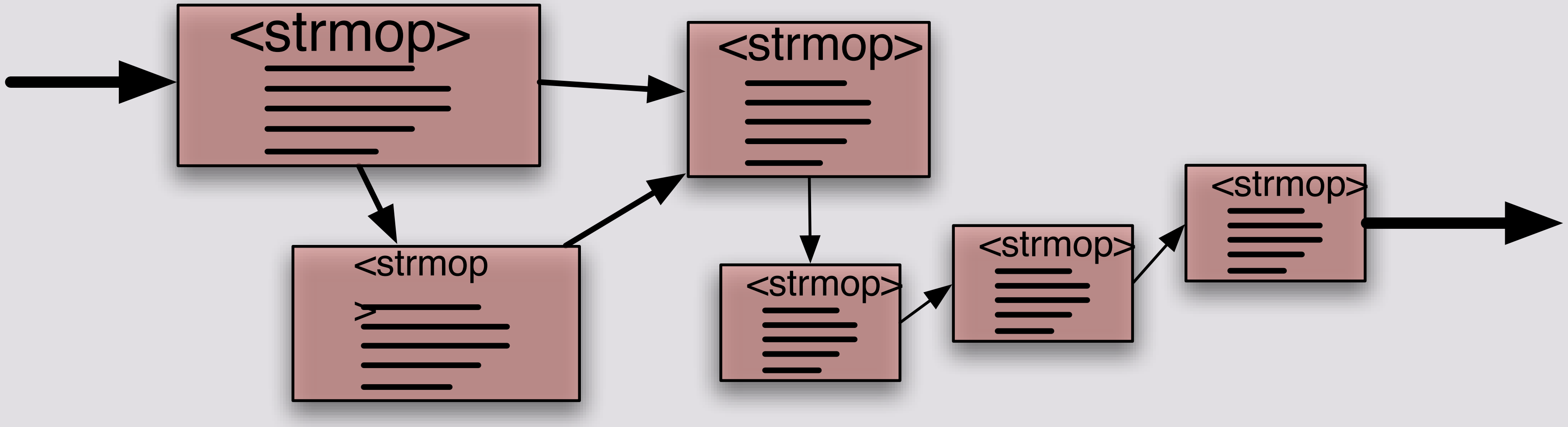
Parallelism at multiple granularities

“WaveScope” - what is it?

- WaveScript Language, Compiler
  - ML (with a few extras - generic printing, Num subkind)
  - Two-stage evaluation (metaprogramming)
  - Run-time model is a graph of stream operators
- XStream engine
  - Links to C backend, other WS backends as well
- Stream/Signal processing libraries
  - Higher order language, polymorphism, metaprogramming enables reuse, new abstractions

Execution Model

- Graph of operators (aka kernels, actors, agents)



- Many possible semantics: synchronous or asynchronous, (non)deterministic operators, shared state, atomicity/execution order, etc
- WaveScript: asynchronous streams of discrete events. Operators may fire whenever an input token is available.
  - Fairness is scheduler's only requirement.
  - NO shared state

A bit of WaveScript code

- One primary stream operator (also merge)

```
iterate x in strm {
  state { cnt = 0 }
  cnt += 1;
  emit g(x,cnt);
  emit f(x,cnt);
}
```

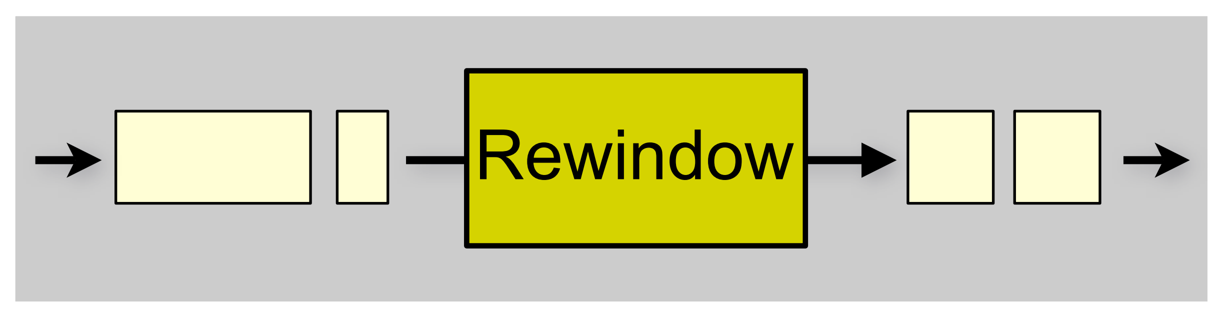
- iterate = Syntactic sugar
  - second class references
- iterate could be a pure combinator,

```
iterate :: ((a,s) -> ([b],s))
              -> Stream a -> Stream b
```

A bit of WaveScript code

- One primary stream operator (also merge)

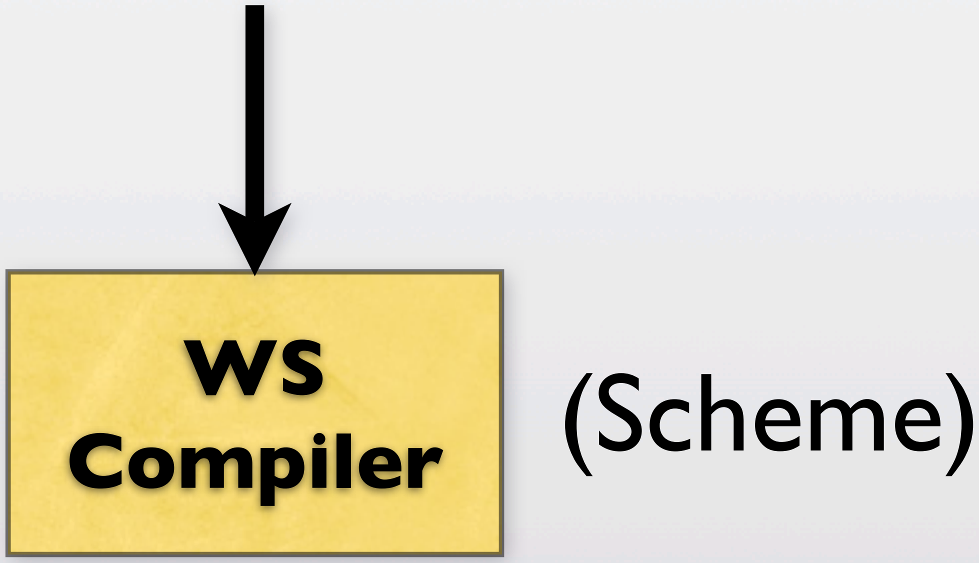
```
iterate x in strm {
   state { cnt = 0 }
   cnt += 1;
   emit g(x,cnt);
   emit f(x,cnt);
}
```



- iterate = Syntactic sugar
  - second class references
- iterate could be a pure combinator,

```
iterate :: ((a,s) -> ([b],s))
           -> Stream a -> Stream b
```

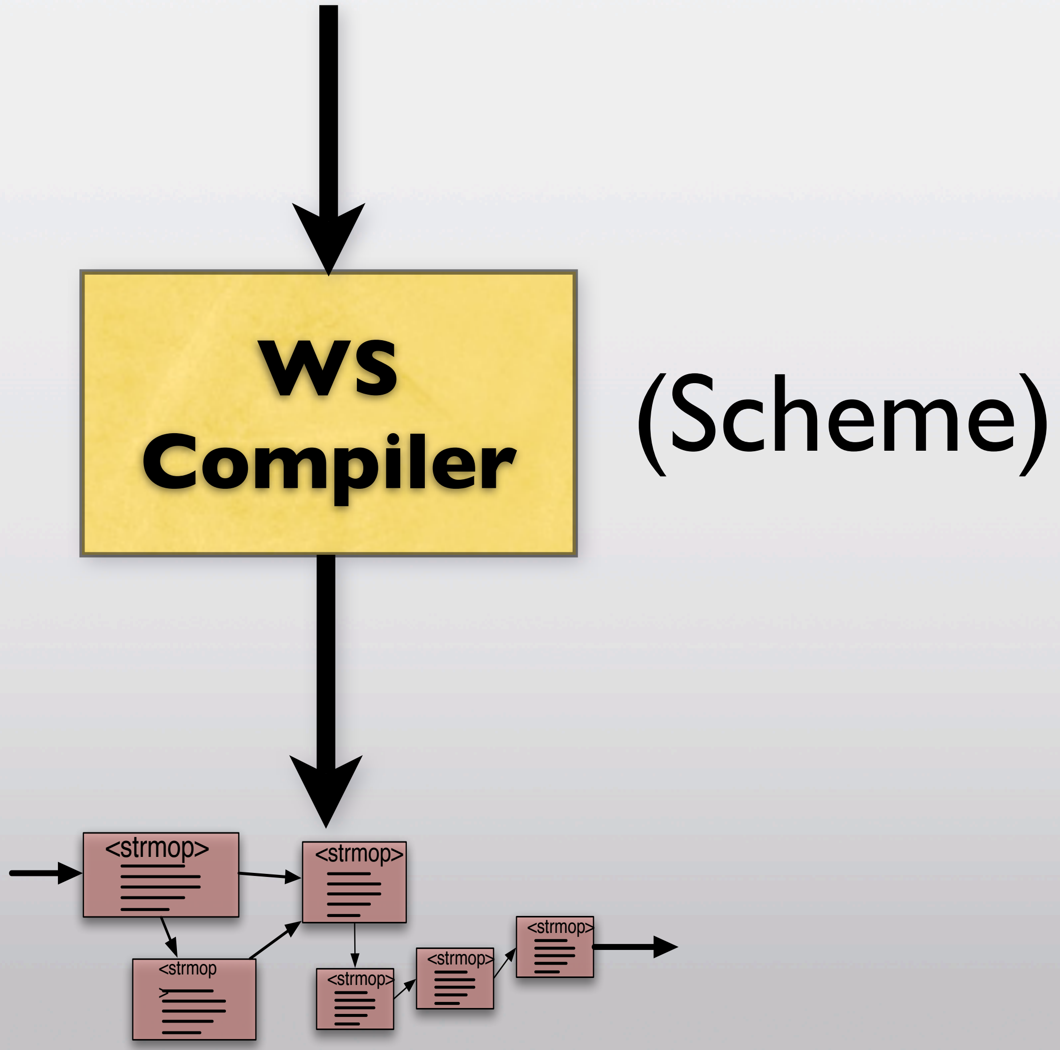
Compilation Workflow



WS Compiler

(Scheme)

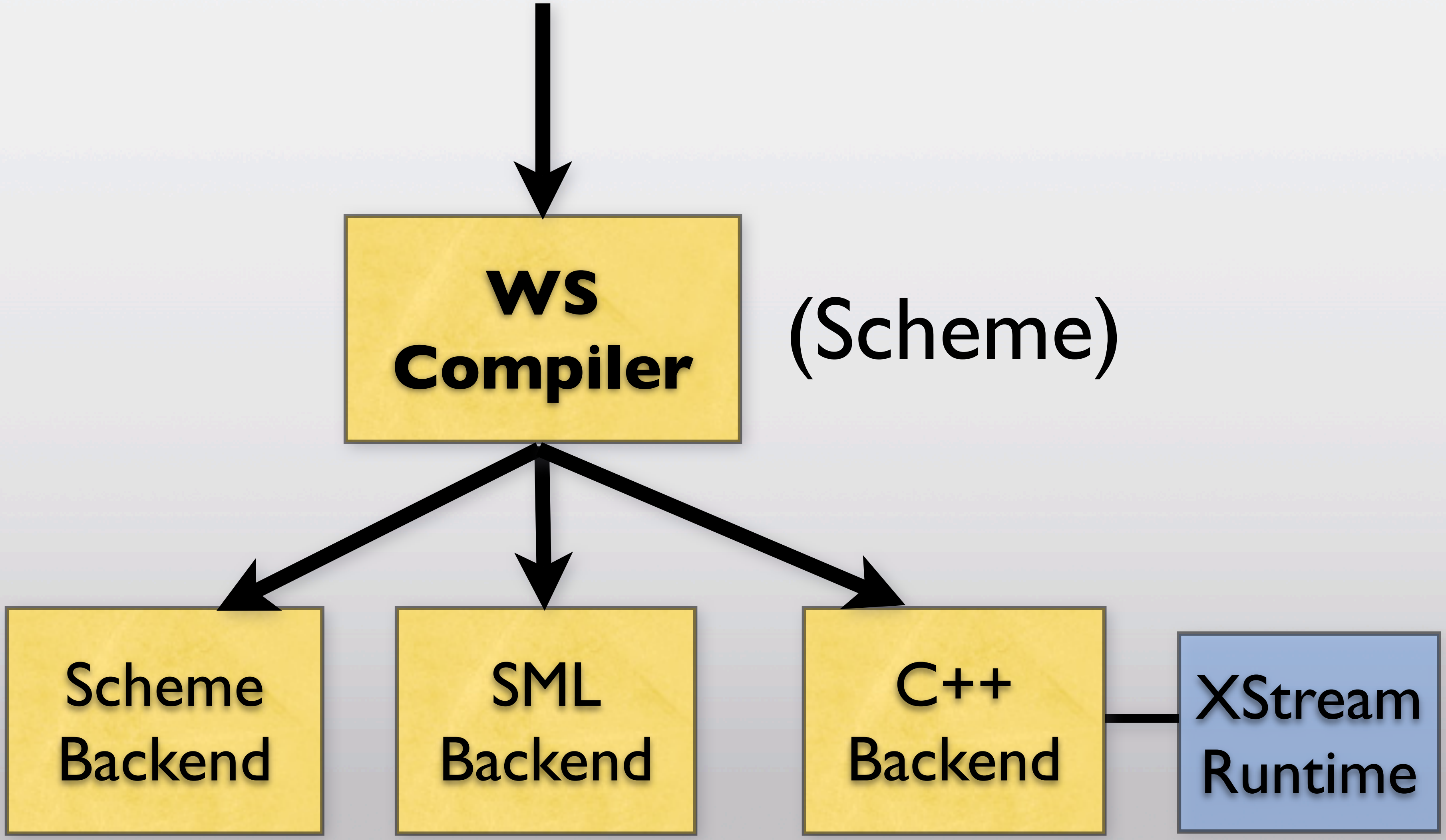
Compilation Workflow



WS Compiler

(Scheme)

Compilation Workflow



WS Compiler

(Scheme)

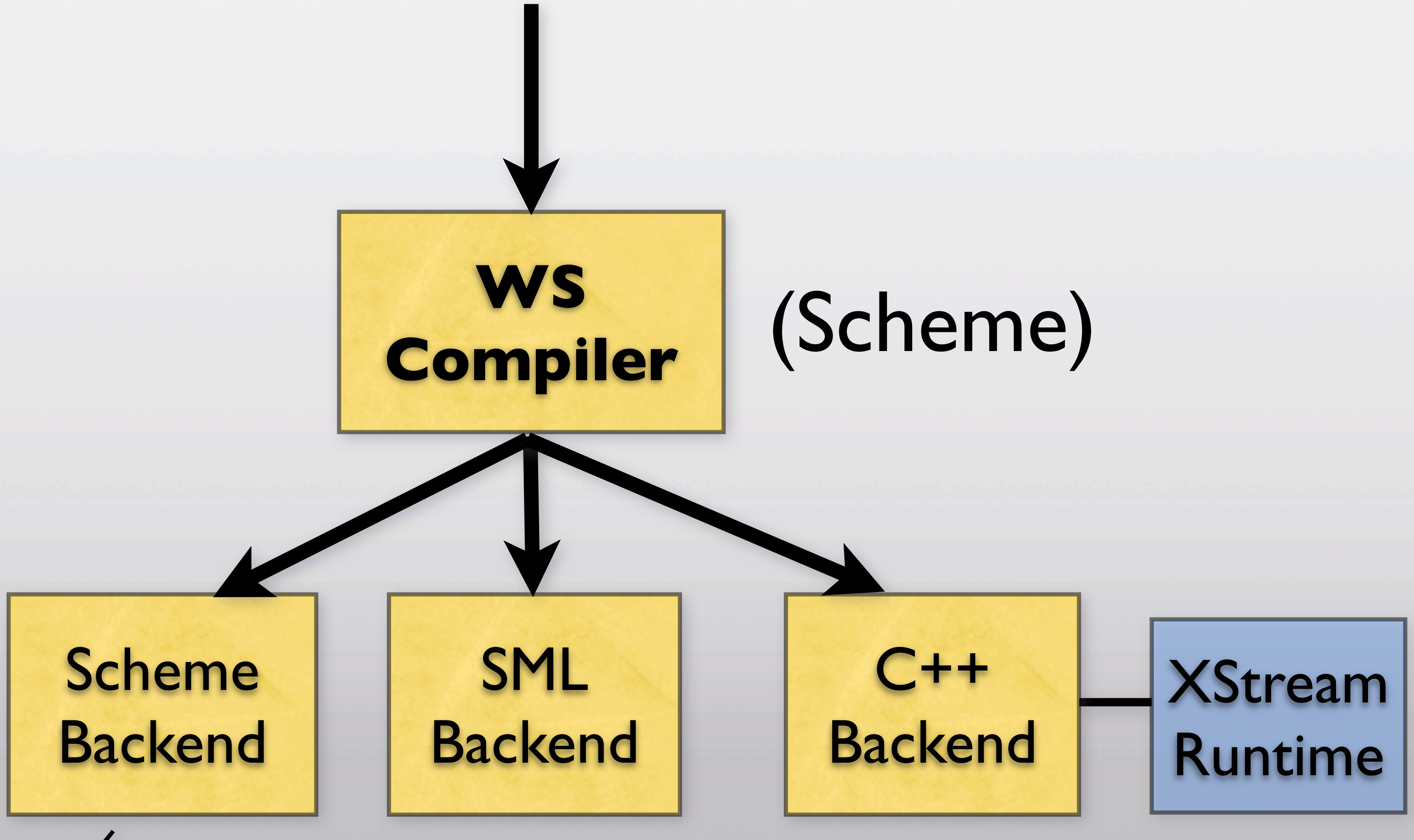
Scheme Backend

SML Backend

C++ Backend

XStream Runtime

Compilation Workflow



WS Compiler

(Scheme)

Scheme Backend

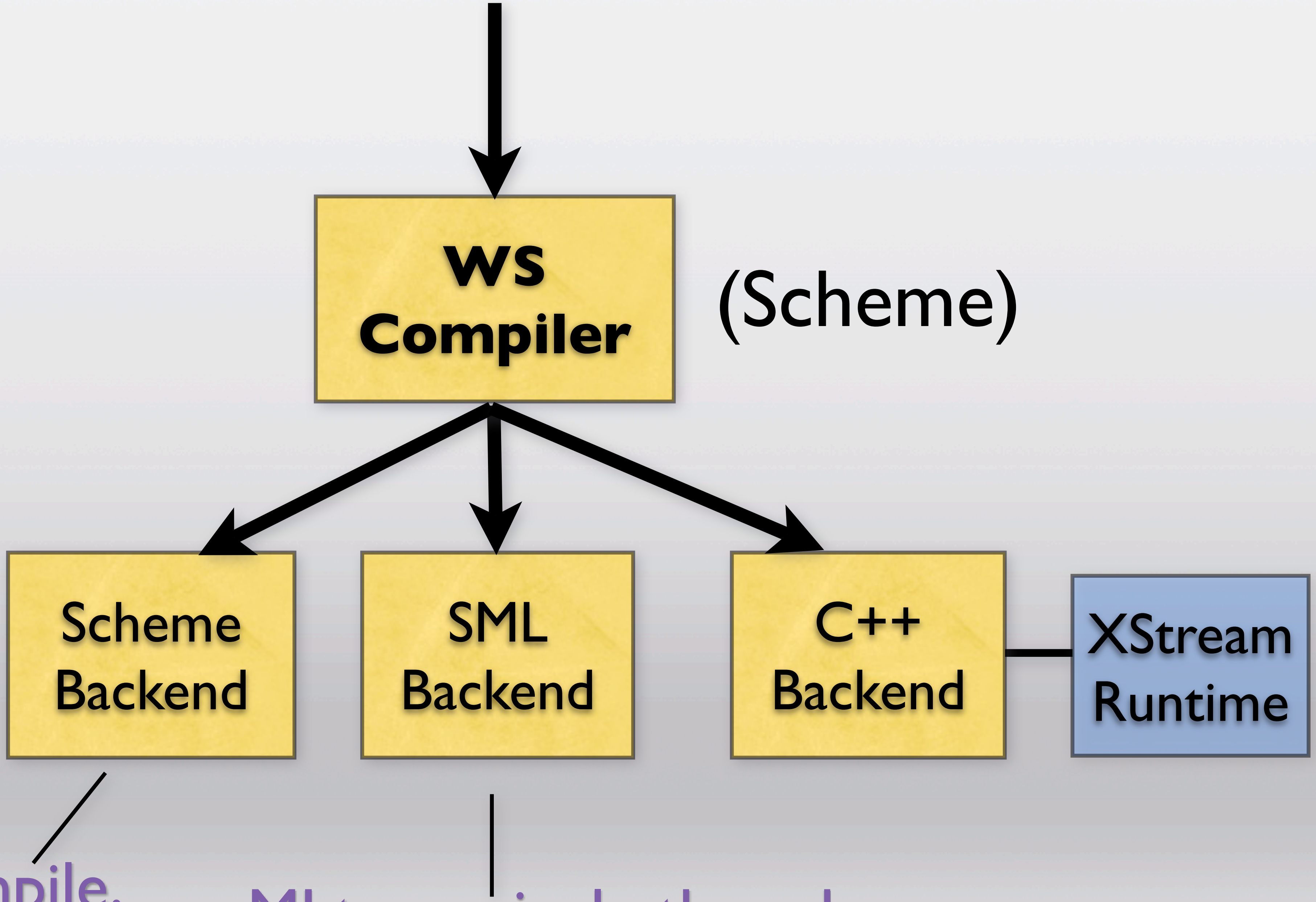
SML Backend

C++ Backend

XStream Runtime

quick compile, interactive queries

Compilation Workflow



WS Compiler

(Scheme)

Scheme Backend

SML Backend

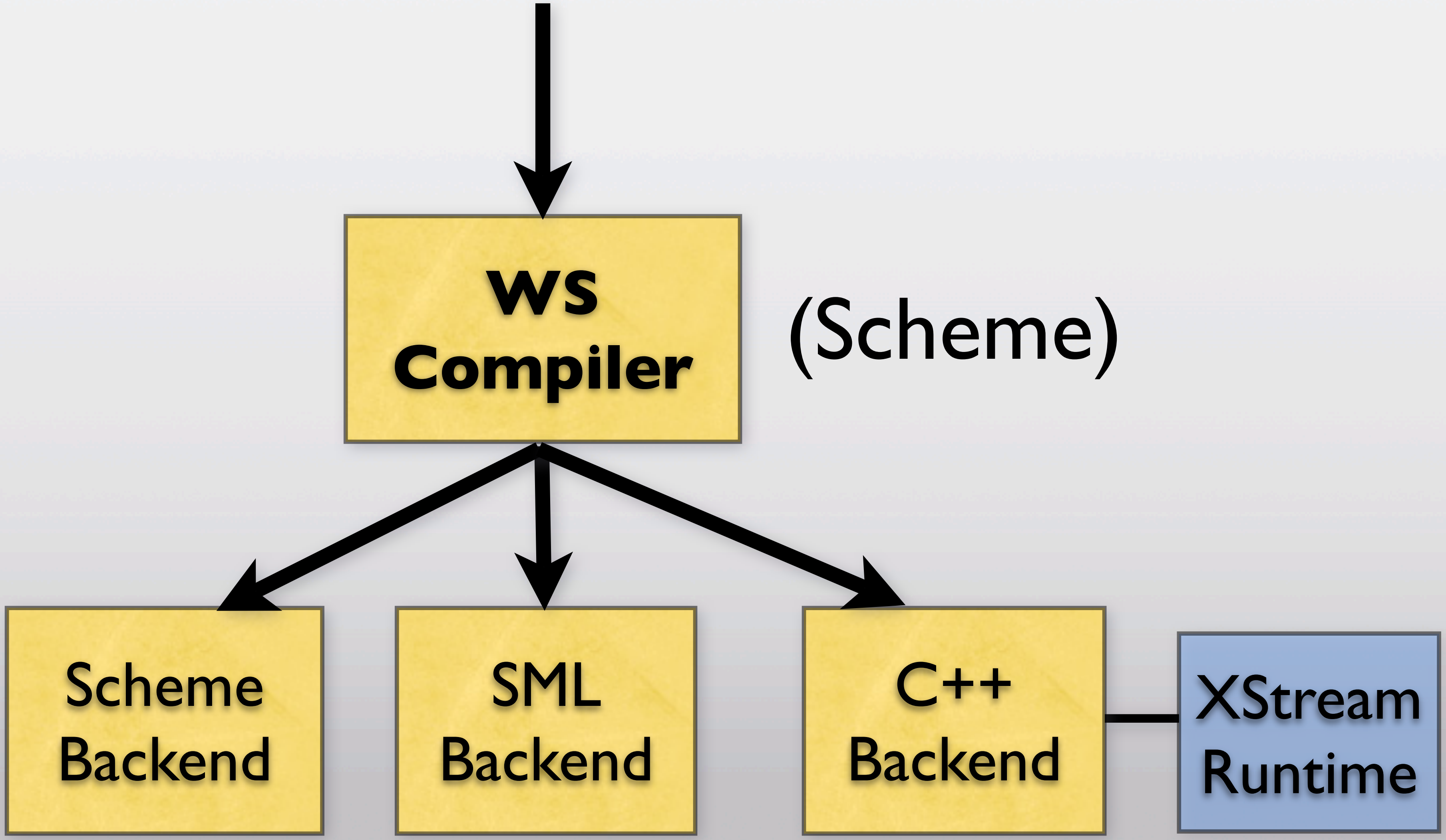
C++ Backend

XStream Runtime

quick compile,
interactive queries

MLton - single thread
copying collector

Compilation Workflow



WS Compiler

(Scheme)

