

# Ouros

## A Dataflow-Driven Processor for Lazy Functional Programming Languages

---

Yukang Xie, Craig Ramsay, Robert Stewart, and Hans-Wolfgang Loidl

Heriot-Watt University

September, 2026

(Lazy) FP language is a thing

---

# (Lazy) FP language is a thing

---

- First-class functions
- Immutability
- Laziness

# (Lazy) FP language is a thing

---

- First-class functions
- Immutability
- Laziness
- Good expressiveness
- Good maintainability
- Easy to verify

# (Lazy) FP language is a thing

---

- First-class functions
- Immutability
- Laziness
- Good expressiveness
- Good maintainability
- Easy to verify

Specialised processors for FP has a long history (*but haven't been successful so far*).

Can we reinvestigate this idea in a modern context?

# Evaluating lazy FP: Graph reduction

---

laziness (call-by-need) =  
call-by-name + sharing

# Evaluating lazy FP: Graph reduction

---

laziness (call-by-need) =  
call-by-name + sharing

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

# Evaluating lazy FP: Graph reduction

---

laziness (call-by-need) =  
call-by-name + sharing

```
fib n = if n <= 1  
       then 1  
       else fib (n - 2) + fib (n - 1)
```

`fib (3 * 3)`

# Evaluating lazy FP: Graph reduction

---

laziness (call-by-need) =  
call-by-name + sharing

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

fib (3 \* 3)



```
if (3 * 3) <= 1
then 1
else fib (• - 2) + fib (• - 1)
```

# Evaluating lazy FP: Graph reduction

---

laziness (call-by-need) =  
call-by-name + sharing

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

fib (3 \* 3)



```
if (3 * 3) <= 1
then 1
else fib (• - 2) + fib (• - 1)
```



```
if 9 <= 1
then 1
else fib (• - 2) + fib (• - 1)
```

# Evaluating lazy FP: Graph reduction

laziness (call-by-need) =  
call-by-name + sharing

```
fib n = if n <= 1  
      then 1  
      else fib (n - 2) + fib (n - 1)
```

fib (3 \* 3)



```
if (3 * 3) <= 1  
then 1  
else fib (• - 2) + fib (• - 1)
```



```
if 9 <= 1  
then 1  
else fib (• - 2) + fib (• - 1)
```



9  
fib (• - 2) + fib (• - 1)

# Combinators

---

# Combinators

---

- Combinators are functions without free variables in its body

# Combinators

---

- Combinators are functions without free variables in its body

| Combinator | $\lambda$ -form      | Haskell                          |
|------------|----------------------|----------------------------------|
| S          | $\lambda abc.ac(bc)$ | <code>s a b c = a c (b c)</code> |
| K          | $\lambda ab.a$       | <code>k a b = a</code>           |
| I          | $\lambda a.a$        | <code>i a = a</code>             |

# Combinators

---

- Combinators are functions without free variables in its body

| Combinator | $\lambda$ -form      | Haskell                          |
|------------|----------------------|----------------------------------|
| <b>S</b>   | $\lambda abc.ac(bc)$ | <code>s a b c = a c (b c)</code> |
| <b>K</b>   | $\lambda ab.a$       | <code>k a b = a</code>           |
| <b>I</b>   | $\lambda a.a$        | <code>i a = a</code>             |

- User-defined functions can be decomposed into combinators

# Combinators

---

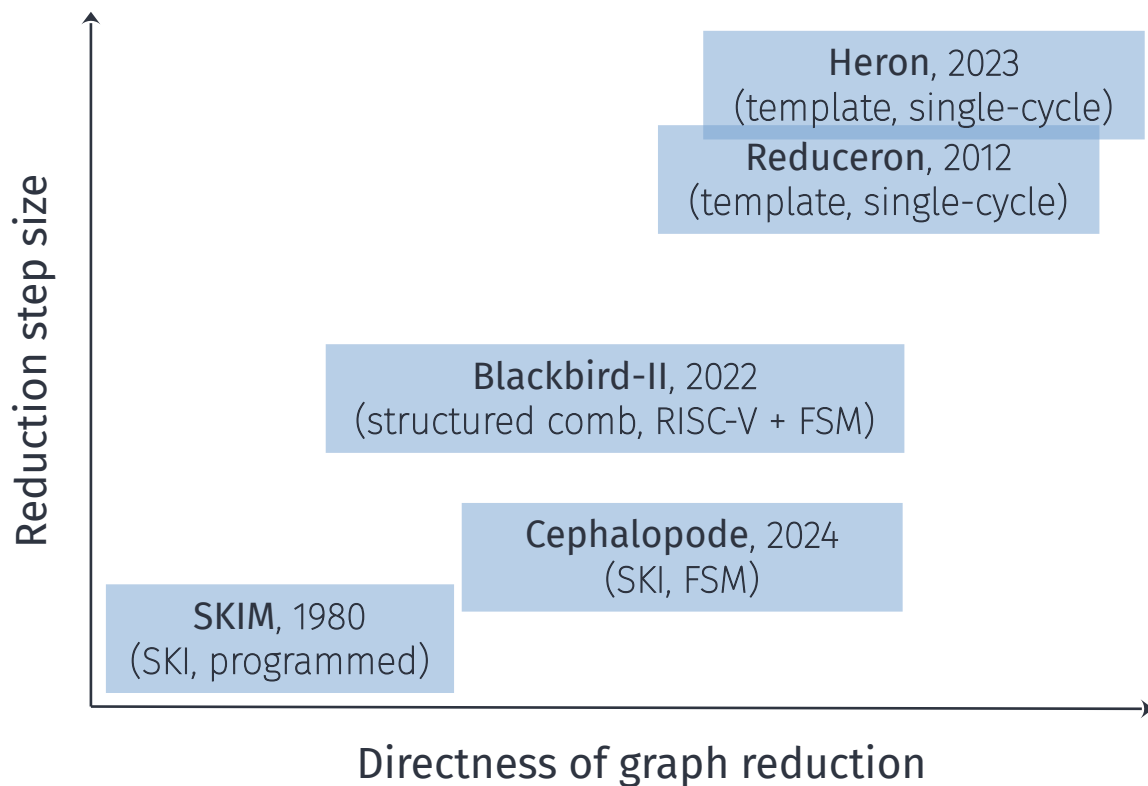
- Combinators are functions without free variables in its body