Scheme Backend

SML Backend

C++ Backend

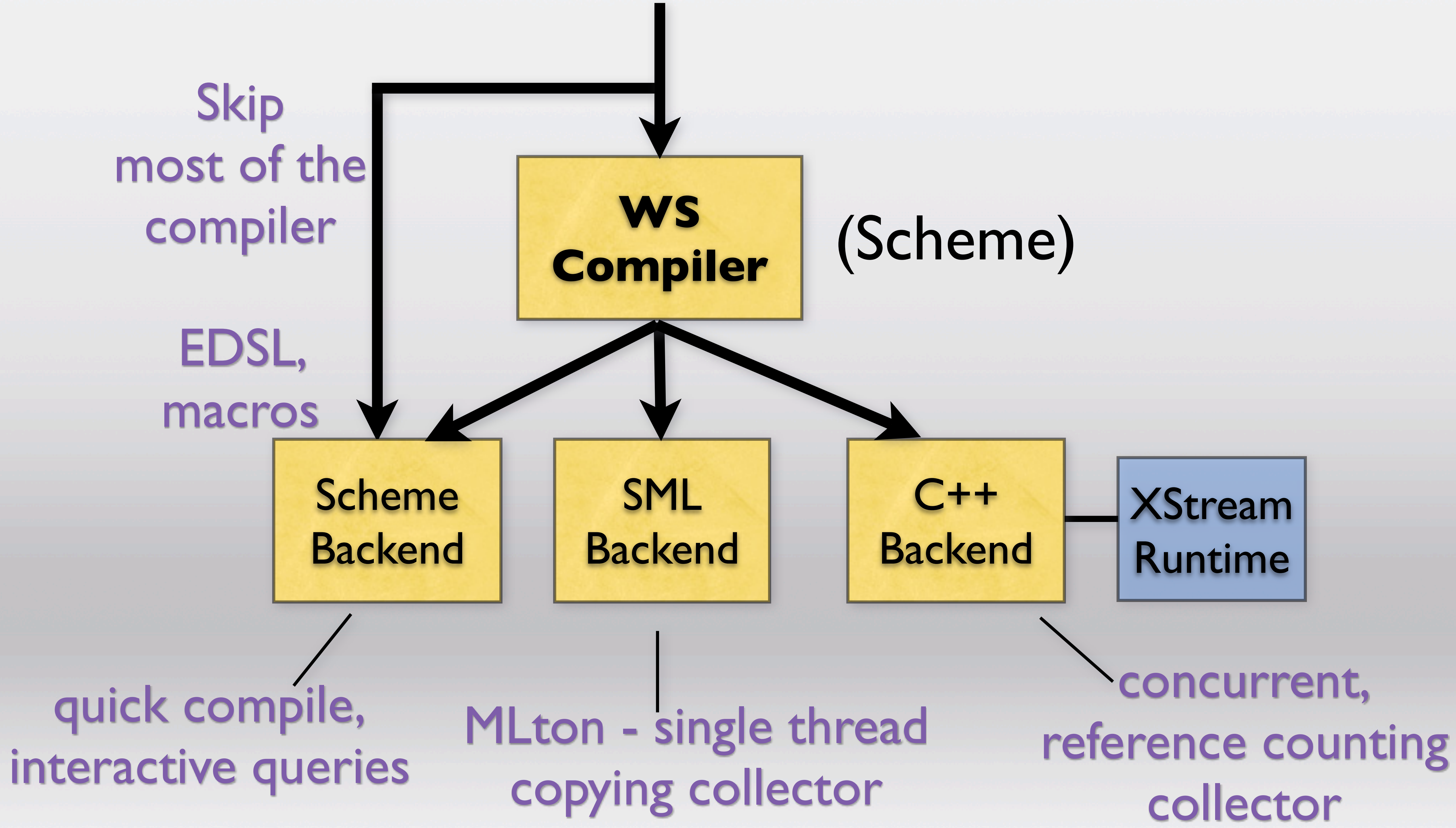
XStream Runtime

quick compile, interactive queries

MLton - single thread copying collector

concurrent, reference counting collector

Compilation Workflow



Skip most of the compiler

EDSL, macros

WS Compiler

(Scheme)

Scheme Backend

SML Backend

C++ Backend

XStream Runtime

quick compile, interactive queries

MLton - single thread copying collector

concurrent, reference counting collector

Why not a general-purpose, higher order language?

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

- Probably not the most efficient code

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

- Probably not the most efficient code
- You’re stuck with the execution environment

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

- Probably not the most efficient code
- You’re stuck with the execution environment
  - Want to develop runtimes for *tiny* platforms

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

- Probably not the most efficient code
- You're stuck with the execution environment
  - Want to develop runtimes for *tiny* platforms
    - Low-overhead memory management

Why not a general-purpose, higher order language?

```
type Stream t = () → (t, Stream t)
```

```
type Sink t = t → ()
type Stream t = Sink t → ()
```

- Probably not the most efficient code
- You’re stuck with the execution environment
  - Want to develop runtimes for *tiny* platforms
    - Low-overhead memory management
  - *Parallel* runtime and garbage collector

Two-Stage Evaluation

Two-Stage Evaluation

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”

Two-Stage Evaluation

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”
- Two meta-program evaluators:
  - Term rewriting (beta, delta reduction)
  - Interpret, marshal, inline until monomorphic

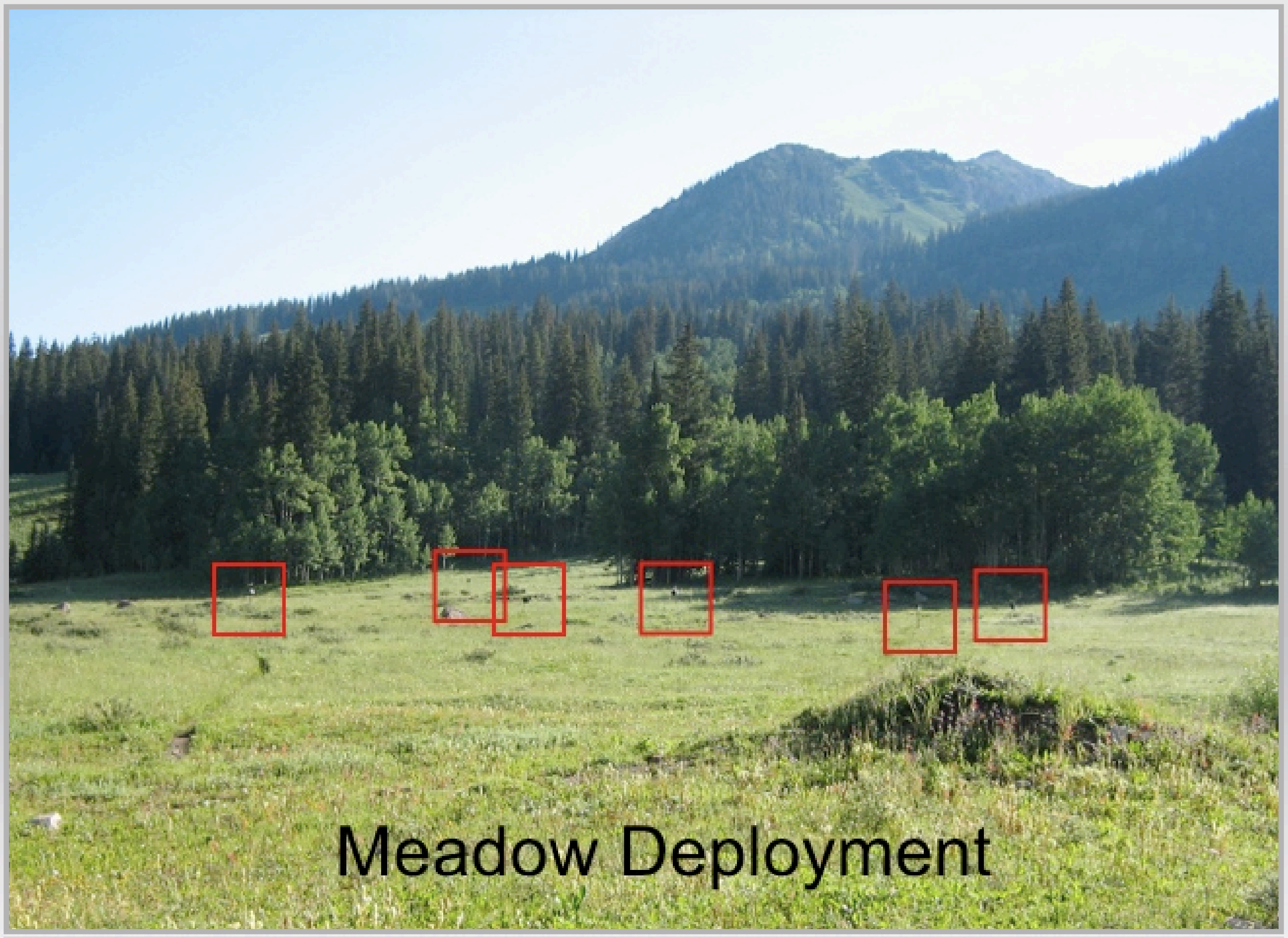
Two-Stage Evaluation

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”
- Two meta-program evaluators:
  - Term rewriting (beta, delta reduction)
  - Interpret, marshal, inline until monomorphic
- Both equivalent to single-stage evaluation
  - With Scheme backend we *allow* single-stage

Two-Stage Evaluation

- Leverage asymmetric metaprogramming
    - Quotation / Anti-quotation free
    - Anything inside an iterate is “object code”
- Two meta-program evaluators:
    - Term rewriting (beta, delta reduction)
    - Interpret, marshal, inline until monomorphic
- Both equivalent to single-stage evaluation
    - With Scheme backend we *allow* single-stage
- Downside: potential code bloat

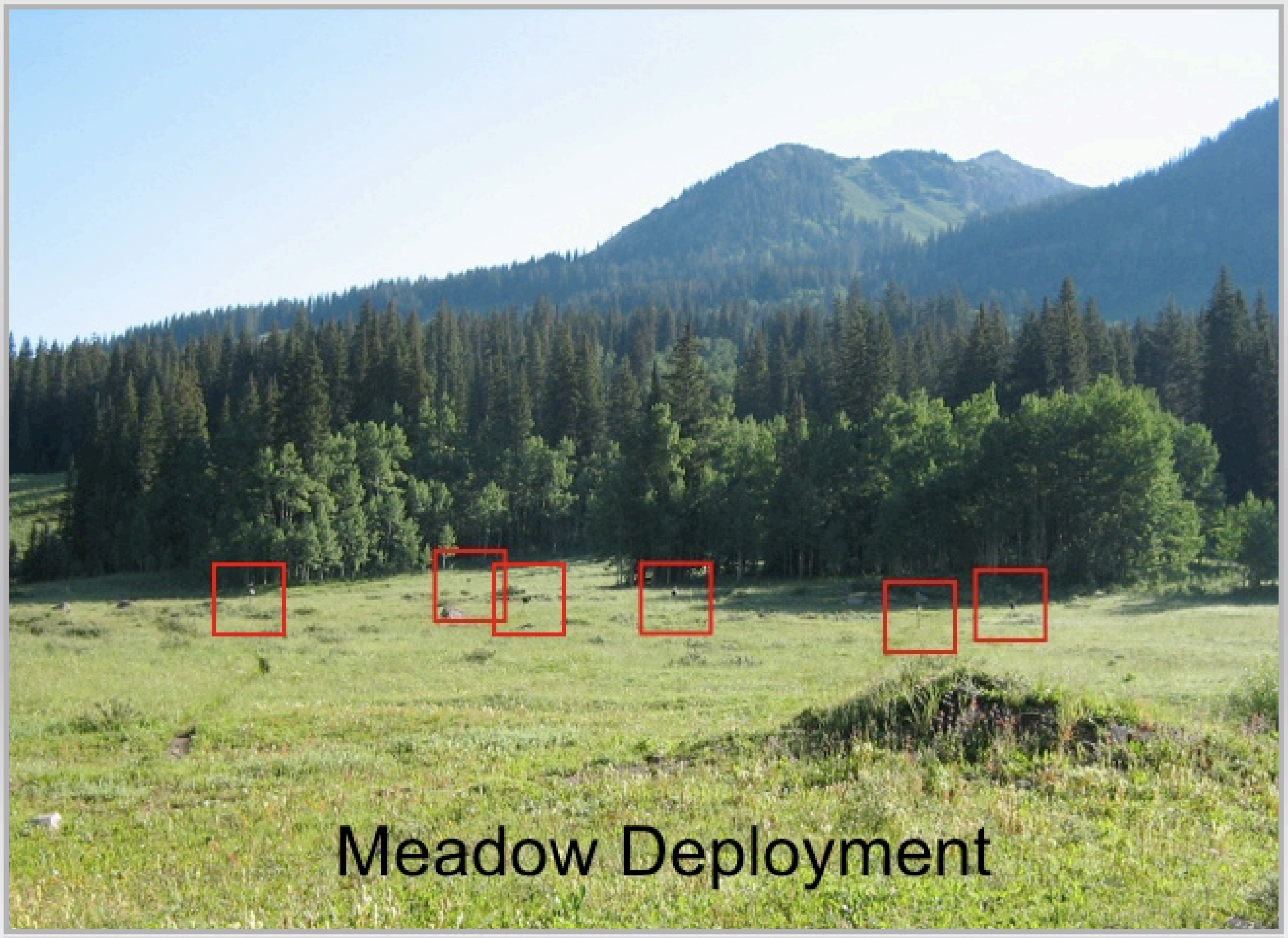
Marmot application



http://wavescope.csail.mit.edu/

Marmot application

- Goal: study calling behavior.

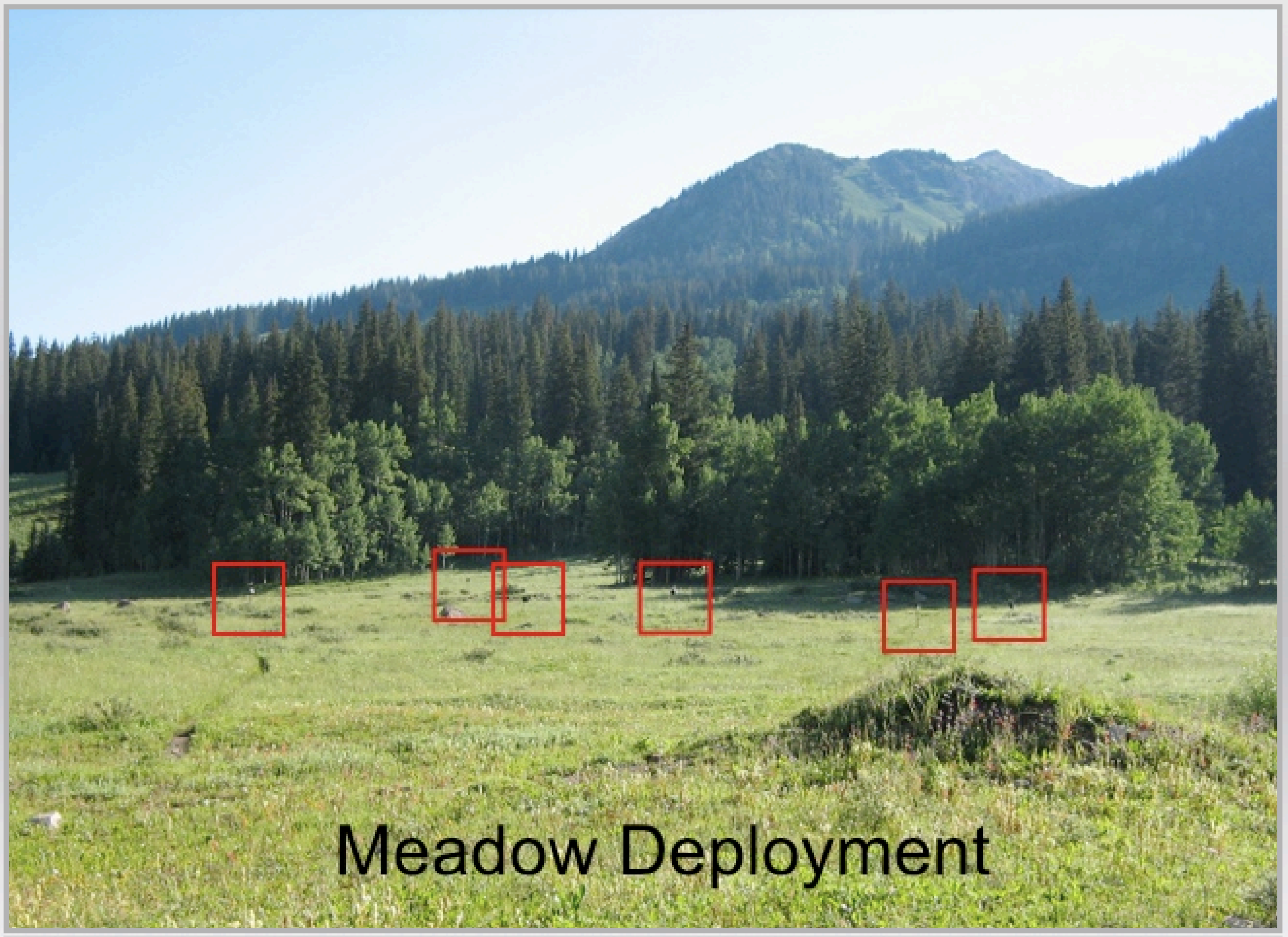


Meadow Deployment

http://wavescope.csail.mit.edu/

Marmot application

- Goal: study calling behavior.



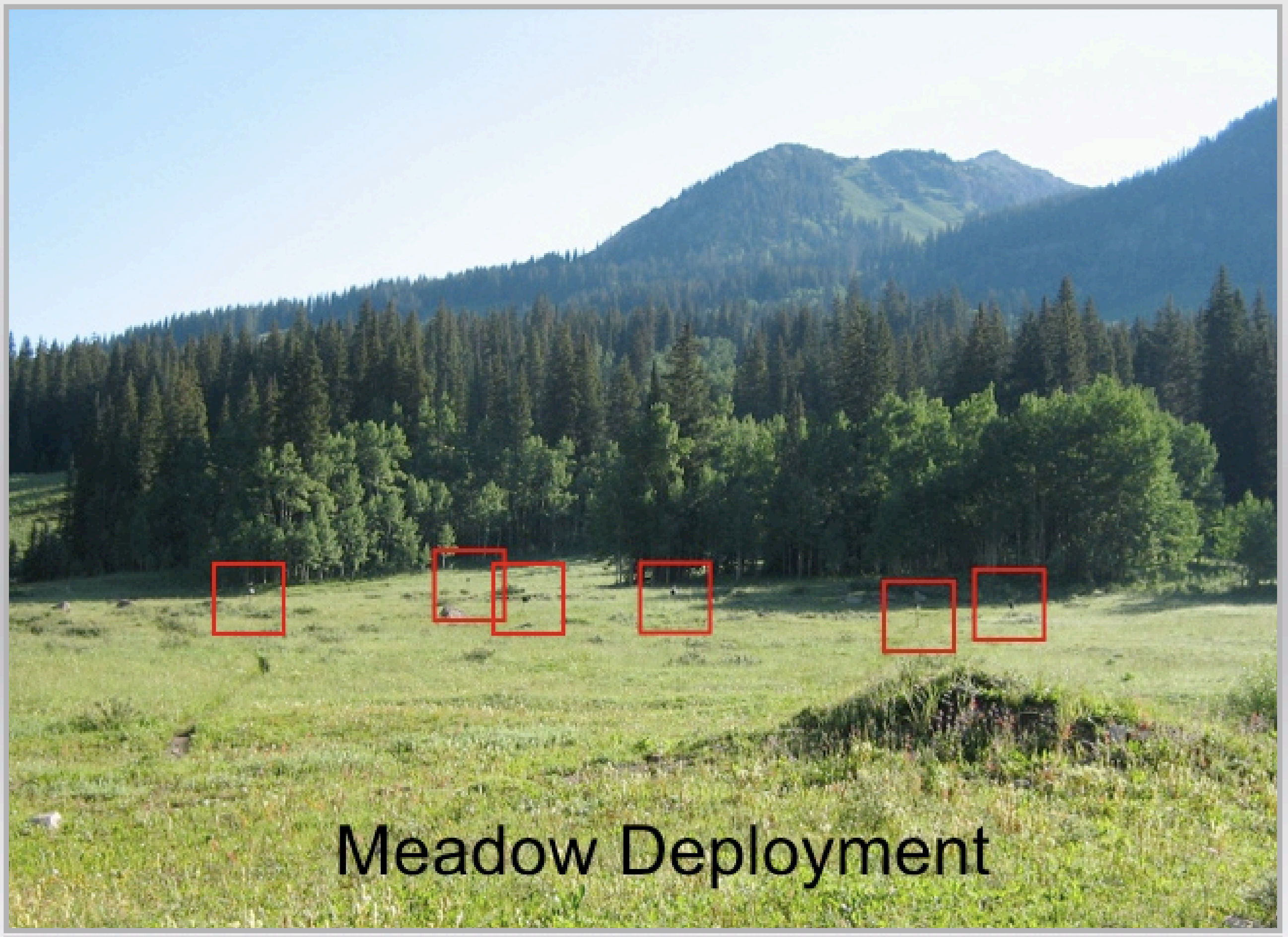
Meadow Deployment

Some Marmots

http://wavescope.csail.mit.edu/

Marmot application

- Goal: study calling behavior.

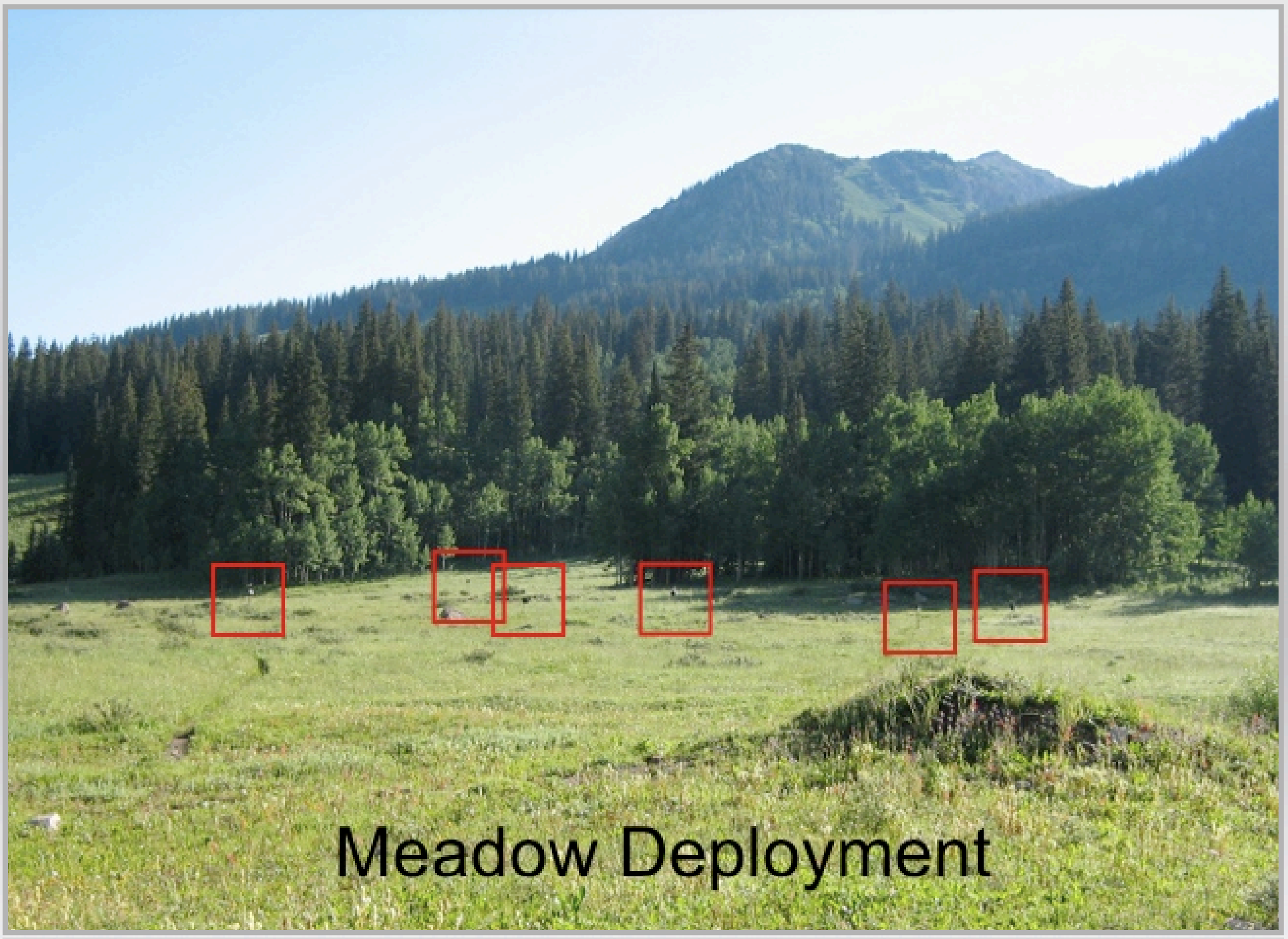


Meadow Deployment

http://wavescope.csail.mit.edu/

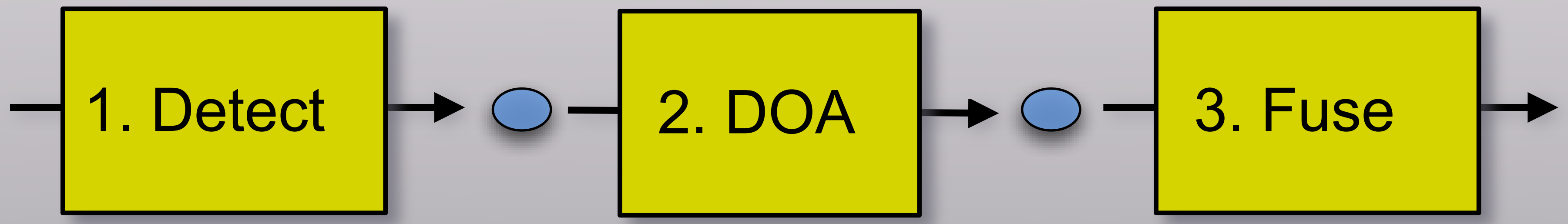
Marmot application

- Goal: study calling behavior.
- Detect, record, localize, classify.



http://wavescope.csail.mit.edu/

Three phases to marmot localization

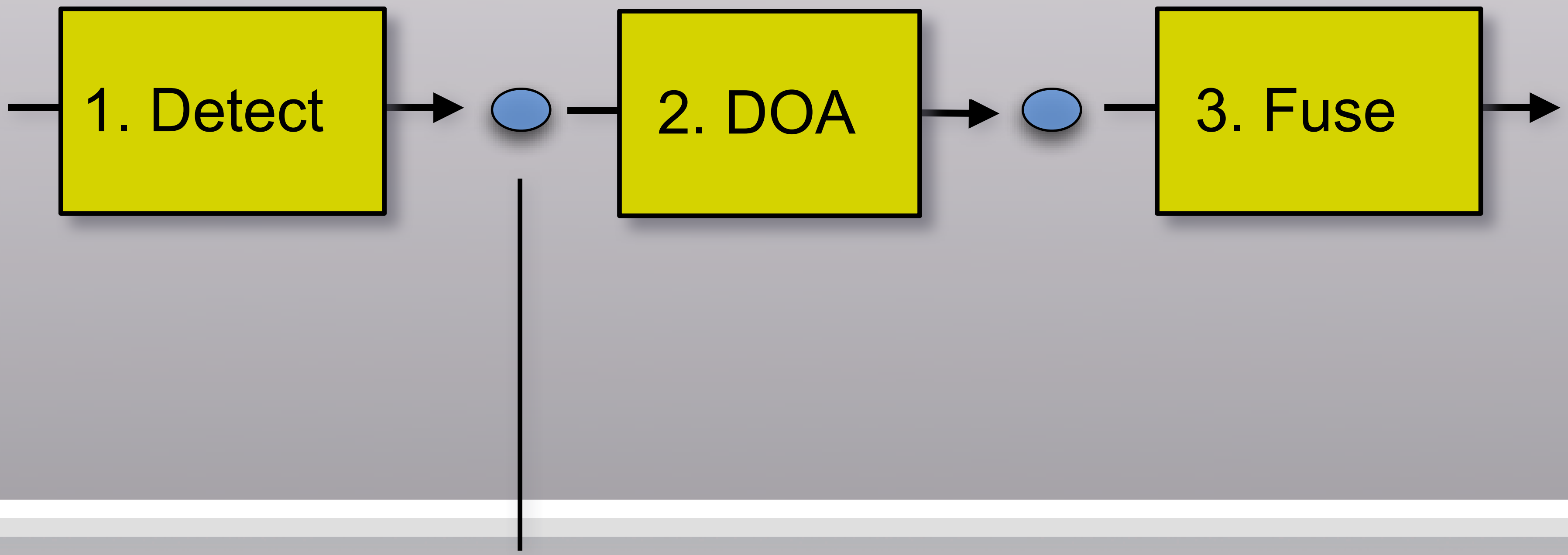


1. Detect

2. DOA

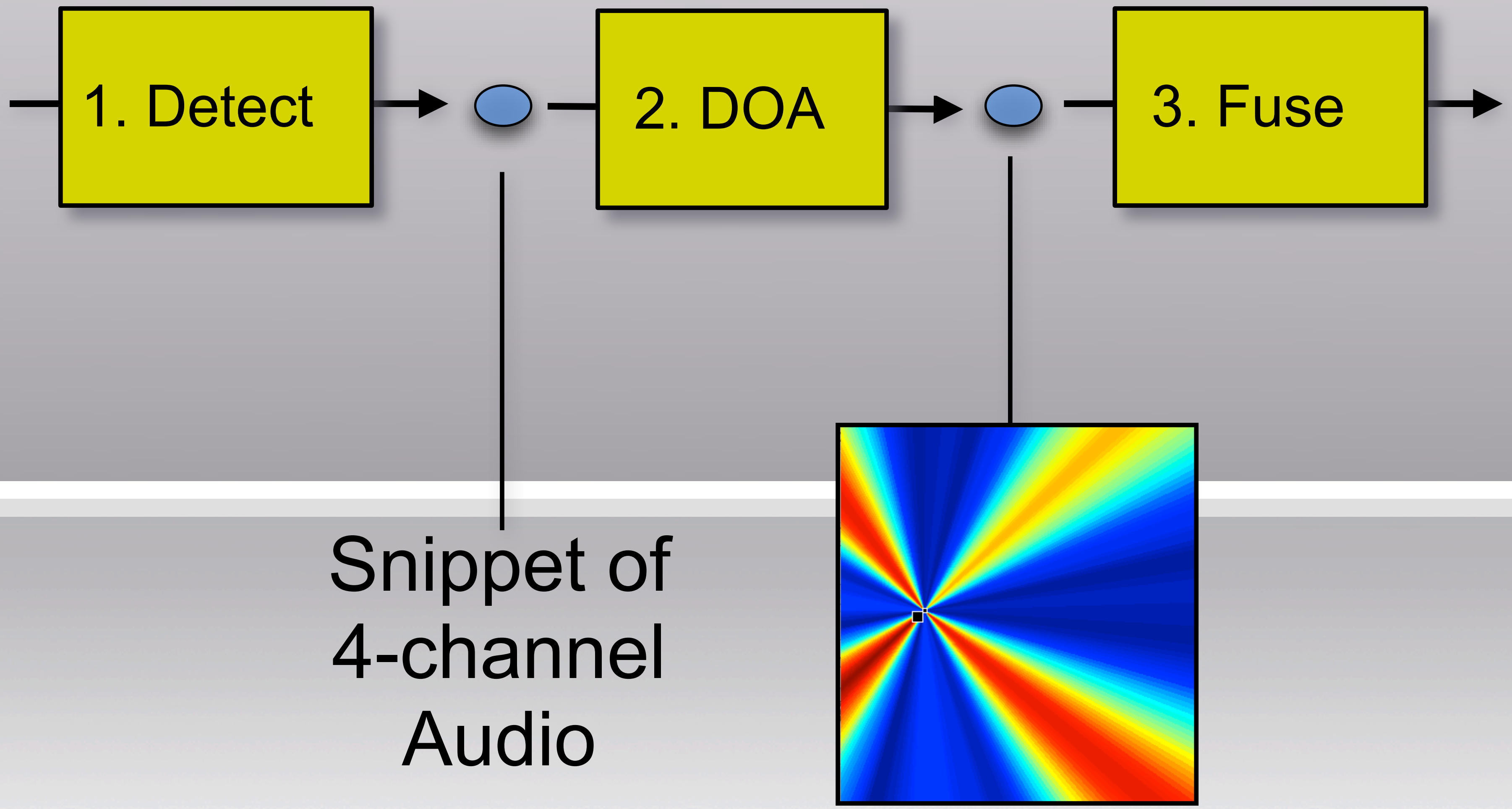
3. Fuse

Three phases to marmot localization



Snippet of
4-channel
Audio

Three phases to marmot localization



1. Detect

2. DOA

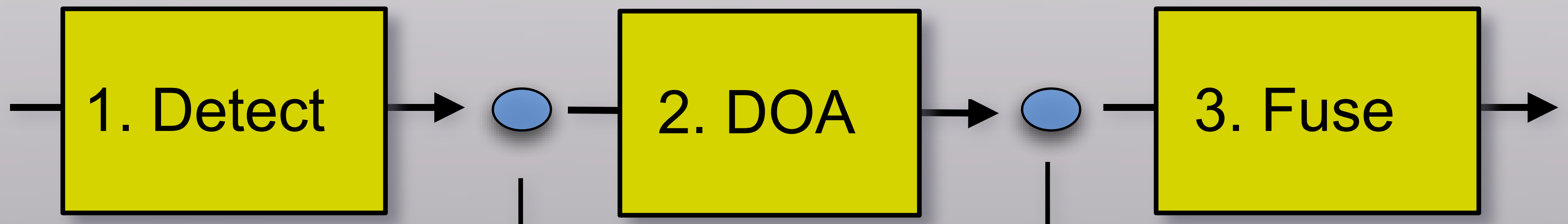
3. Fuse

Snippet of 4-channel Audio

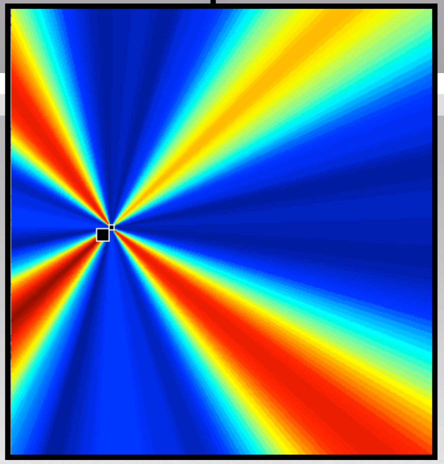
Direction of arrival

http://wavescope.csail.mit.edu/

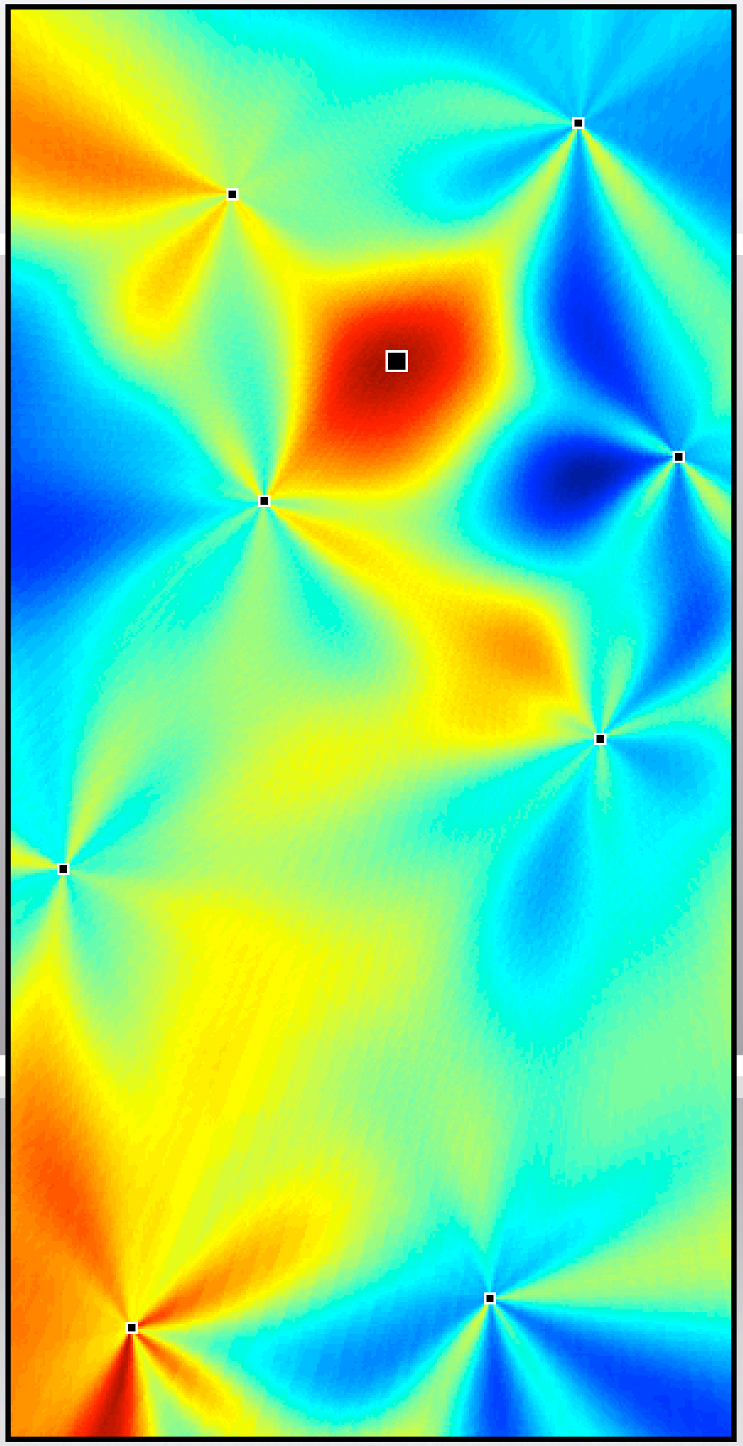
Three phases to marmot localization



Snippet of
4-channel
Audio



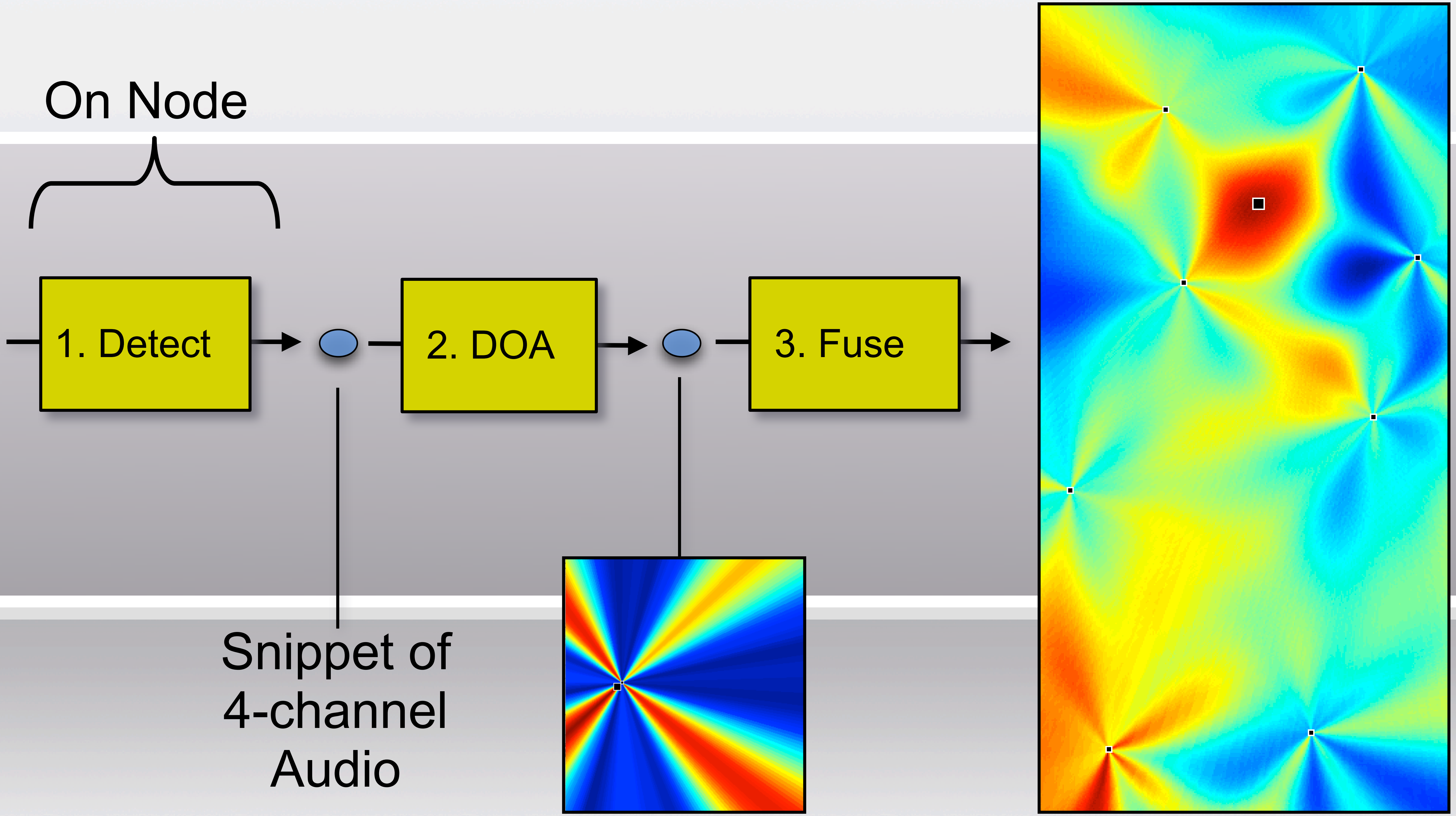
Direction of arrival



Probability Map

http://wavescope.csail.mit.edu/

Three phases to marmot localization



On Node

1. Detect

2. DOA

3. Fuse

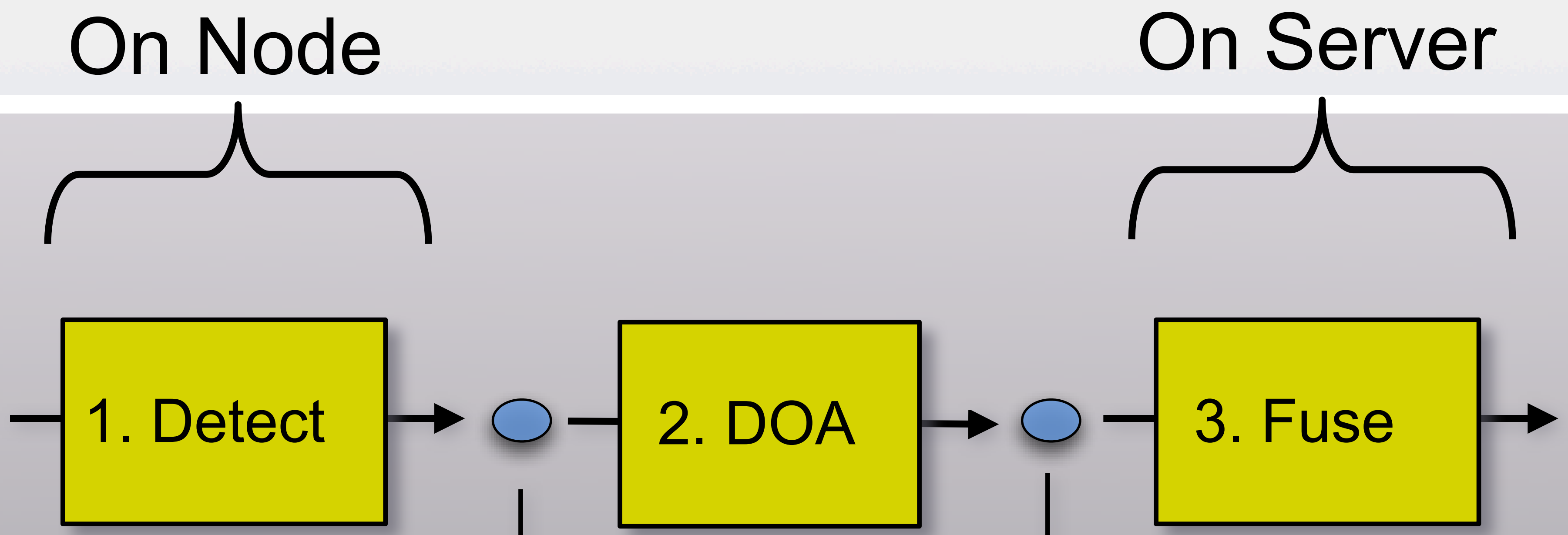
Snippet of 4-channel Audio

Direction of arrival

Probability Map

http://wavescope.csail.mit.edu/

Three phases to marmot localization



On Node

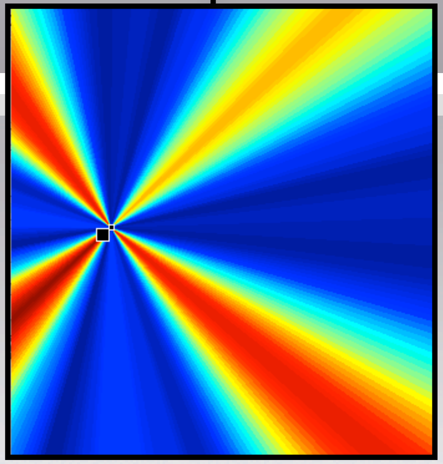
On Server

1. Detect

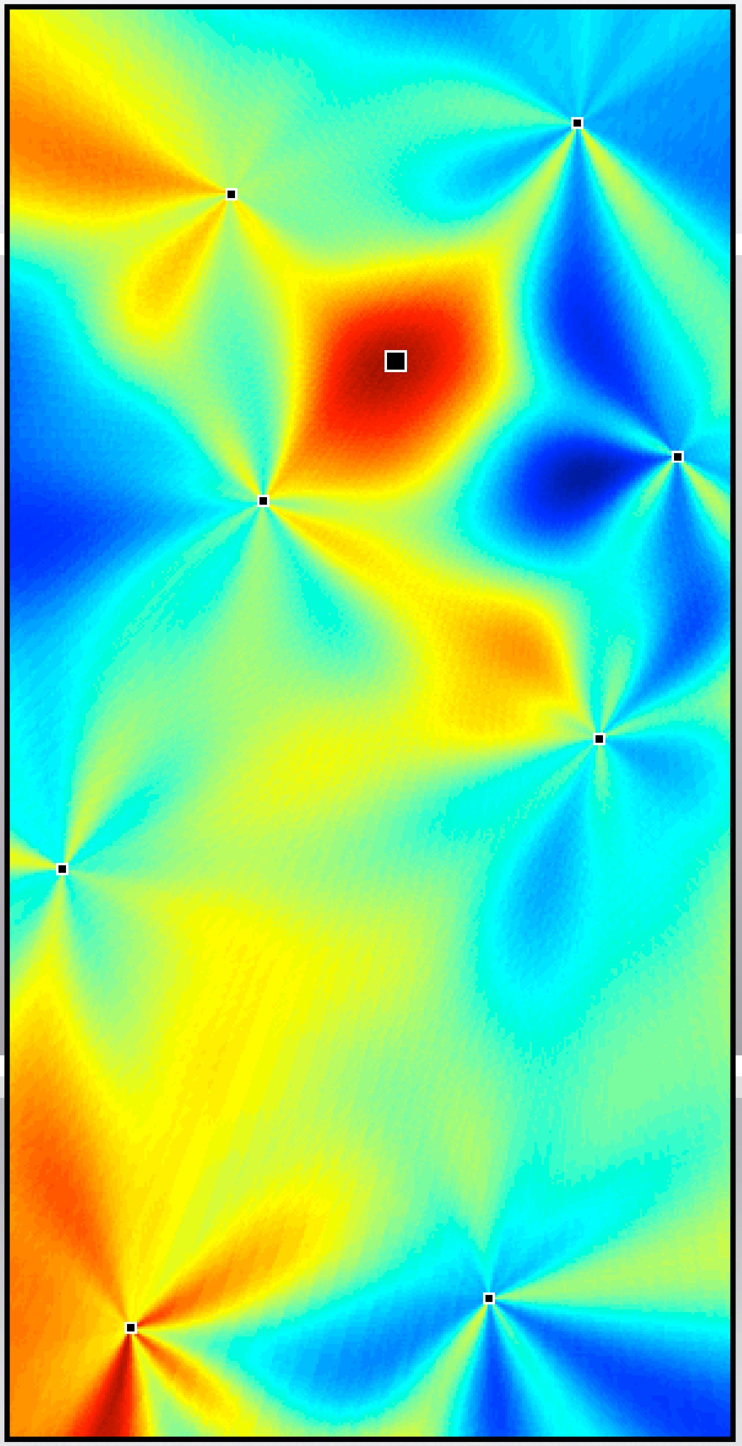
2. DOA

3. Fuse

Snippet of
4-channel
Audio



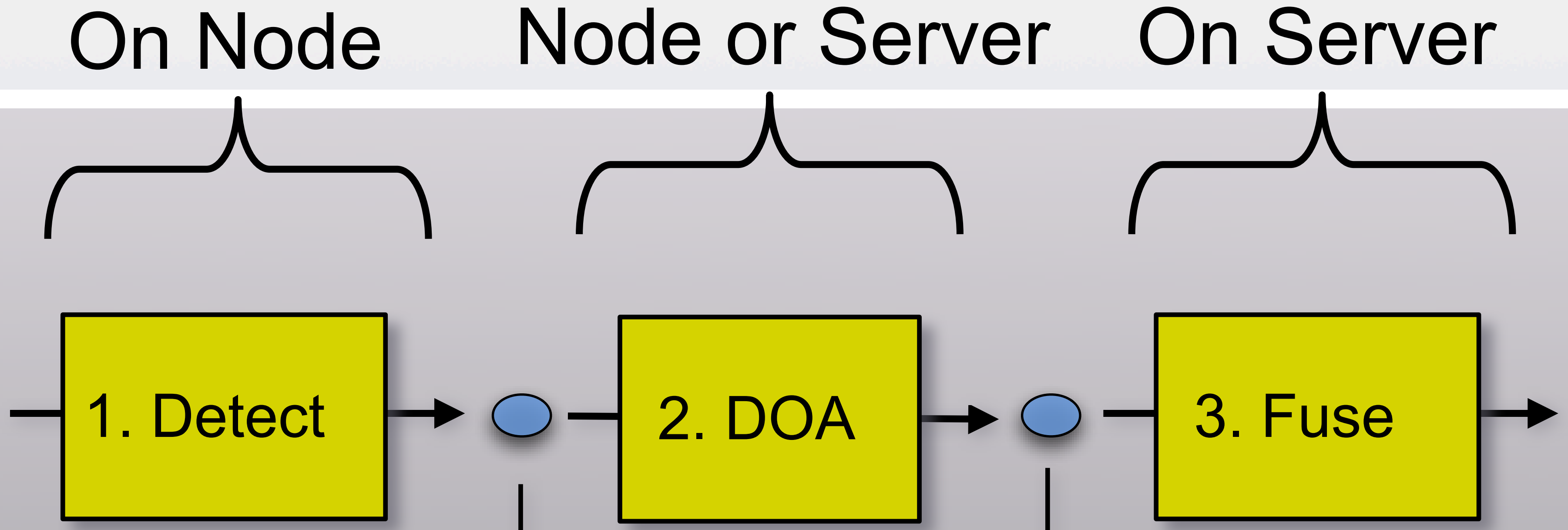
Direction of arrival



Probability Map

http://wavescope.csail.mit.edu/

Three phases to marmot localization



On Node

Node or Server

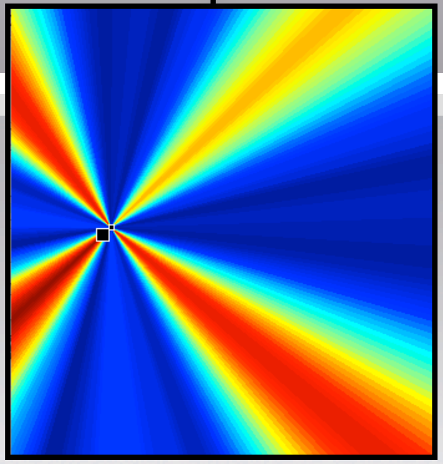
On Server

1. Detect

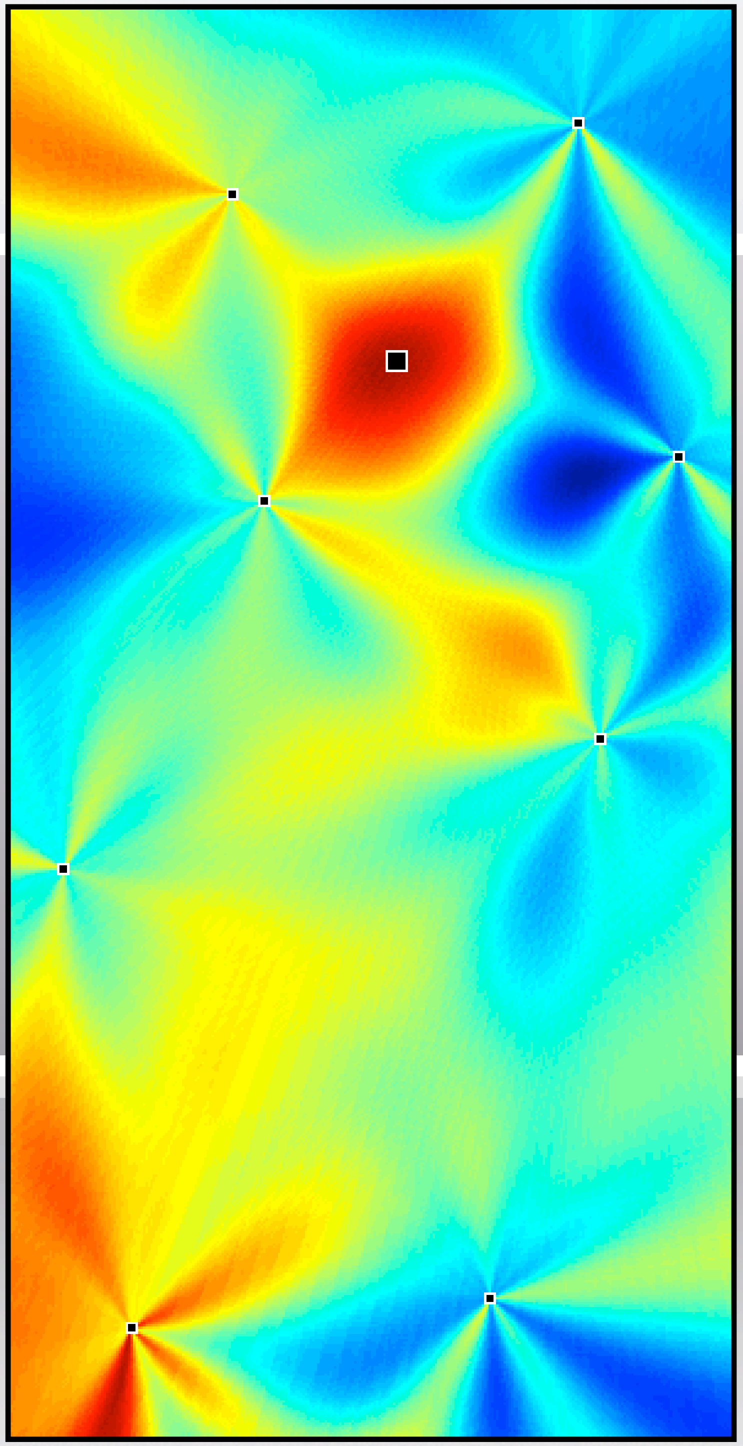
2. DOA

3. Fuse

Snippet of 4-channel Audio



Direction of arrival

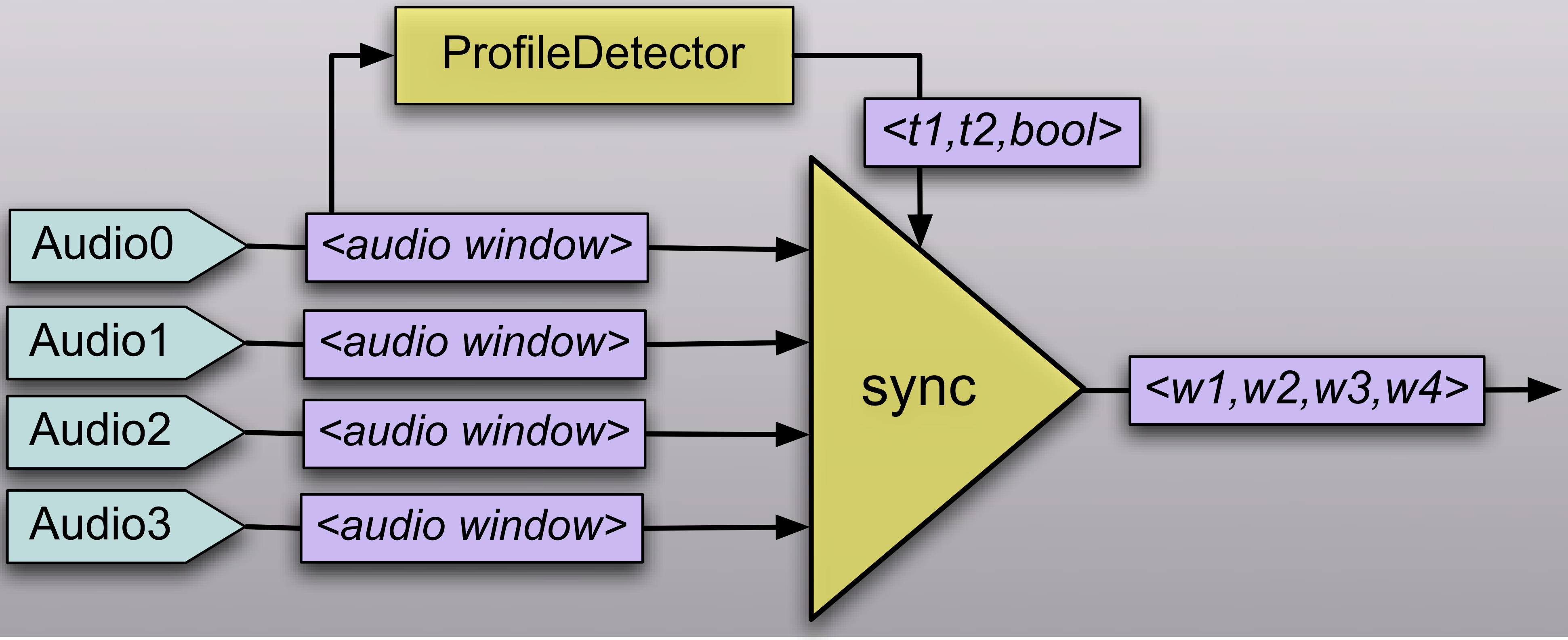


Probability Map

http://wavescope.csail.mit.edu/

Schematic of Marmot-detector

(Phase1)

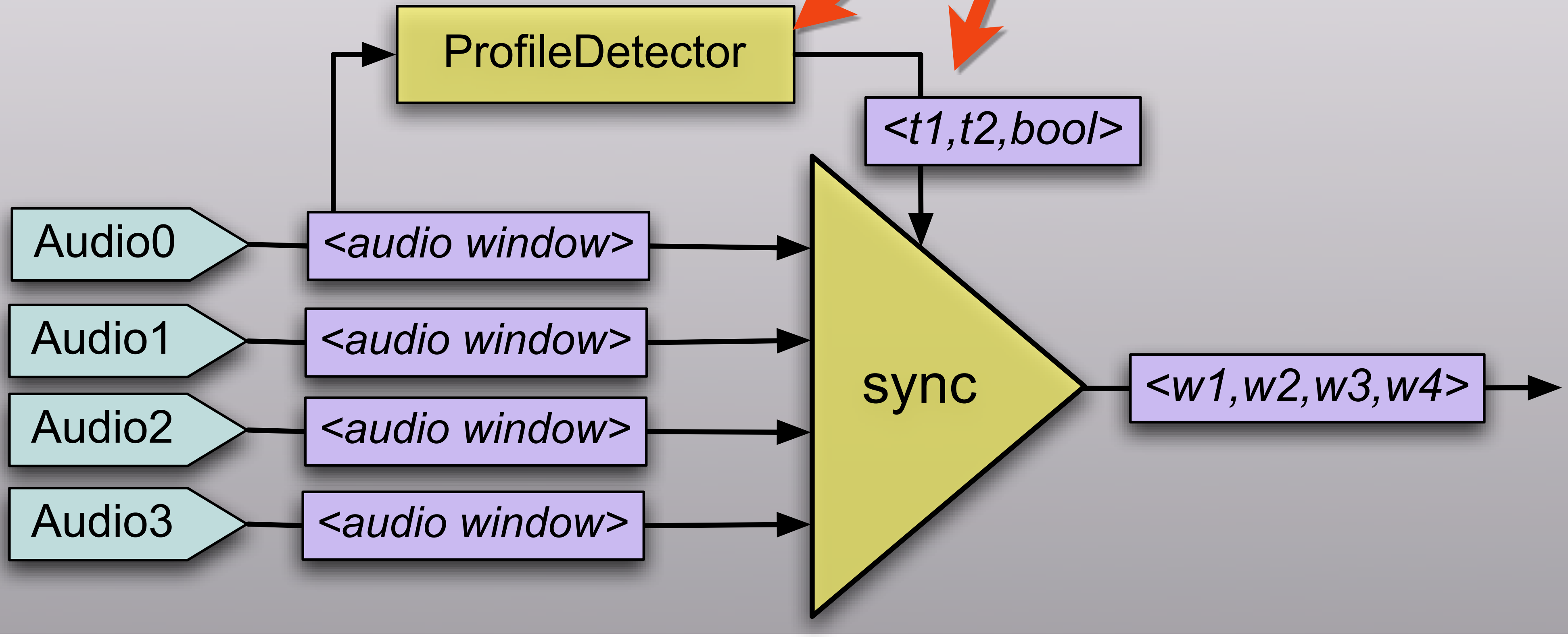


http://wavescope.csail.mit.edu/

Schematic of Marmot-detector

(Phase1)

Fast-path DSP to determine temporal ranges for marmot calls



ProfileDetector

<t1,t2,bool>

Audio0

Audio1

Audio2

Audio3

<audio window>

<audio window>

<audio window>

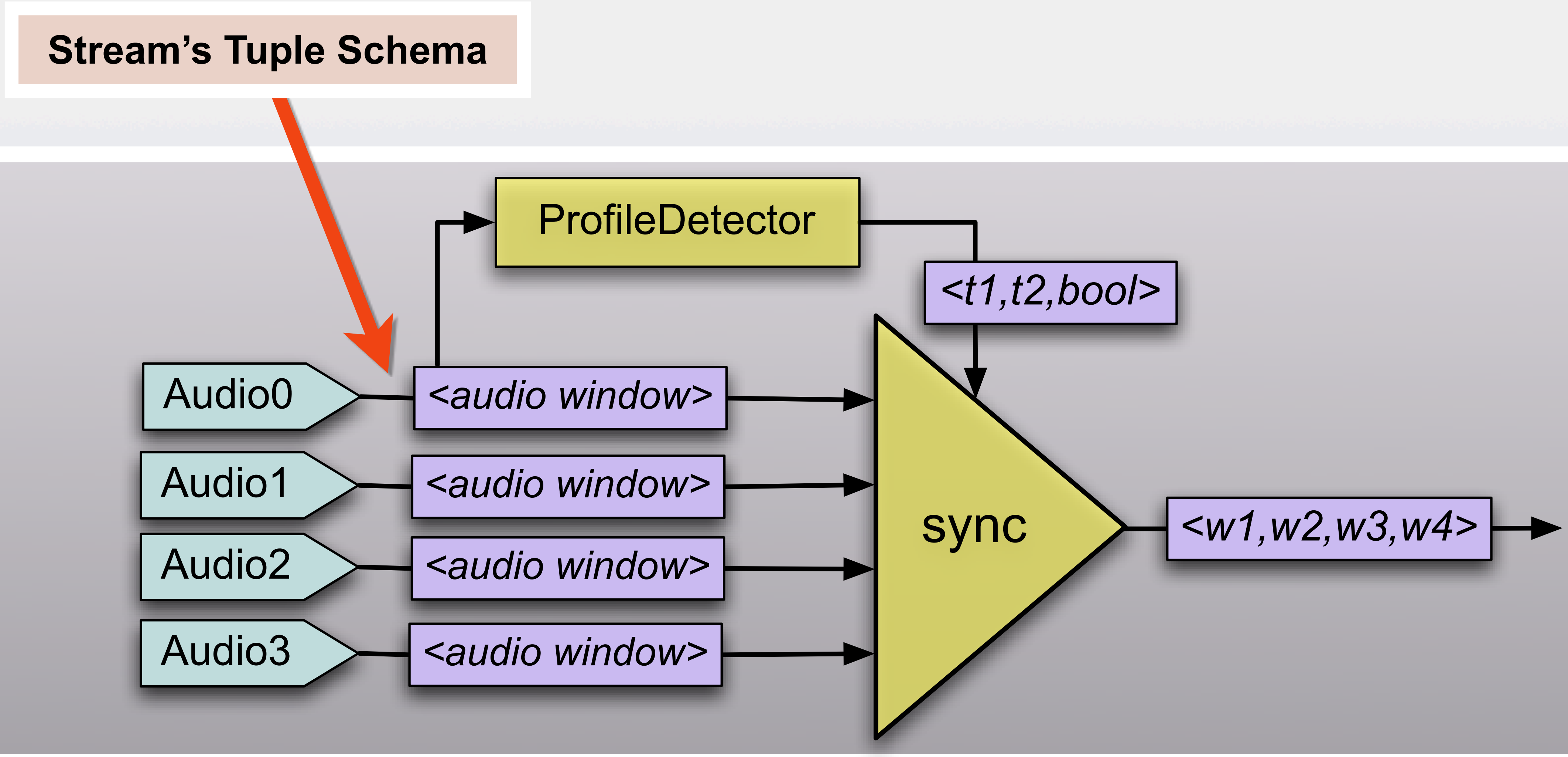
<audio window>

sync

<w1,w2,w3,w4>

Schematic of Marmot-detector

(Phase1)



http://wavescope.csail.mit.edu/

Schematic of Marmot-detector

(Phase1)

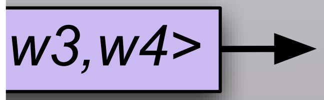
Stream’s Tuple Schema

Data Model:

- **Streams are first-class values.**

- **Streams contain any type but Stream**
  - Algebraic data-types (but not recursive)
  - Size of every type is statically known
  - Dynamic allocation allowed

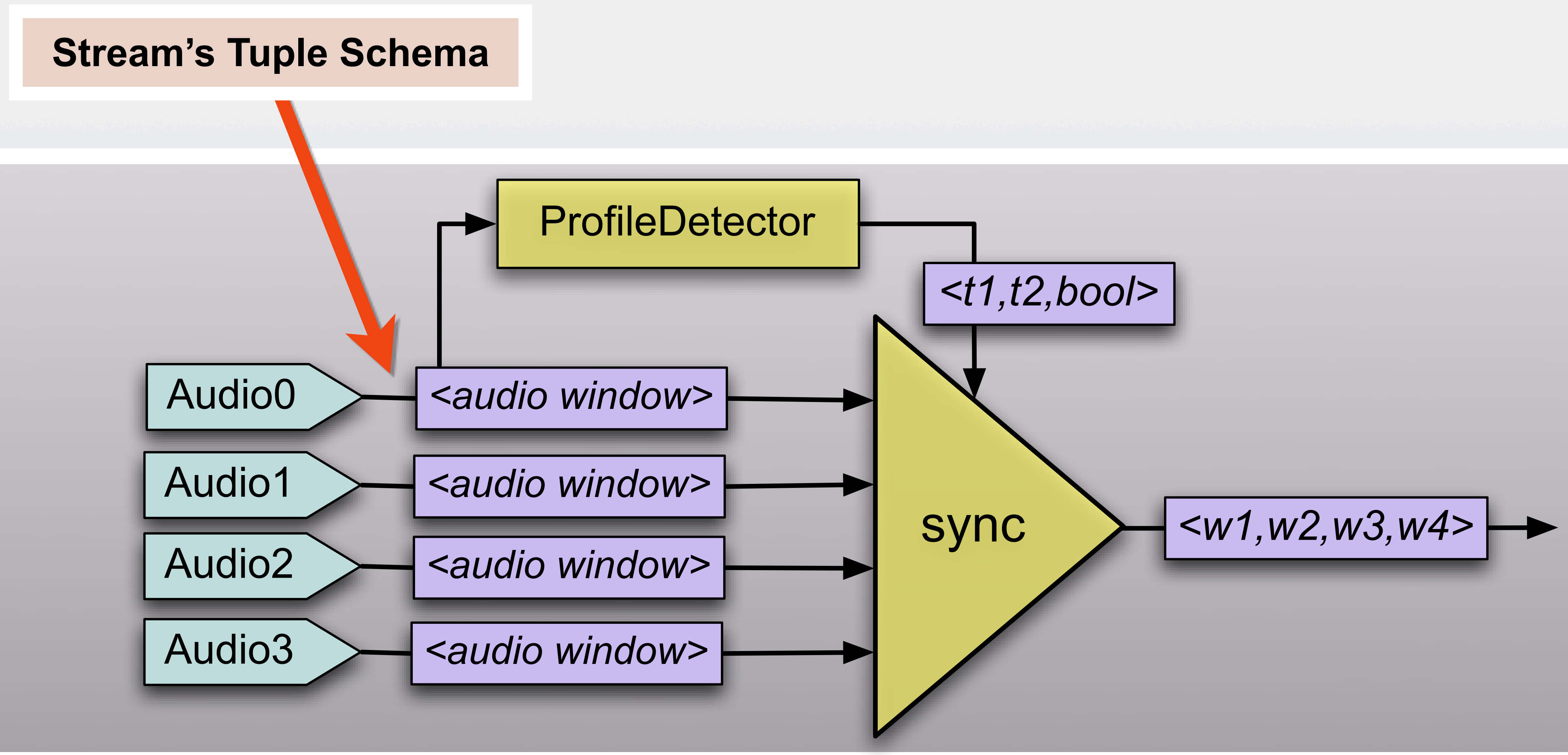
- **SigSegs: efficiently managed windows of samples**
  - cheap to append, copy, forward, rewindow
  - fewer timestamps



http://wavescope.csail.mit.edu/

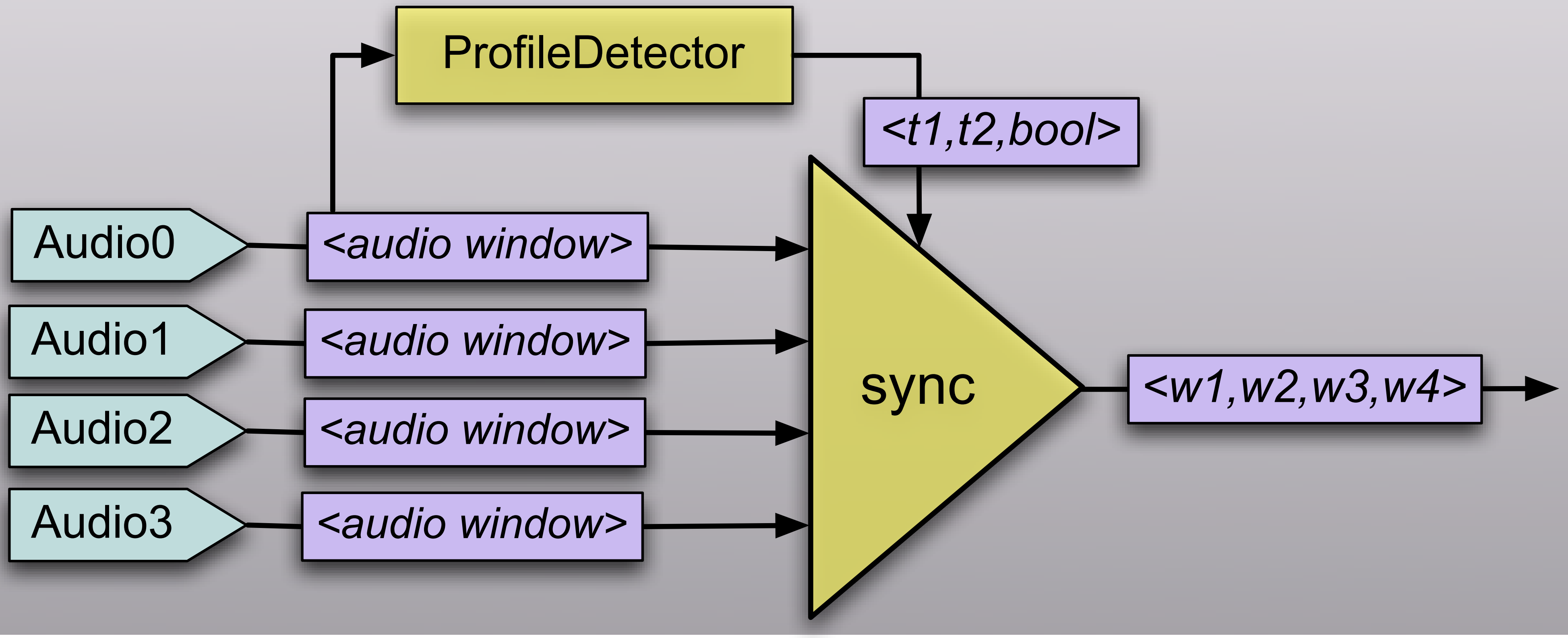
Schematic of Marmot-detector

(Phase1)



http://wavescope.csail.mit.edu/

WaveScript detector code



ProfileDetector

<t1,t2,bool>

Audio0

Audio1

Audio2

Audio3

<audio window>

<audio window>

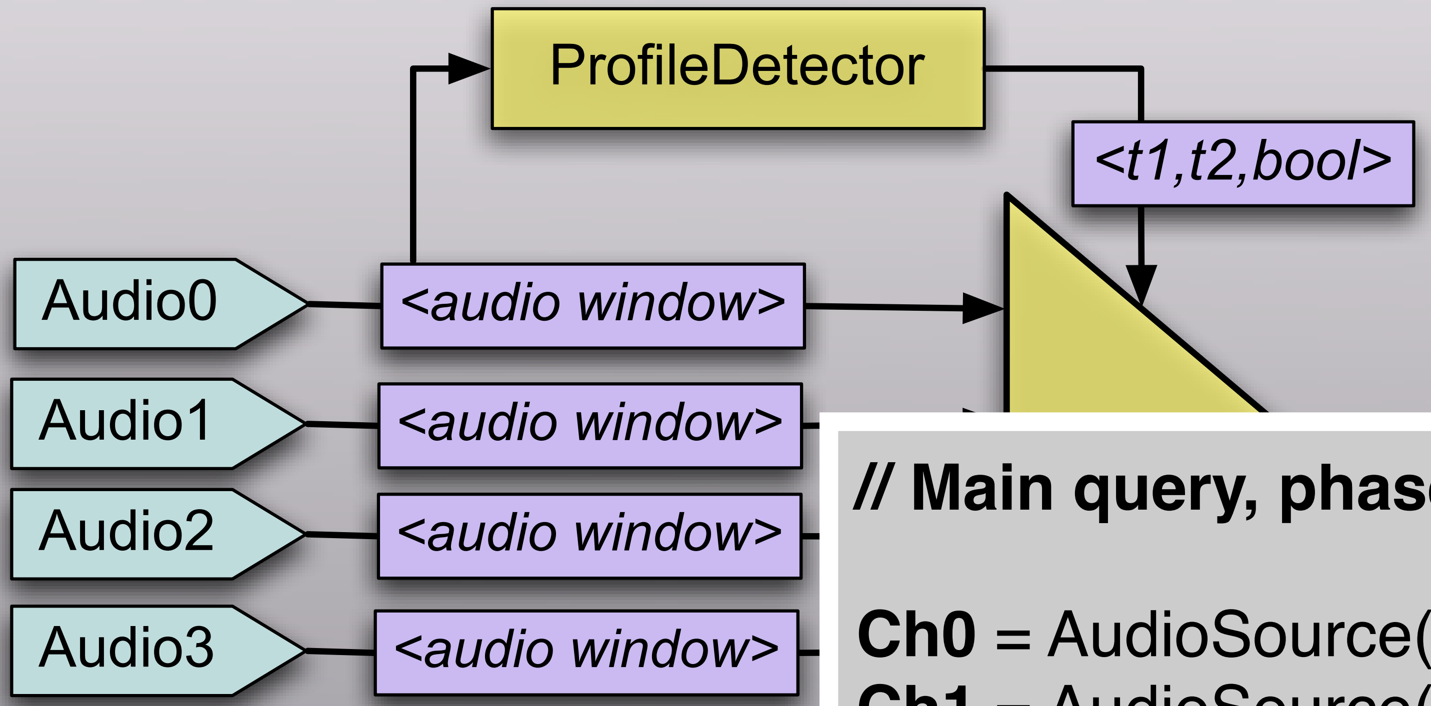
<audio window>

<audio window>

sync

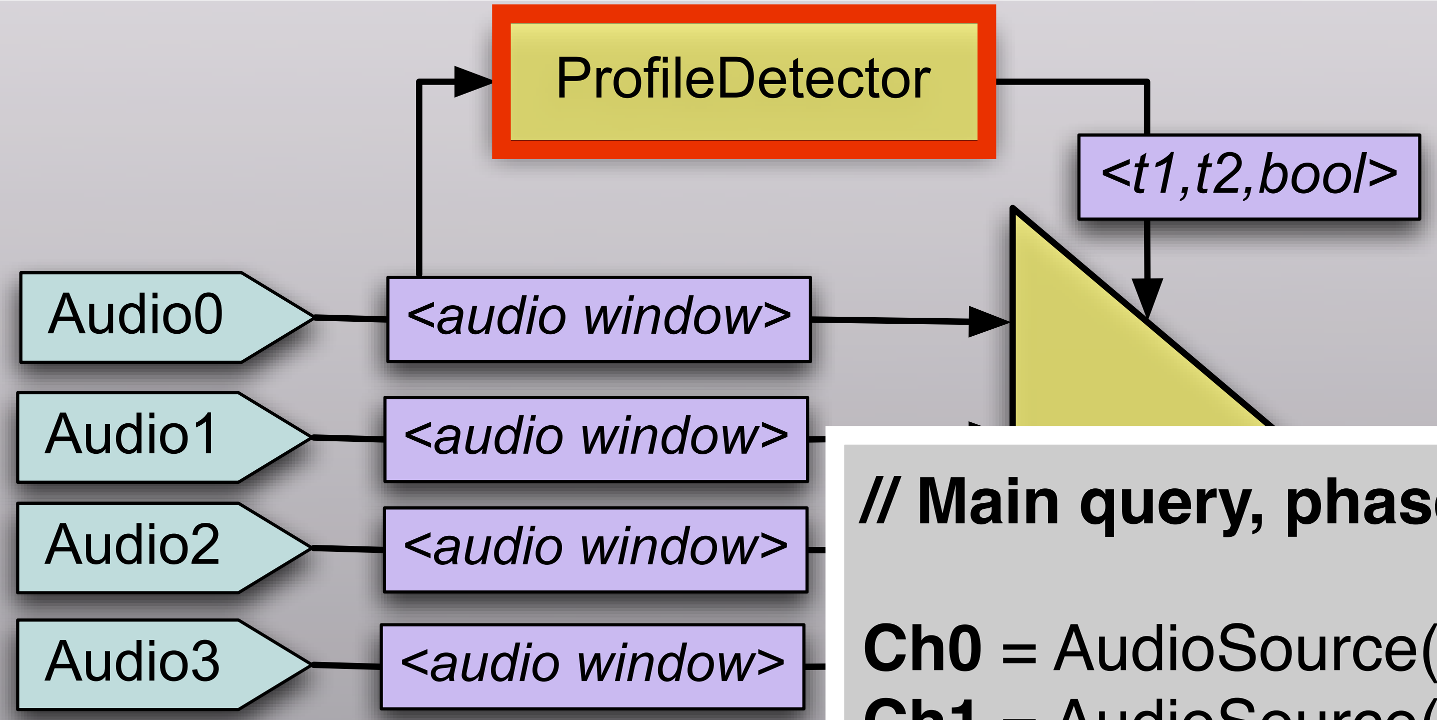
<w1,w2,w3,w4>

WaveScript detector code



```
// Main query, phase 1

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                        (64,192));
datawindows = sync4(control, Ch0, Ch1,
                              Ch2, Ch4);
```

WaveScript detector code



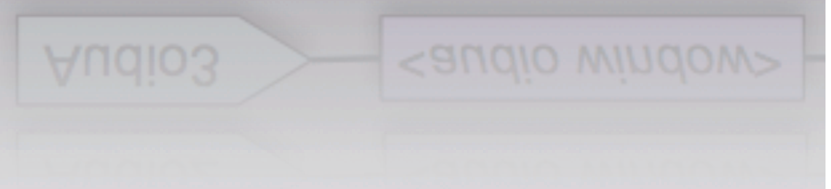
```
// Main query, phase 1

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                        (64,192));
datawindows = sync4(control, Ch0, Ch1,
                               Ch2, Ch4);
```

http://wavescope.csail.mit.edu/

WaveScript detector code

```
fun profileDetect(S : Stream (SigSeg Int16),
                      scorefun,
                      (winsize,step))
{
   wins = rewindow(S, winsize, step);
   scores : Stream Float
   scores = map(scorefun o FFT, wins);
   withscores : Stream (Float, SigSeg Int16)
   withscores = zip2(scores, wins);
   return threshFilter(withscores);
}
```

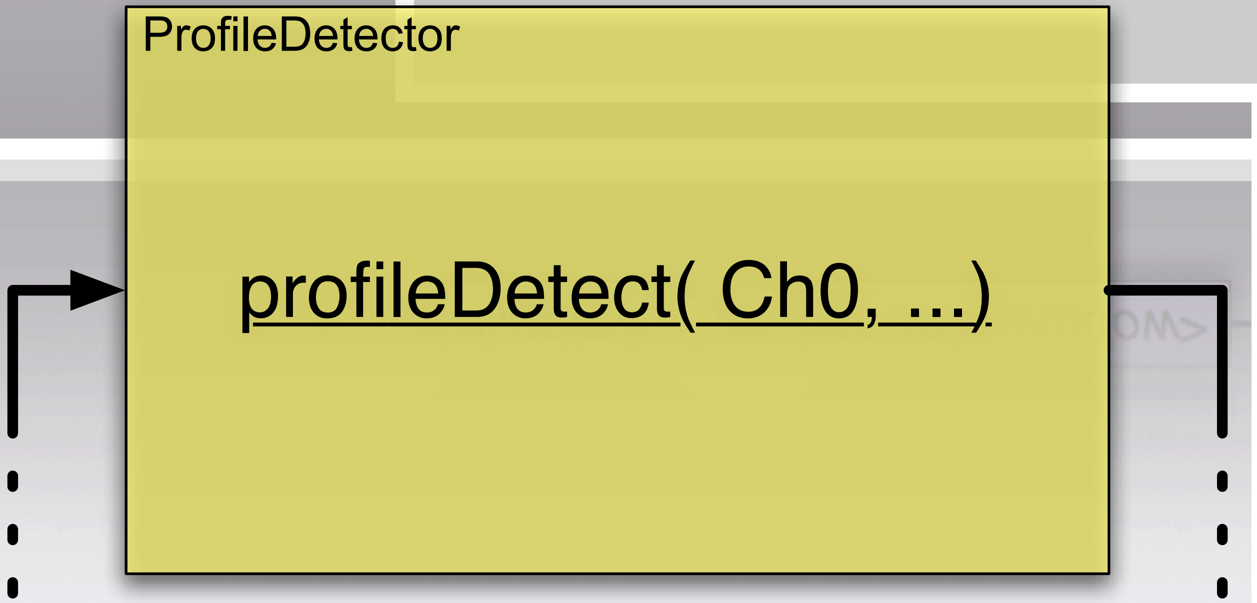
```
ase 1

e(0, 48000, 1024);
e(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                              (64,192));

datawindows = sync4(control, Ch0, Ch1,
                                  Ch2, Ch4);
```



http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S : Stream (SigSeg Int16),
                          scorefun,
                          (winsize,step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}
```

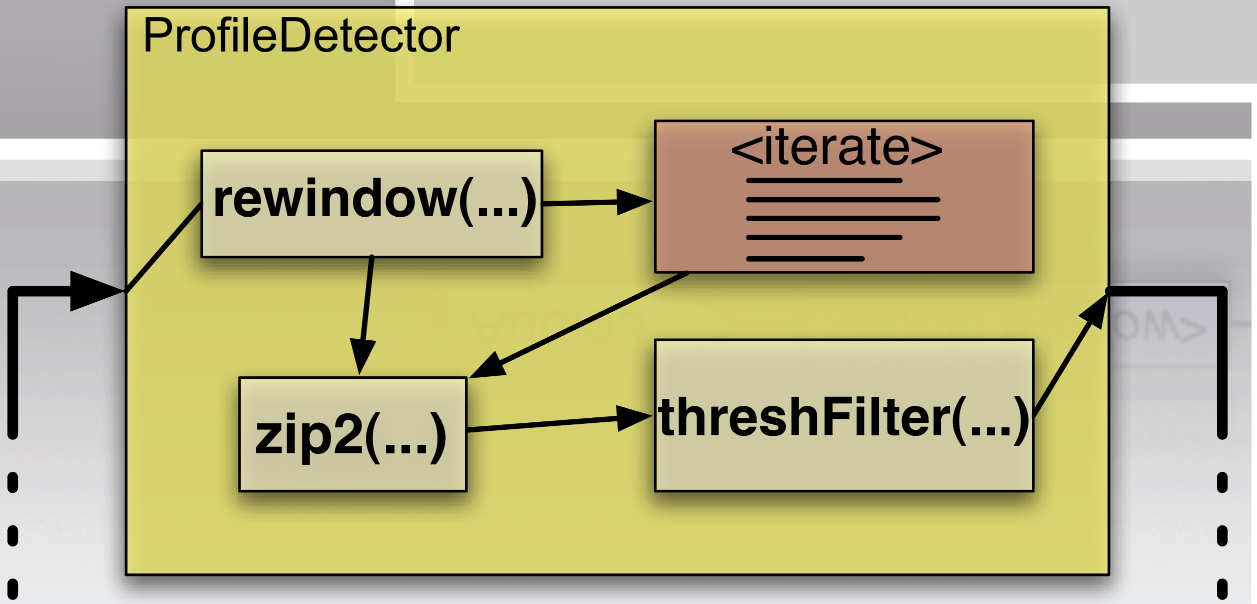


```
ase 1

e(0, 48000, 1024);
e(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                                   (64,192));

datawindows = sync4(control, Ch0, Ch1,
                                     Ch2, Ch4);
```

http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S : Stream (SigSeg Int16),
                   scorefun,
                   (winsize,step))
{
   wins = rewindow(S, winsize, step);
   scores : Stream Float
   scores = map(scorefun o FFT, wins);
   withscores : Stream (Float, SigSeg Int16)
   withscores = zip2(scores, wins);
   return threshFilter(withscores);
}
```



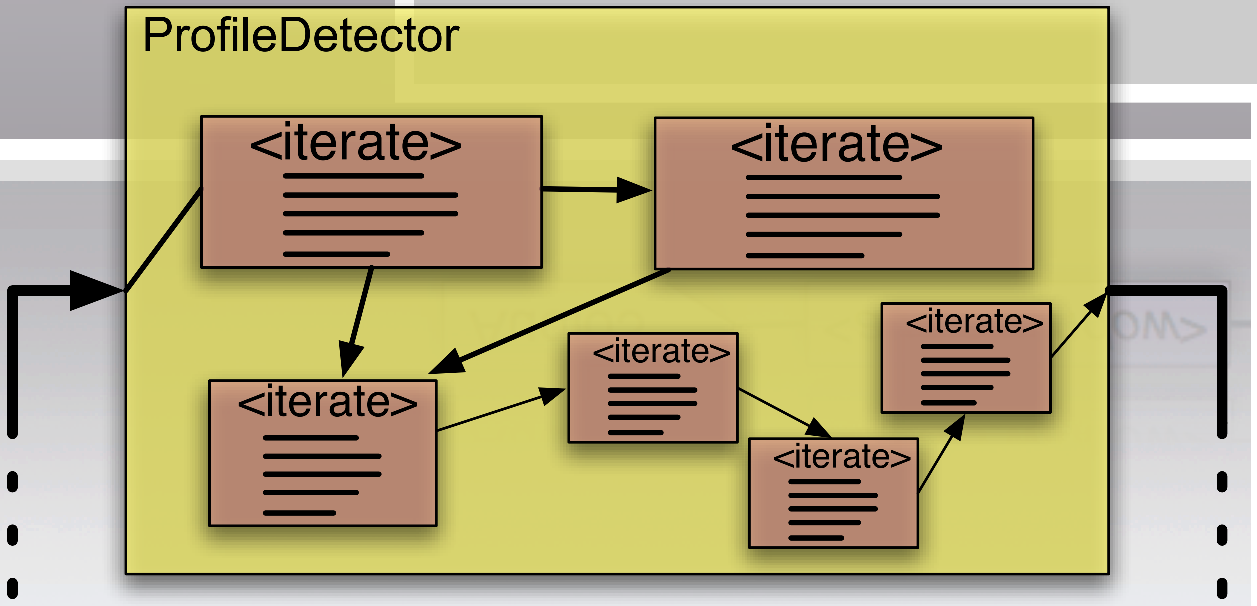
```
ase 1

e(0, 48000, 1024);
e(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                         (64,192));

datawindows = sync4(control, Ch0, Ch1,
                                  Ch2, Ch4);
```

http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S : Stream (SigSeg Int16),
                        scorefun,
                        (winsize,step))
{
   wins = rewindow(S, winsize, step);
   scores : Stream Float
   scores = map(scorefun o FFT, wins);
   withscores : Stream (Float, SigSeg Int16)
   withscores = zip2(scores, wins);
   return threshFilter(withscores);
}
```

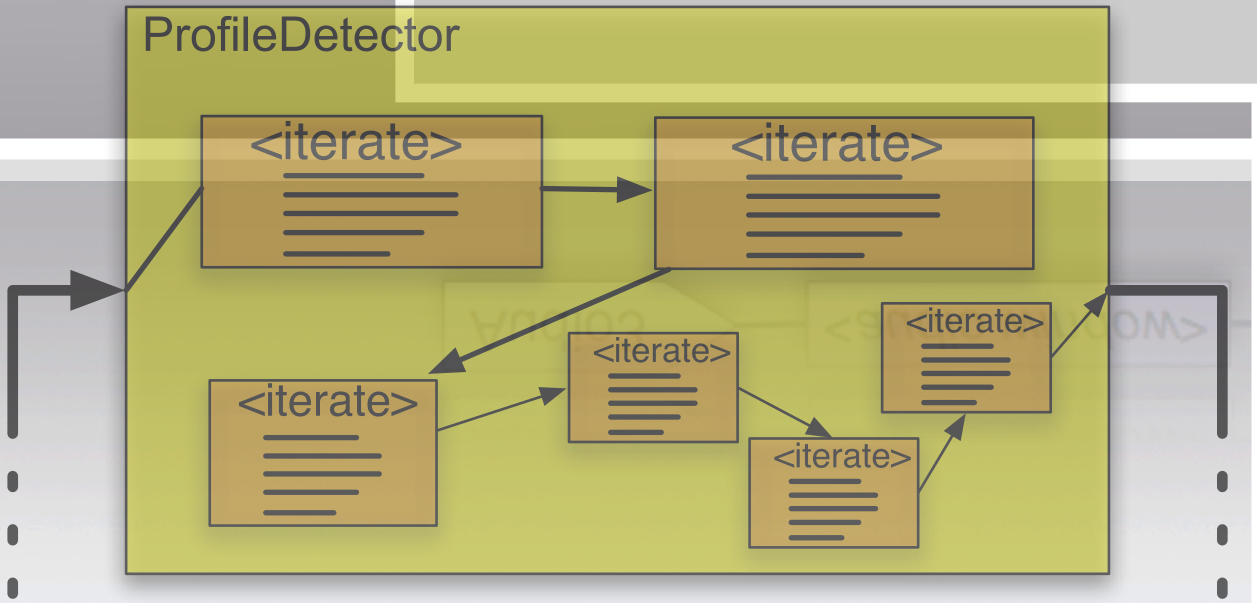
```
ase 1

e(0, 48000, 1024);
e(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                         (64,192));

datawindows = sync4(control, Ch0, Ch1,
                                  Ch2, Ch4);
```



http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S : Stream (SigSeg Int16),
                      scorefun,
                      (winsize,step))
{
   wins = rewindow(S, winsize, step);
   scores : Stream Float
   scores = map(scorefun o FFT, wins);
   withscores : Stream (Float, SigSeg Int16)
   withscores = zip2(scores, wins);
   return threshFilter(withscores);
}
```



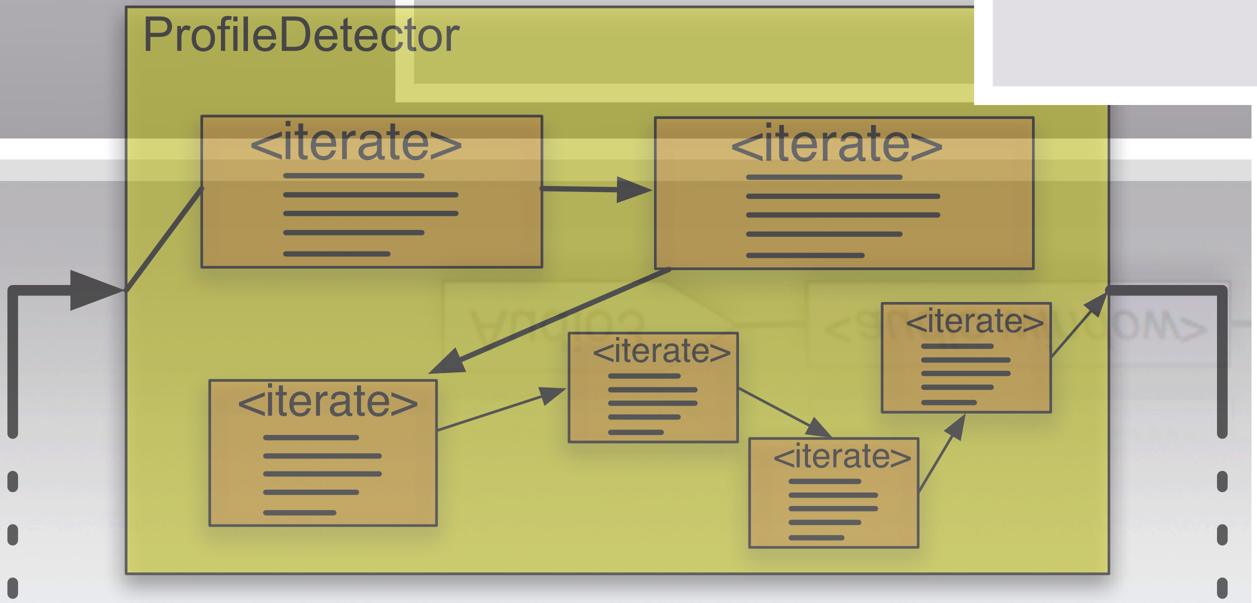
```
ase 1

e(0, 48000, 1024);
e(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                        (64,192));

datawindows = sync4(control, Ch0, Ch1,
                                  Ch2, Ch4);
```

http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S
                  s
                  (
{
  wins = rewindo
  scores : Stream
  scores = map(s
  withscores : Str
  withscores = zi
  return threshFil
}
```

“Wide-band” Language



```
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                        (64,192));

datawindows = sync4(control, Ch0, Ch1,
                    Ch2, Ch4);
```

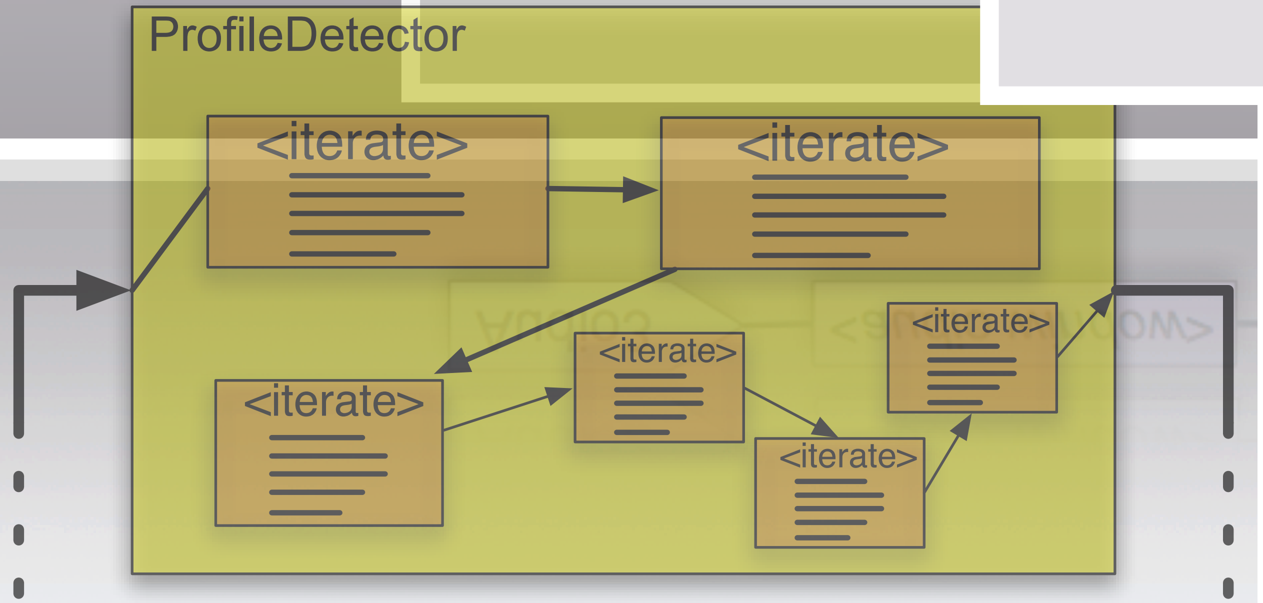
http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S
                  s
                  (
{
   wins = rewindo
   scores : Stream
   scores = map(s
   withscores : Str
   withscores = zi
   return threshFil
}
```

“Wide-band” Language

- High-level, query-like declarative programs
  - map, project, filter streams
  - apply library signal-processing ops



```
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                         (64,192));

datawindows = sync4(control, Ch0, Ch1,
                    Ch2, Ch4);
```

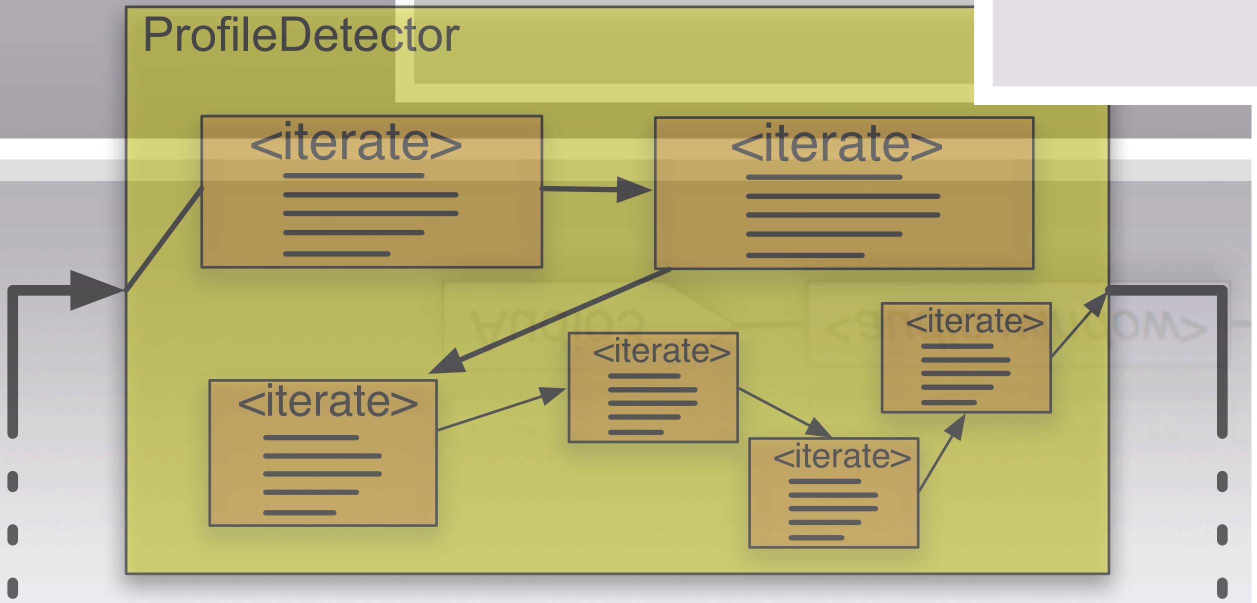
http://wavescope.csail.mit.edu/

WaveScript Code for Detector

```
fun profileDetect(S
                  s
                  (
{
  wins = rewindo
  scores : Stream
  scores = map(s
  withscores : St
  withscores = zip
  return threshFil
}
```

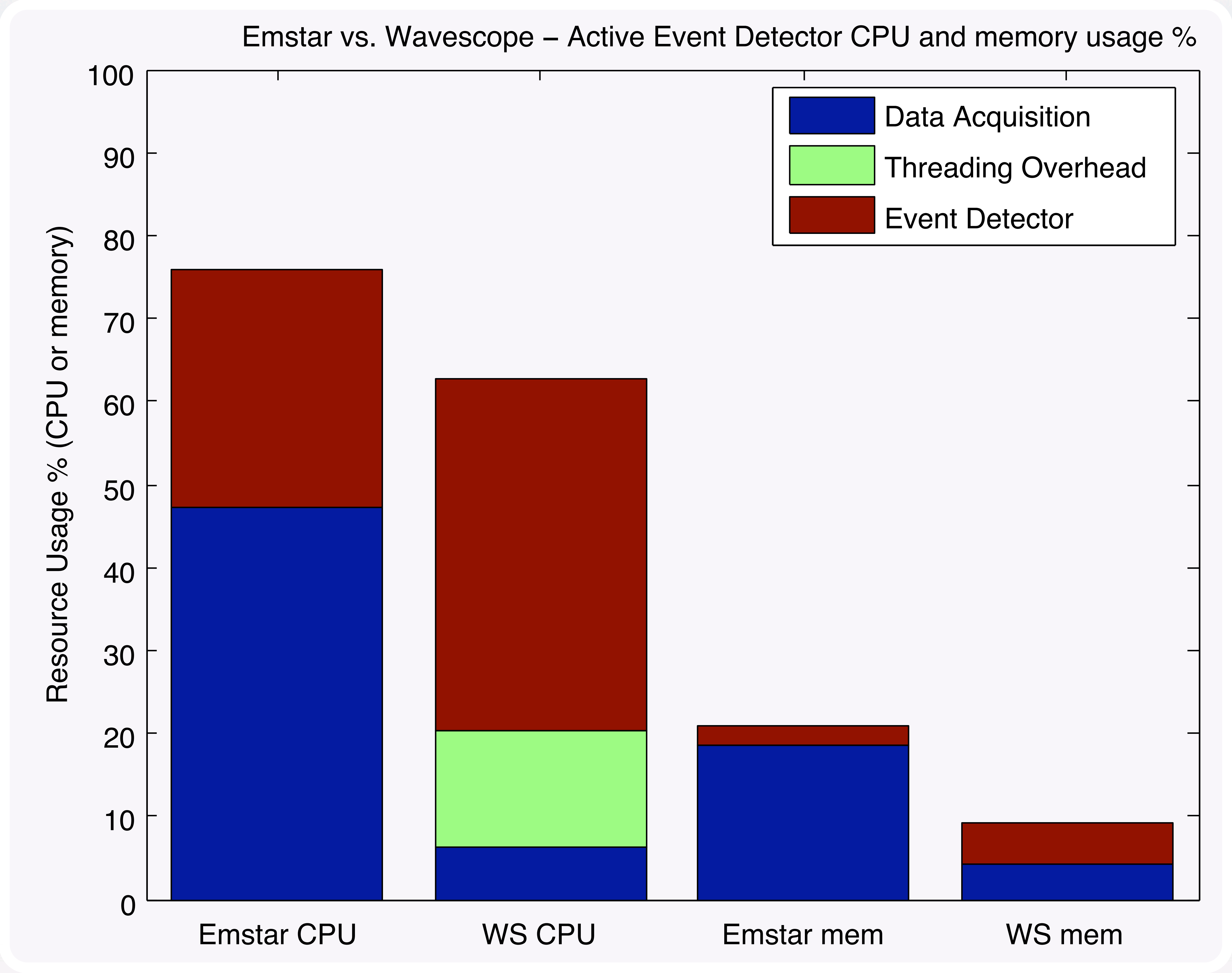
“Wide-band” Language

- High-level, query-like declarative programs
  - map, project, filter streams
  - apply library signal-processing ops
- Low-level, imperative code within custom-operators
  - use iterate to introduce



```
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore,
                        (64,192));

datawindows = sync4(control, Ch0, Ch1,
                    Ch2, Ch4);
```

Comparing to handwritten C



Comparing to handwritten C

Step 2. Direction of arrival search exec times in seconds

	Min	Mean	Max
C	2.85	3.05	3.4
WS	2.2	2.4	2.8

(Using MLton backend)

Implementation: *Leveraging the DS in DSL*

Implementation: *Leveraging the DS in DSL*

- No shared state between operators.

Implementation: *Leveraging the DS in DSL*

- No shared state between operators.
- We can manage separate heaps efficiently
  - This suggests an efficient form of delayed reference counting (next slides)

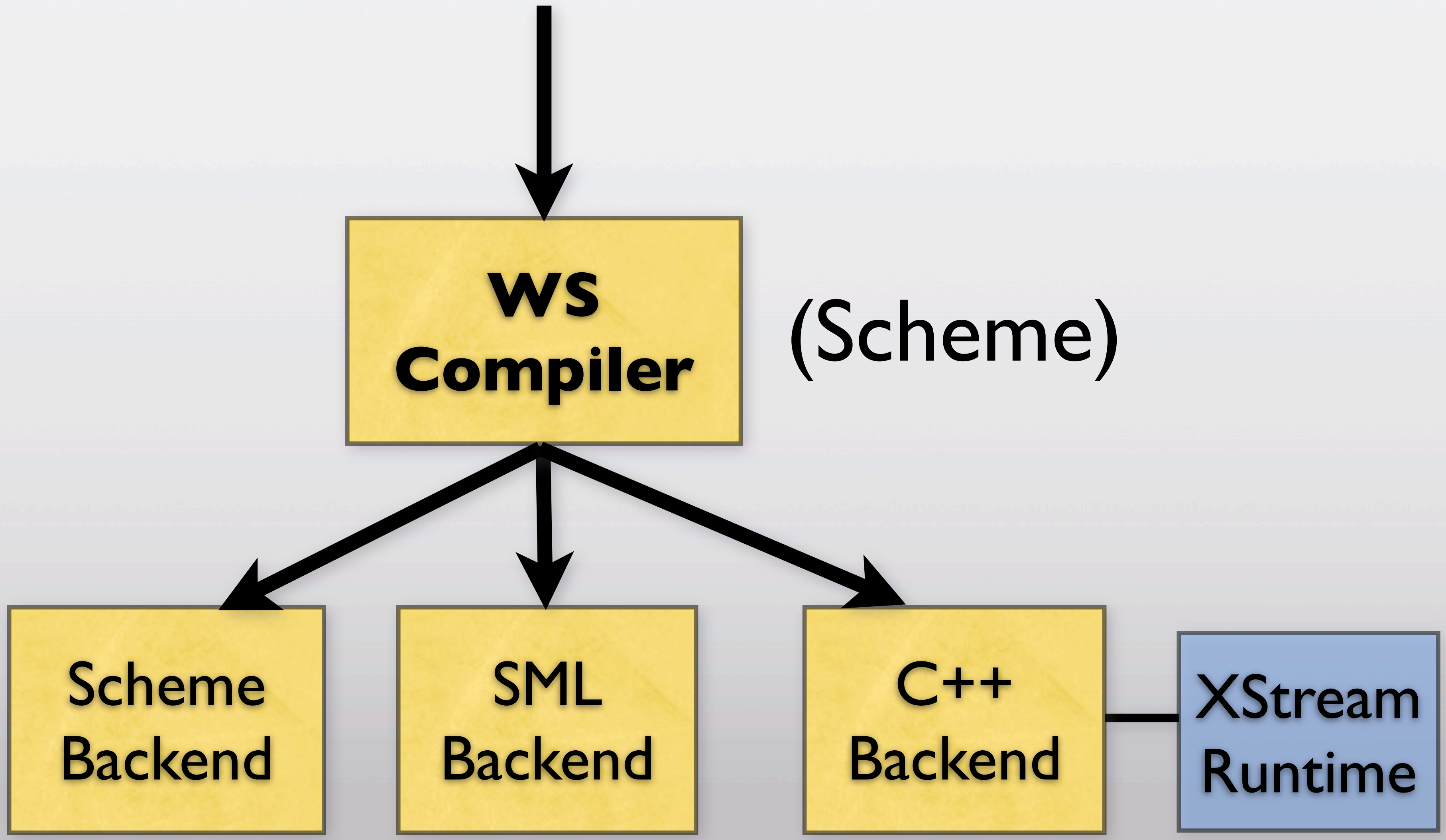
Implementation:
Leveraging the DS in DSL

- No shared state between operators.
- We can manage separate heaps efficiently
  - This suggests an efficient form of delayed reference counting (next slides)
- Distributed execution
  - Stream graph executes across multiple machines

Implementation: *Leveraging the DS in DSL*

- No shared state between operators.
- We can manage separate heaps efficiently
  - This suggests an efficient form of delayed reference counting (next slides)
- Distributed execution
  - Stream graph executes across multiple machines
- Intra-machine parallelism (multicore/processor)
  - For example, processing terabytes of offline data

Backends, Cont.



WS Compiler

(Scheme)

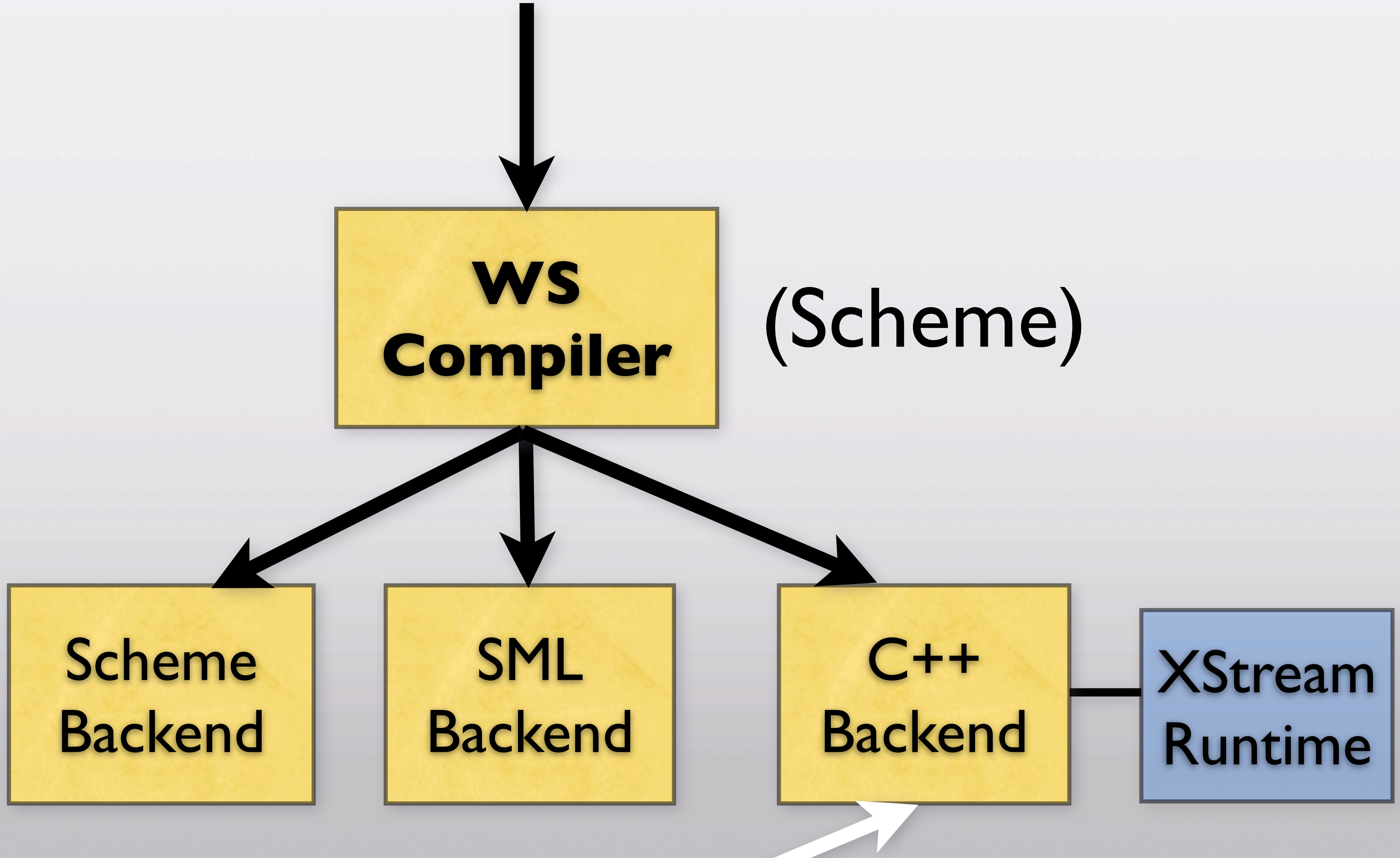
Scheme Backend

SML Backend

C++ Backend

XStream Runtime

Backends, Cont.



WS Compiler

(Scheme)

Scheme Backend

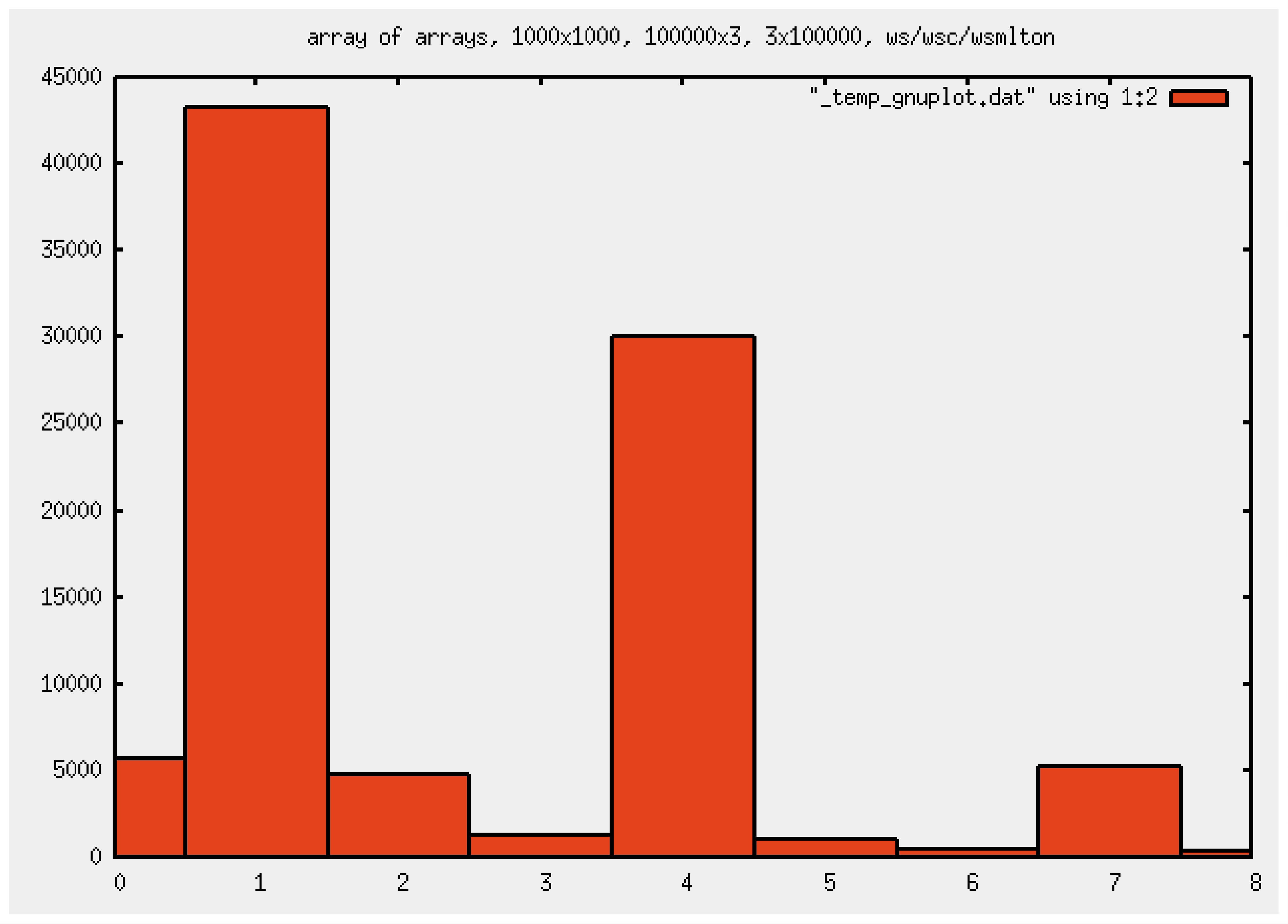
SML Backend

C++ Backend

XStream Runtime

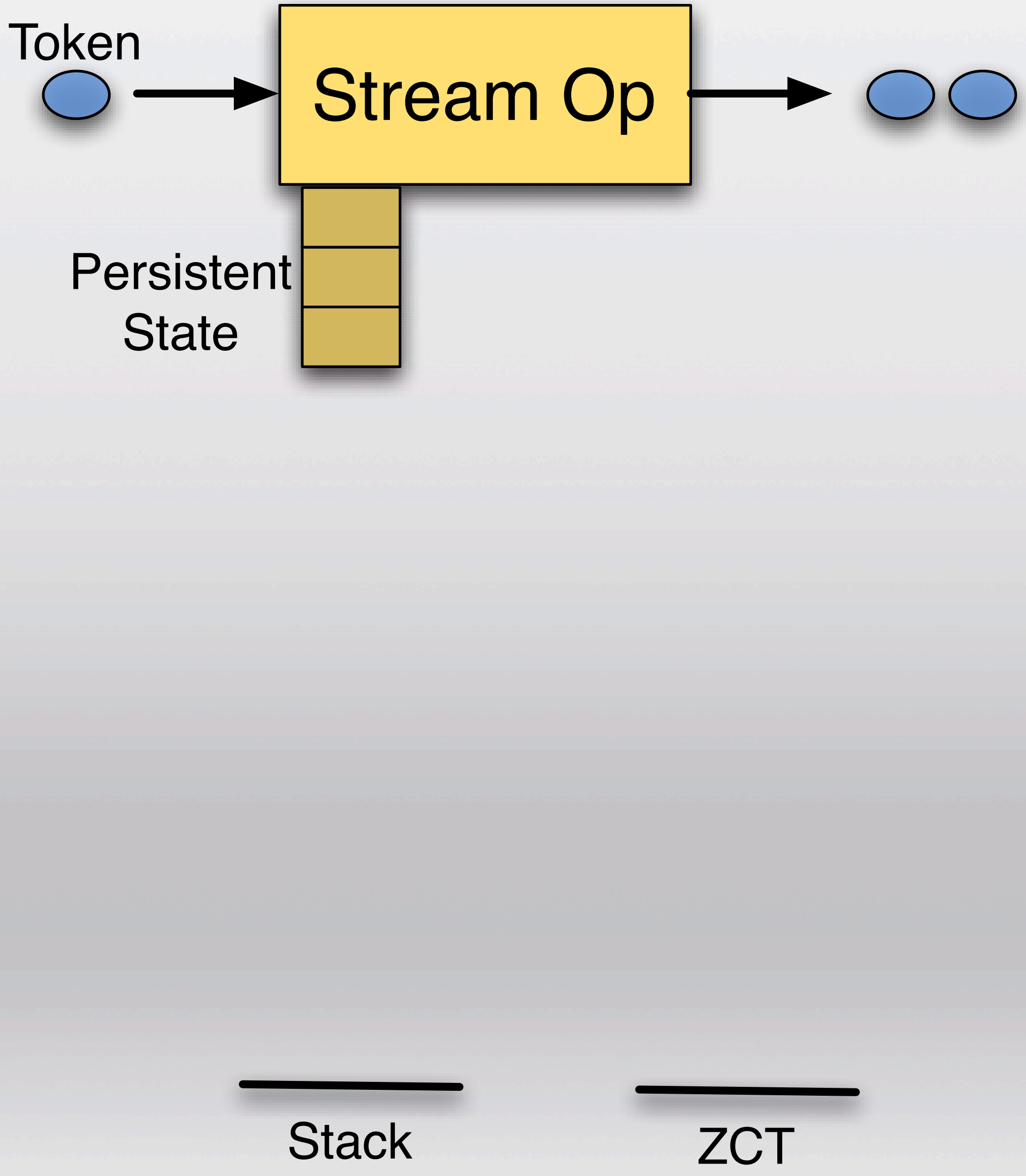
Currently: naive reference counting

Backends, Cont.



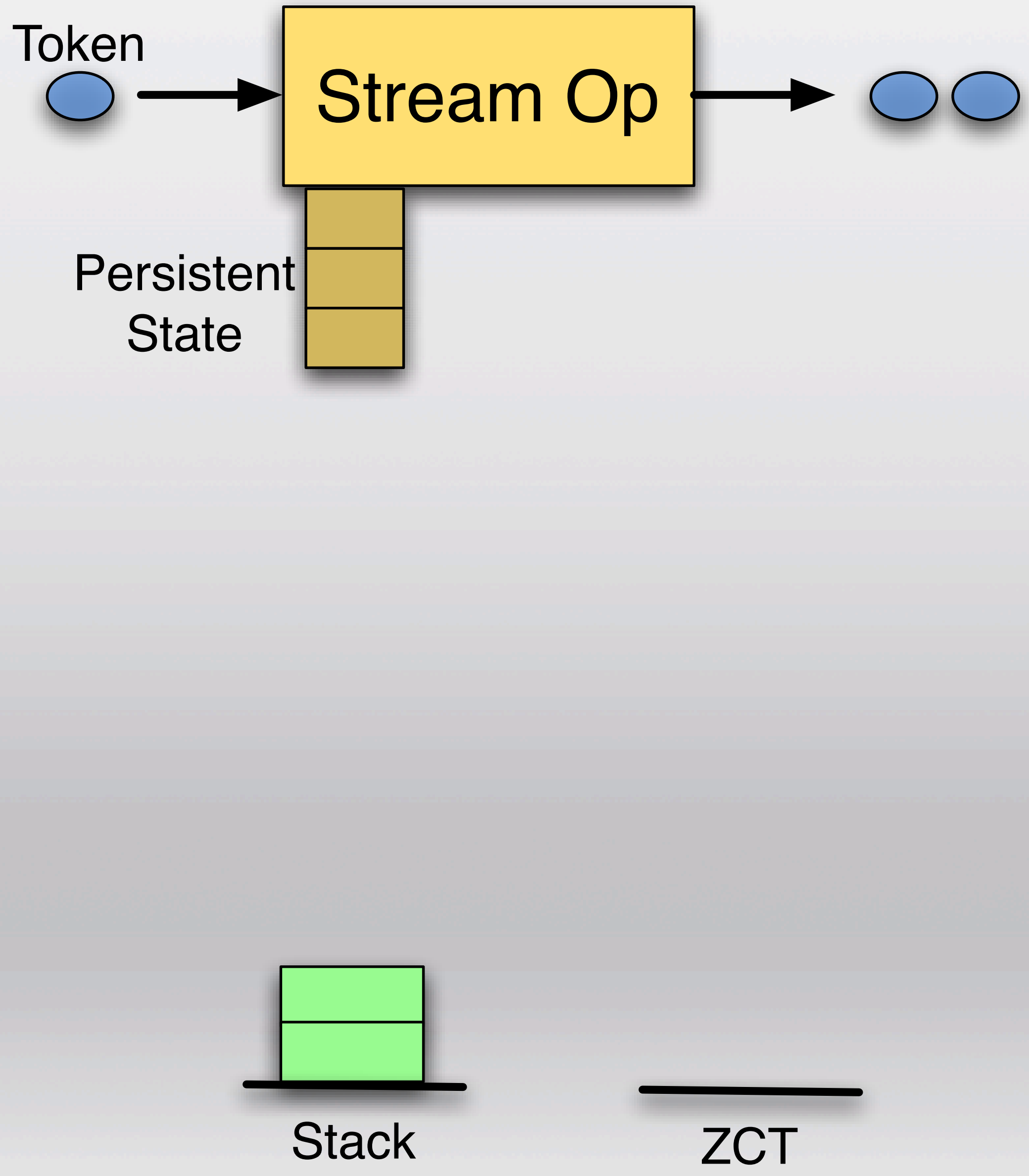
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



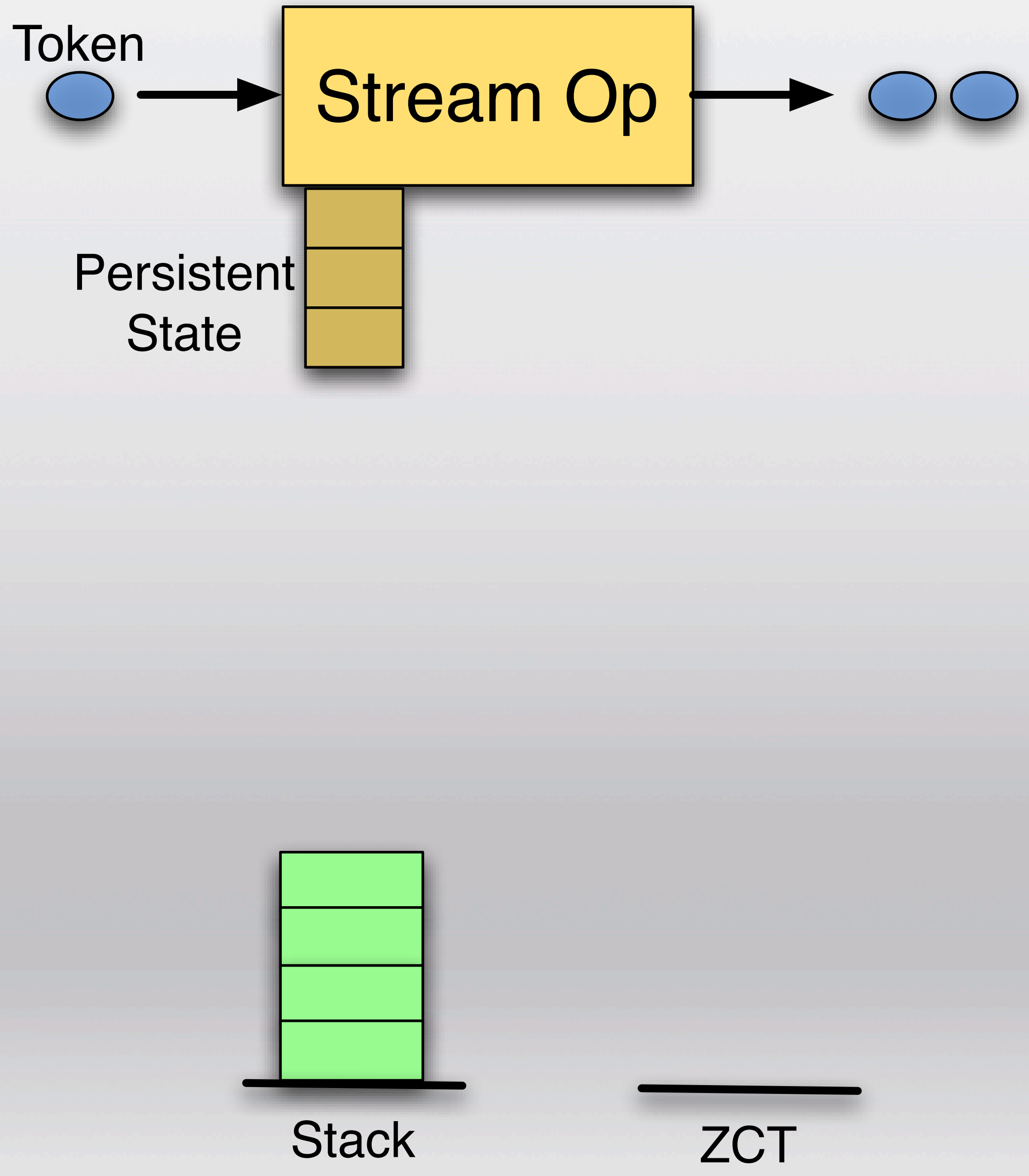
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



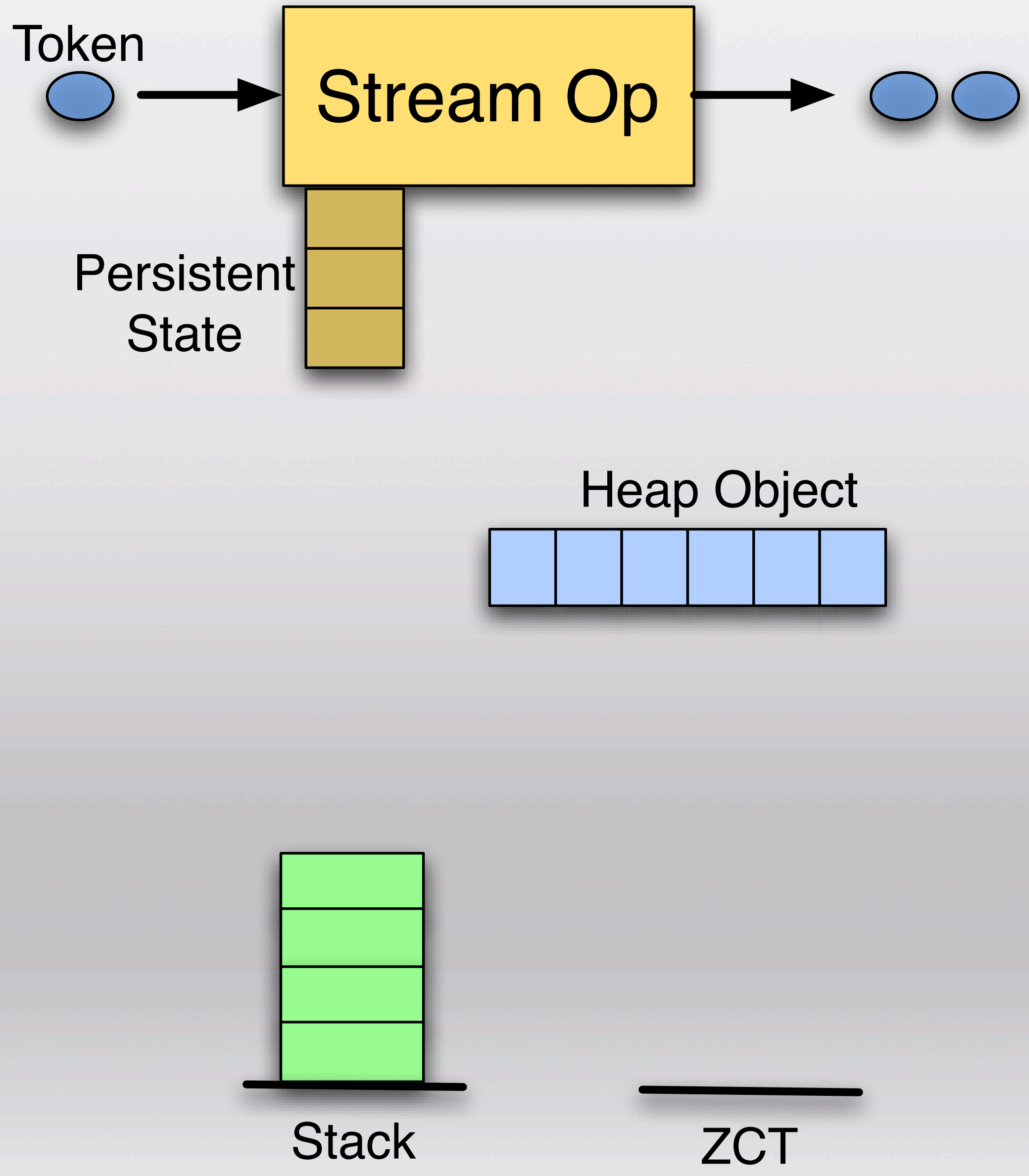
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



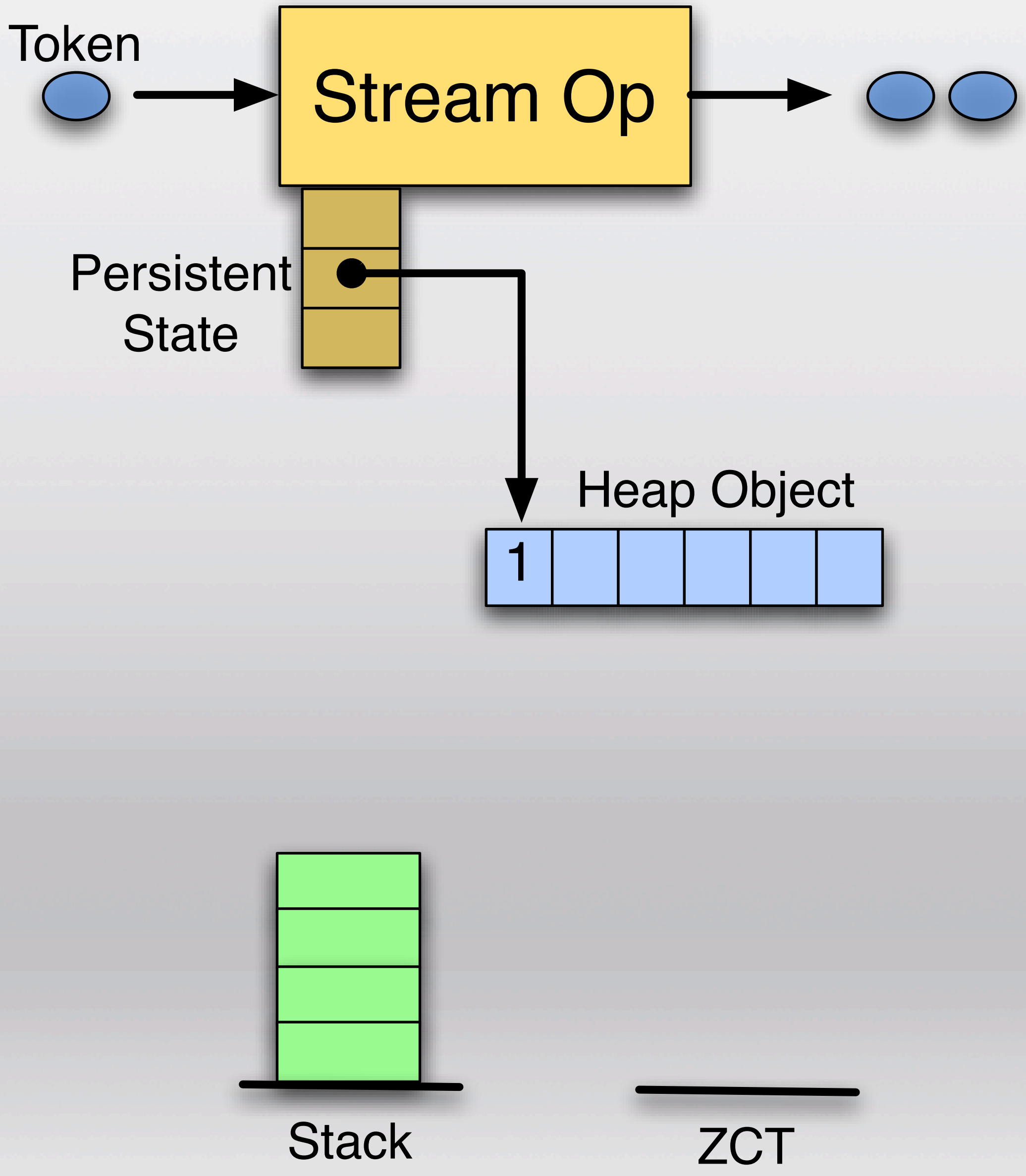
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



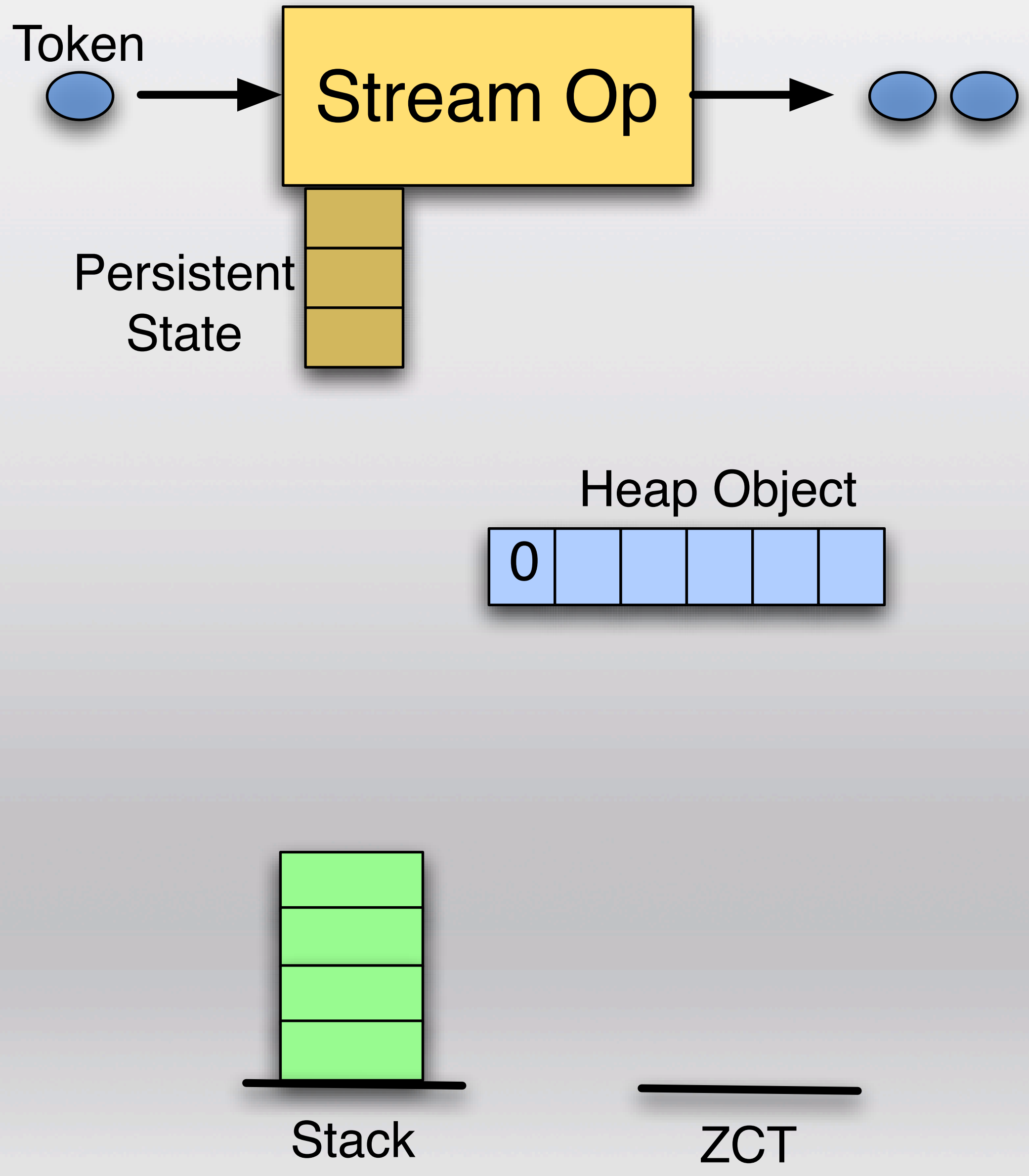
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



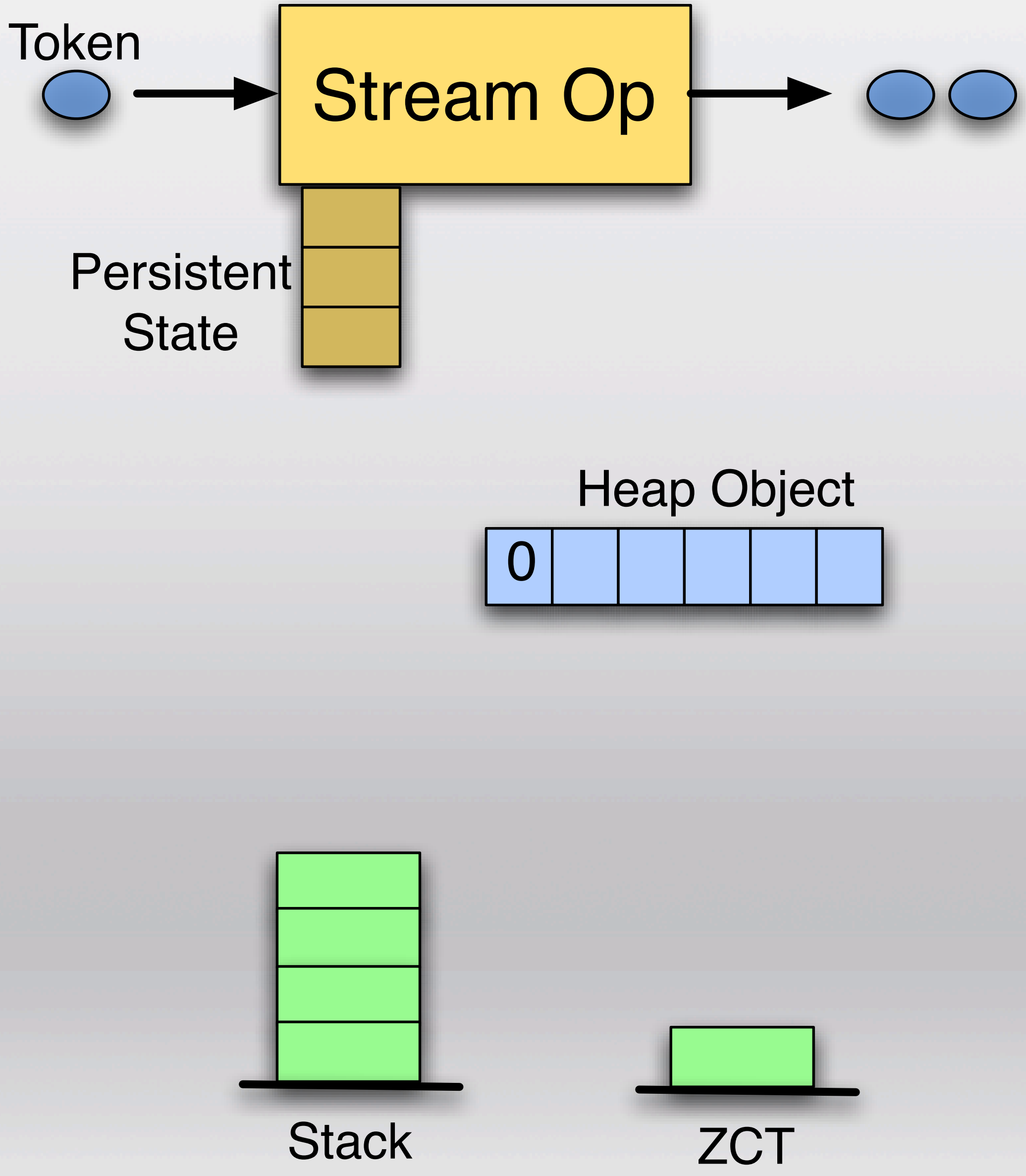
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



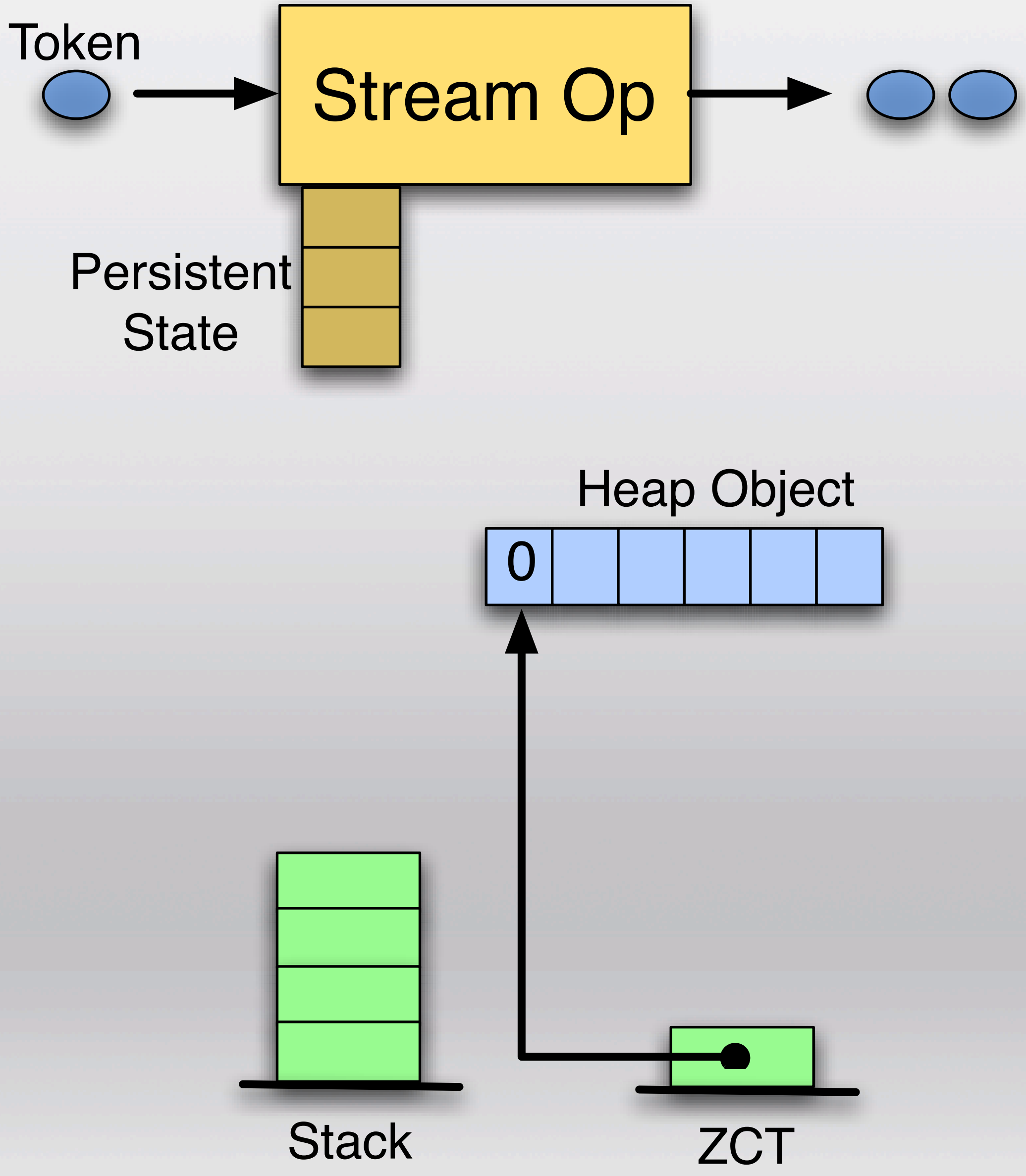
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally



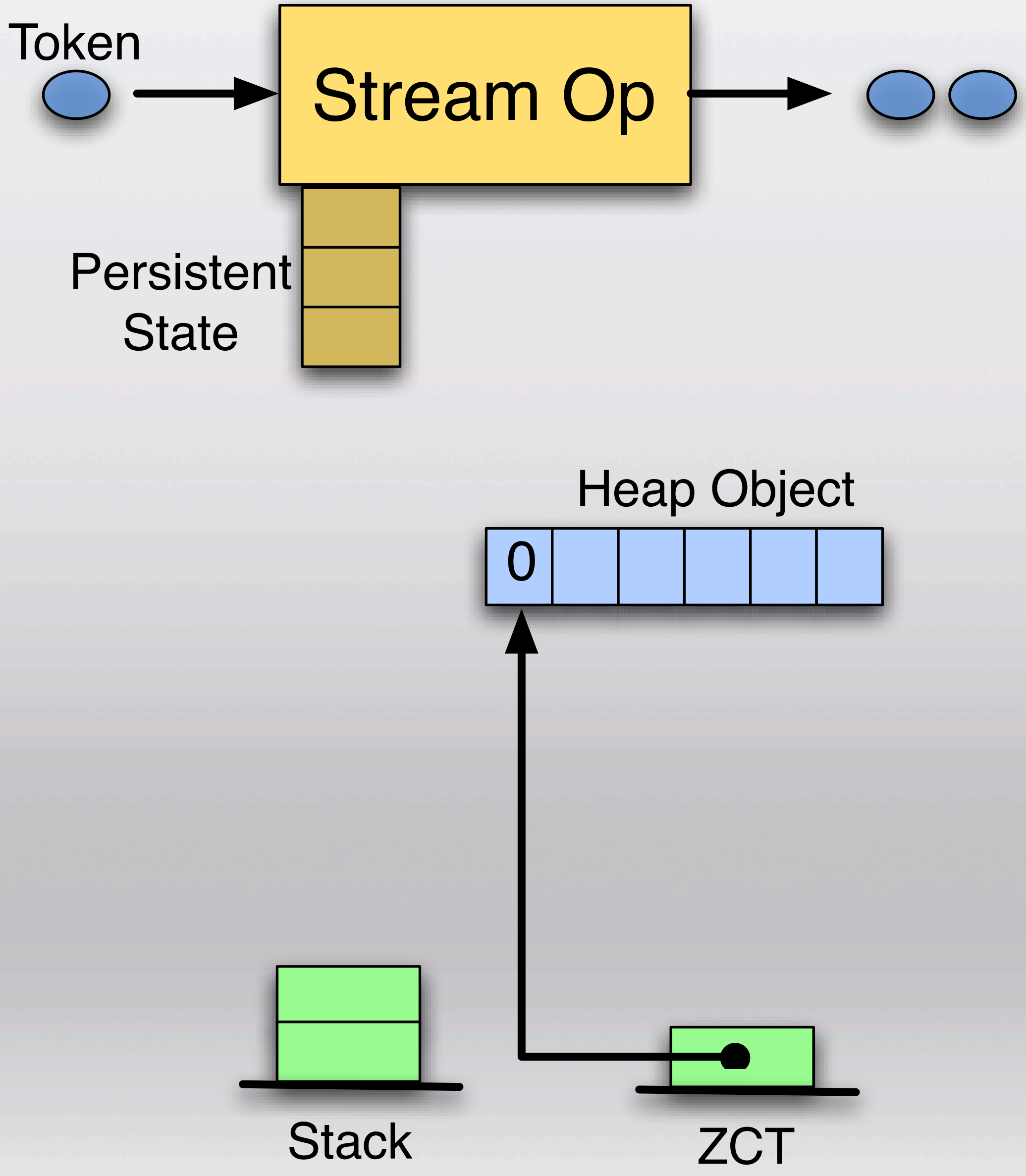
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally



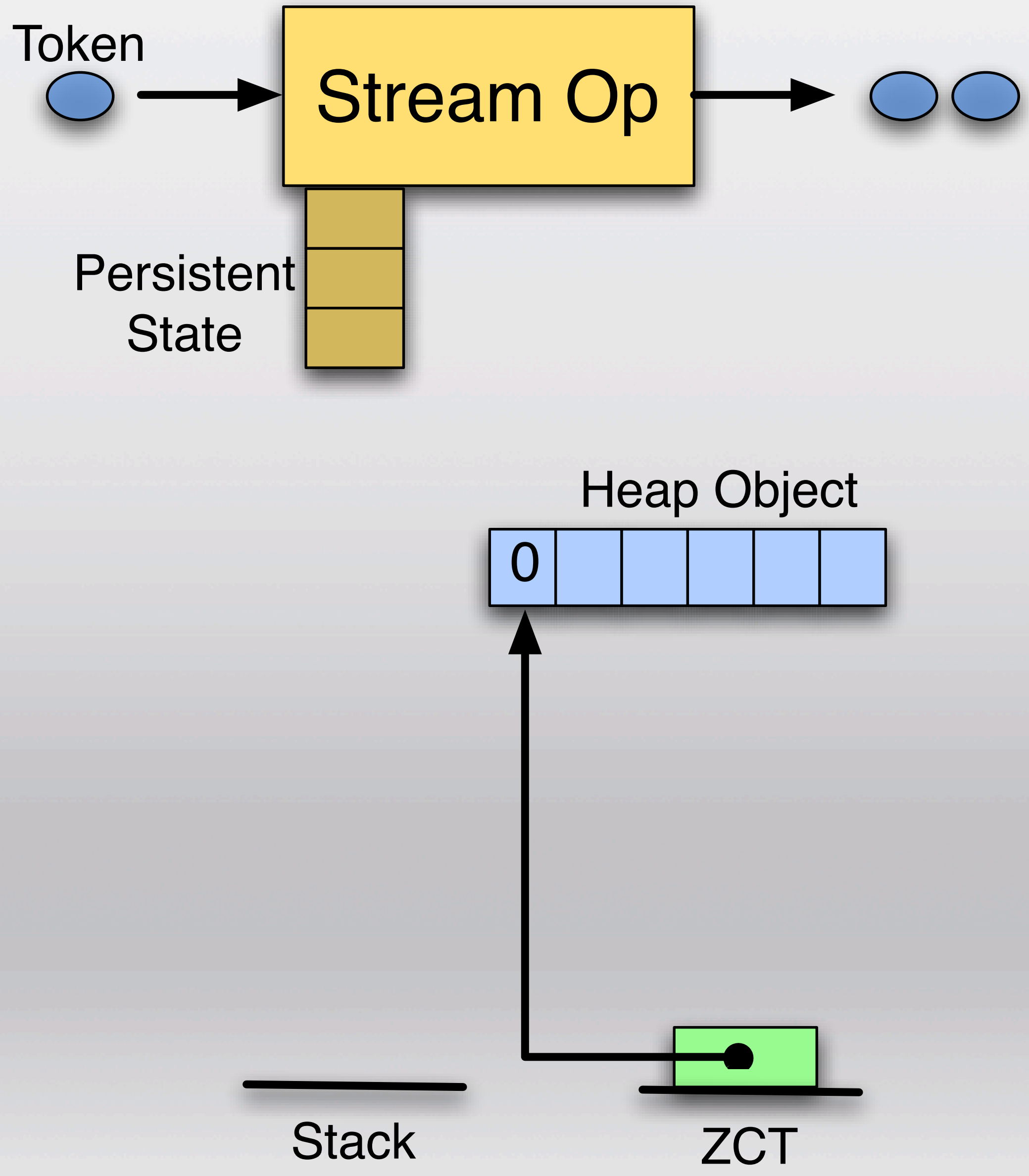
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



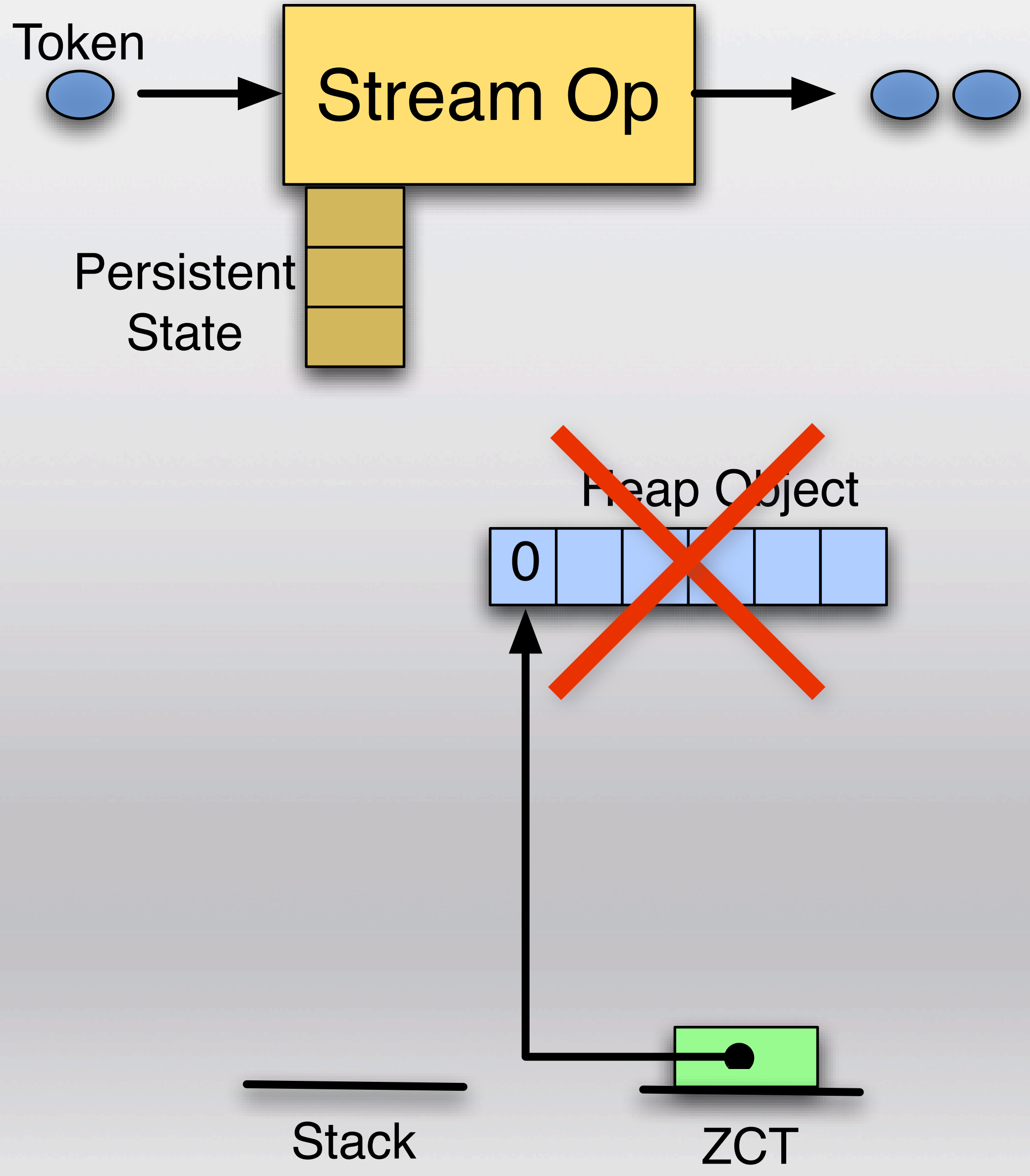
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



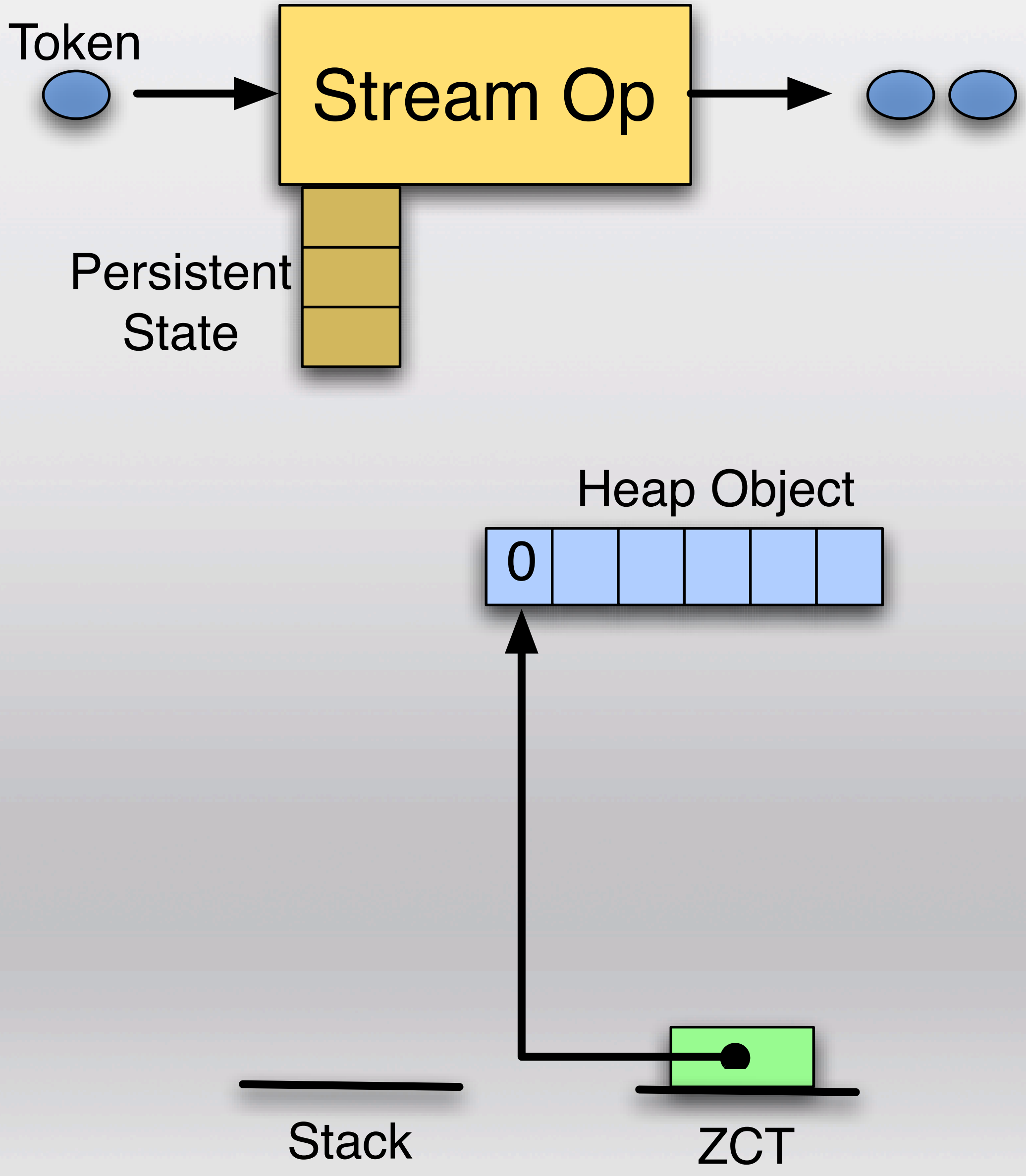
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



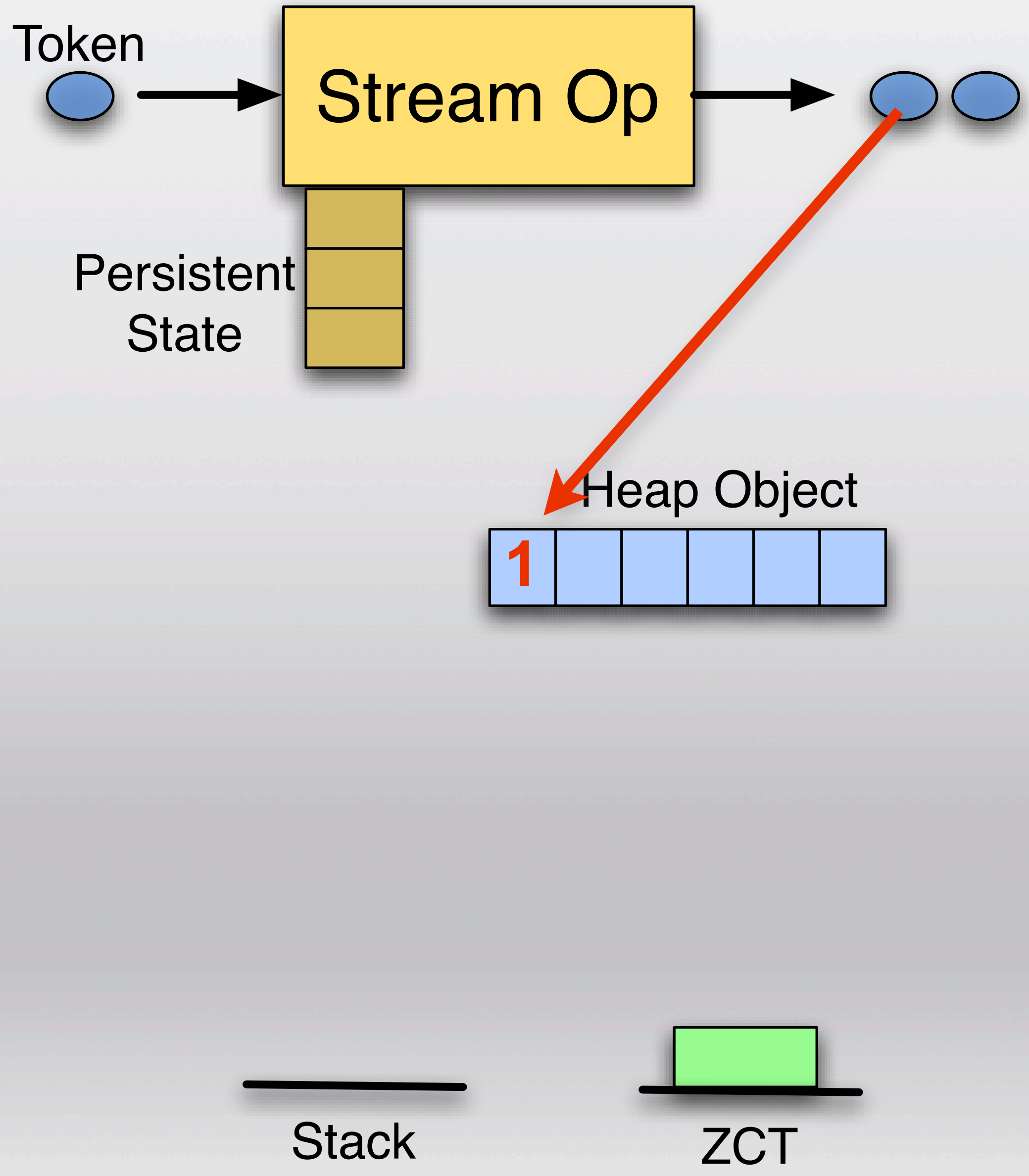
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



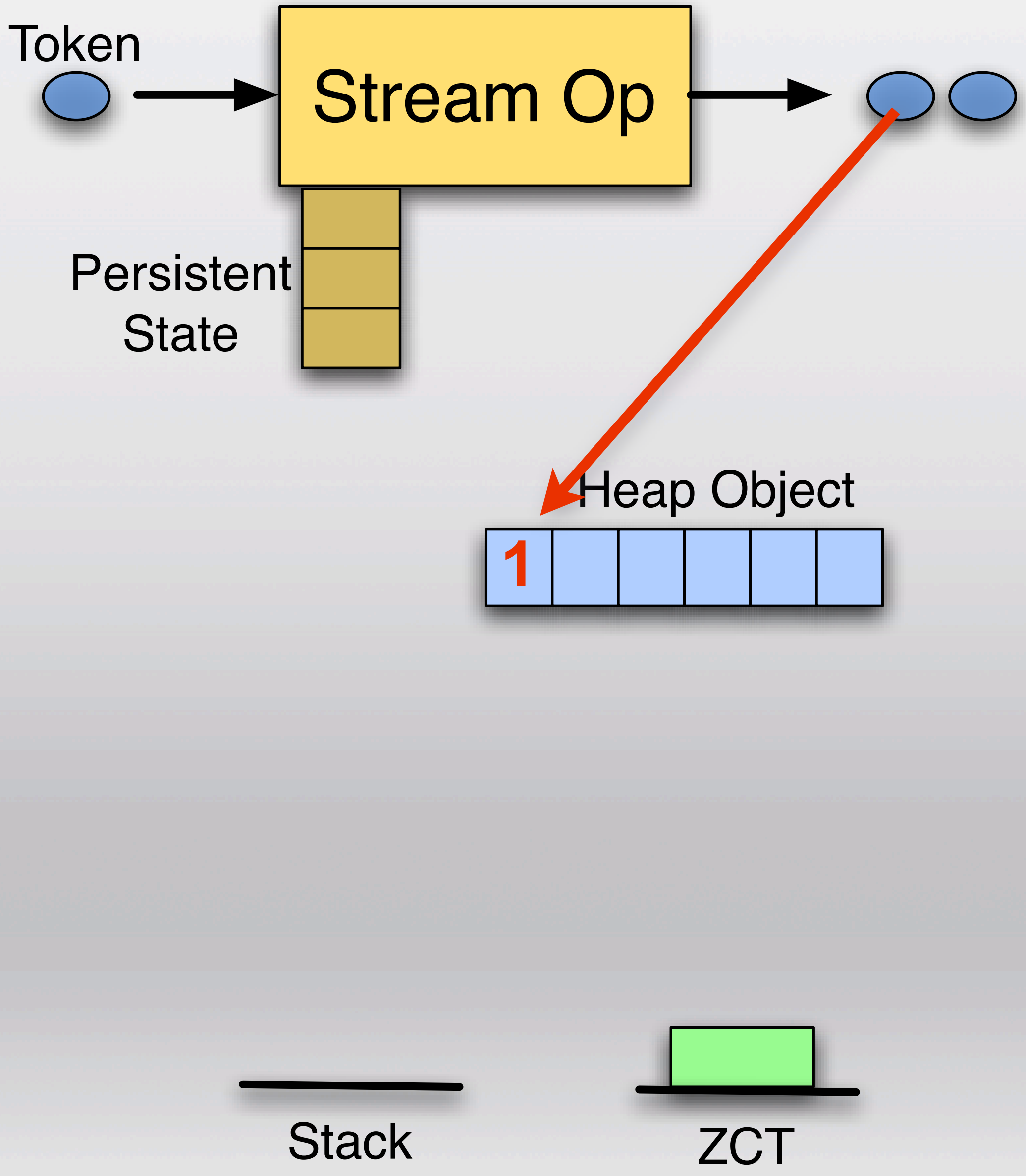
Current work: efficient GC

- Delayed RC: don't track stack references
  - But then you need to trace occasionally



Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
- We can manage separate heaps efficiently
  - Therefore we can collect at the end of an operator’s execution



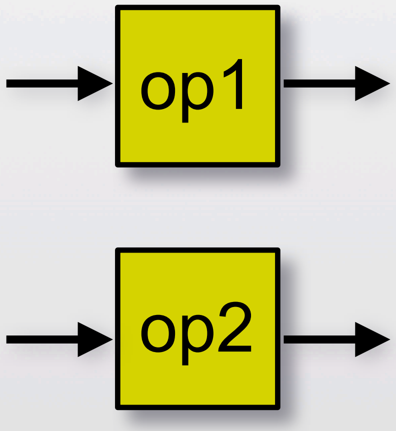
Task, Data, and Pipeline Parallelism

(Intra-machine parallelism)

Task, Data, and Pipeline Parallelism

(Intra-machine parallelism)

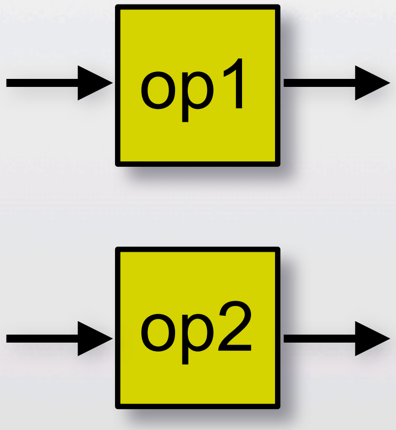
Task



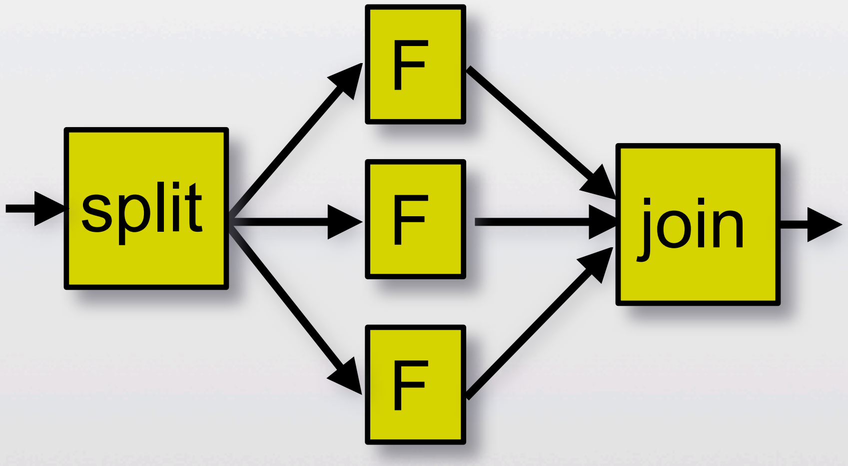
Task, Data, and Pipeline Parallelism

(Intra-machine parallelism)

Task



Data

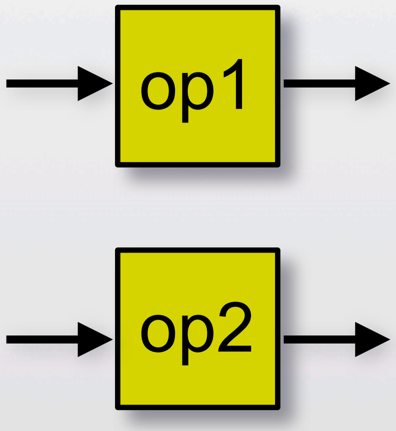


`map(F,strm)`

Task, Data, and Pipeline Parallelism

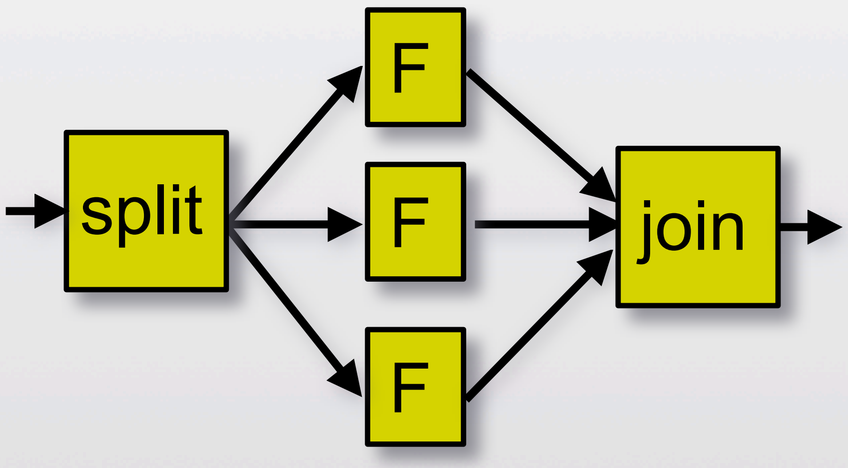
(Intra-machine parallelism)

Task


op1

op2

Data


split

F

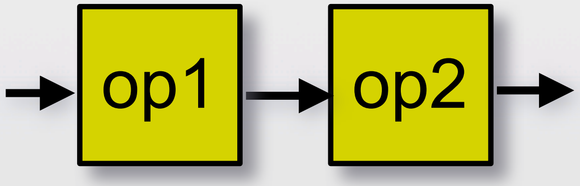
F

F

join

`map(F,strm)`

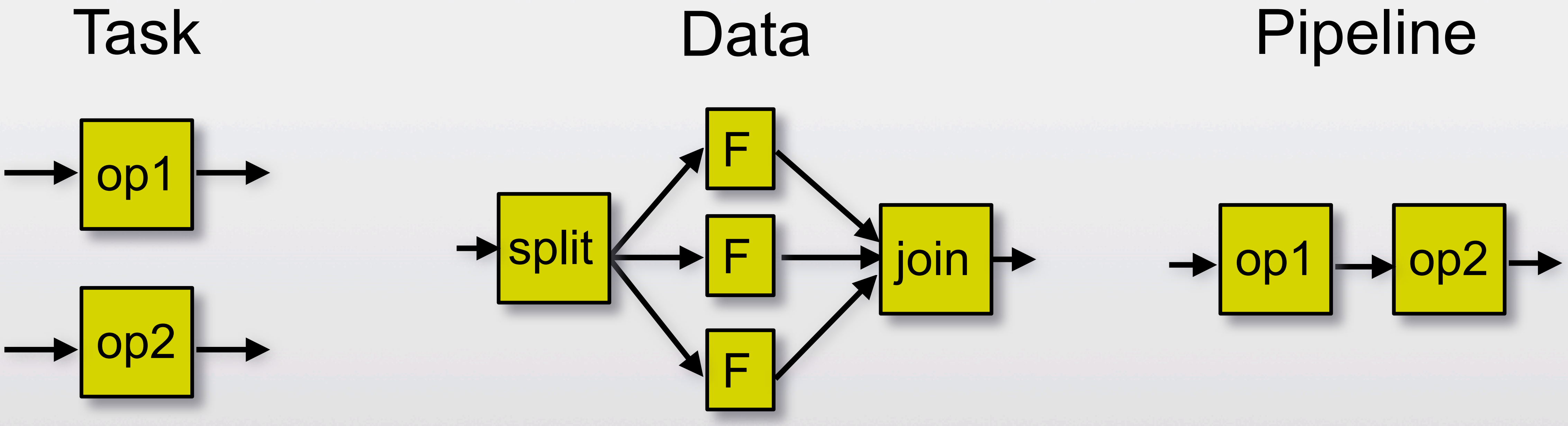
Pipeline


op1

op2

Task, Data, and Pipeline Parallelism

(Intra-machine parallelism)



`map(F,strm)`

- See StreamIT for good work on optimizing parallelism
  - You need information on *data rates*
- We use PROFILING based on *sample data*
  - Helps with other optimizations: e.g. data representation transforms

Building Stream Abstractions

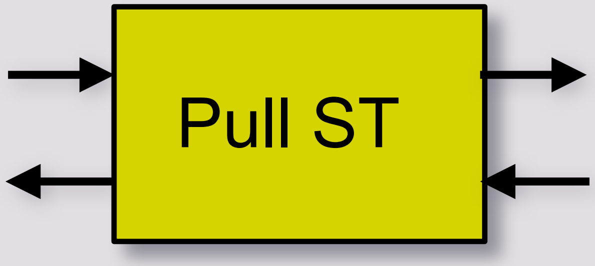
- Further abstract the representation of a stream, enable changing the “glue”.

Building Stream Abstractions

- Further abstract the representation of a stream, enable changing the “glue”.
  - Pull based streams.
    - type PullStrm t = Stream () → Stream t;
  - Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
    - type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)
  - Self-marshaling stream operators

Building Stream Abstractions

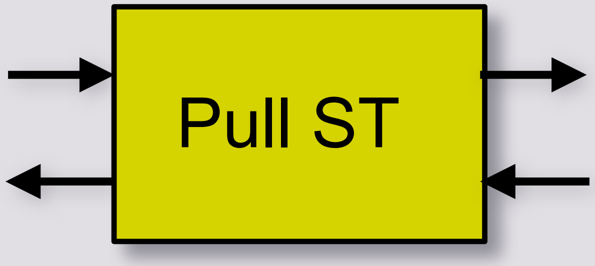
- Further abstract the representation of a stream, enable changing the “glue”.
  - Pull based streams.
    - type PullStrm t = Stream () → Stream t;



  - Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
    - type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)
  - Self-marshaling stream operators

Building Stream Abstractions

- Further abstract the representation of a stream, enable changing the “glue”.
  - Pull based streams.
    - type PullStrm t = Stream () → Stream t;



  - Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
    - type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)

  - Self-marshaling stream operators

Conclusions

End.

wavescope.csail.mit.edu