| Combinator | $\lambda$ -form      | Haskell                          |
|------------|----------------------|----------------------------------|
| <b>S</b>   | $\lambda abc.ac(bc)$ | <code>s a b c = a c (b c)</code> |
| <b>K</b>   | $\lambda ab.a$       | <code>k a b = a</code>           |
| <b>I</b>   | $\lambda a.a$        | <code>i a = a</code>             |

- User-defined functions can be decomposed into combinators
- Friendly for hardware implementation

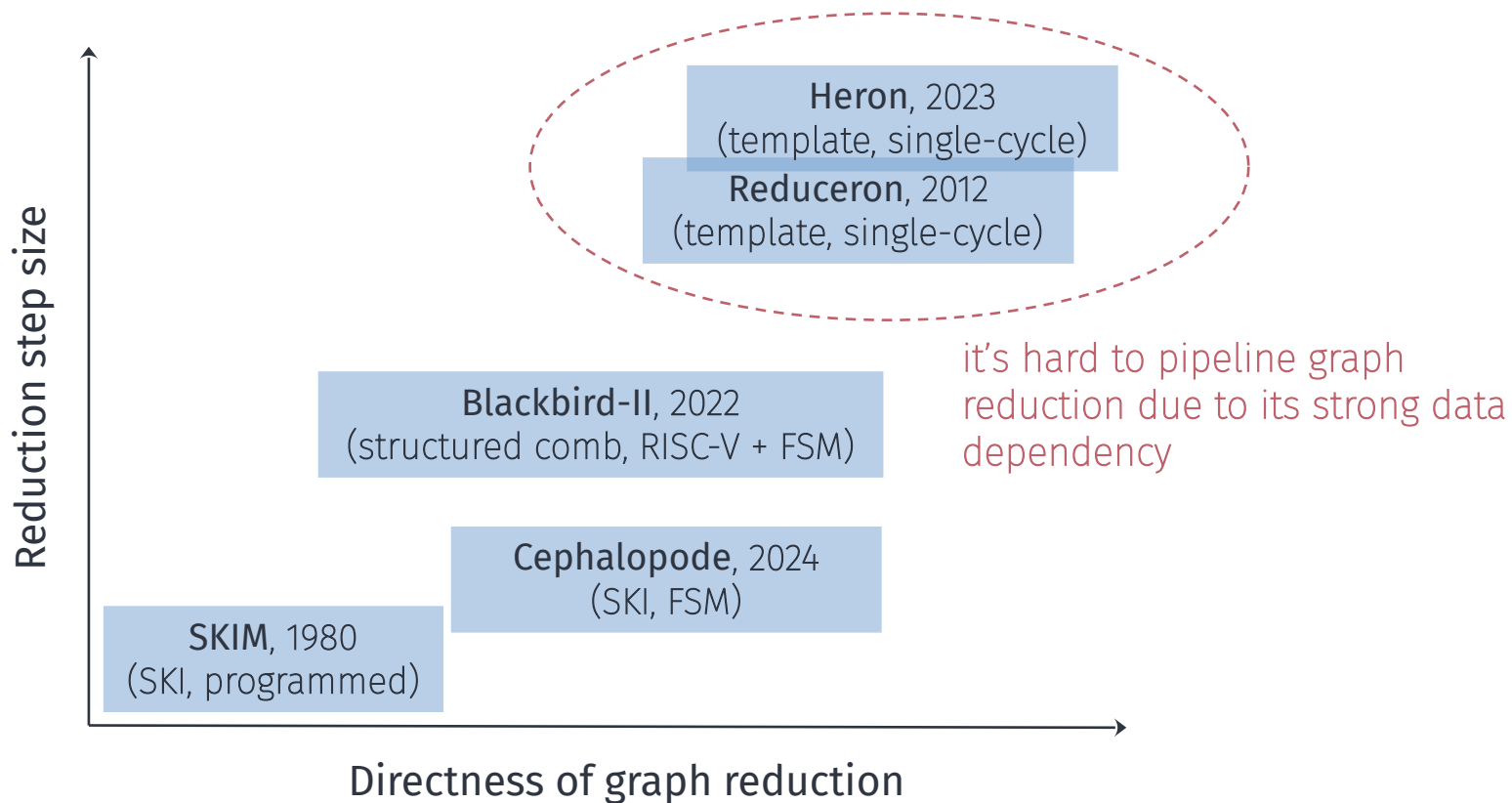
# Specialised processors for FP

---

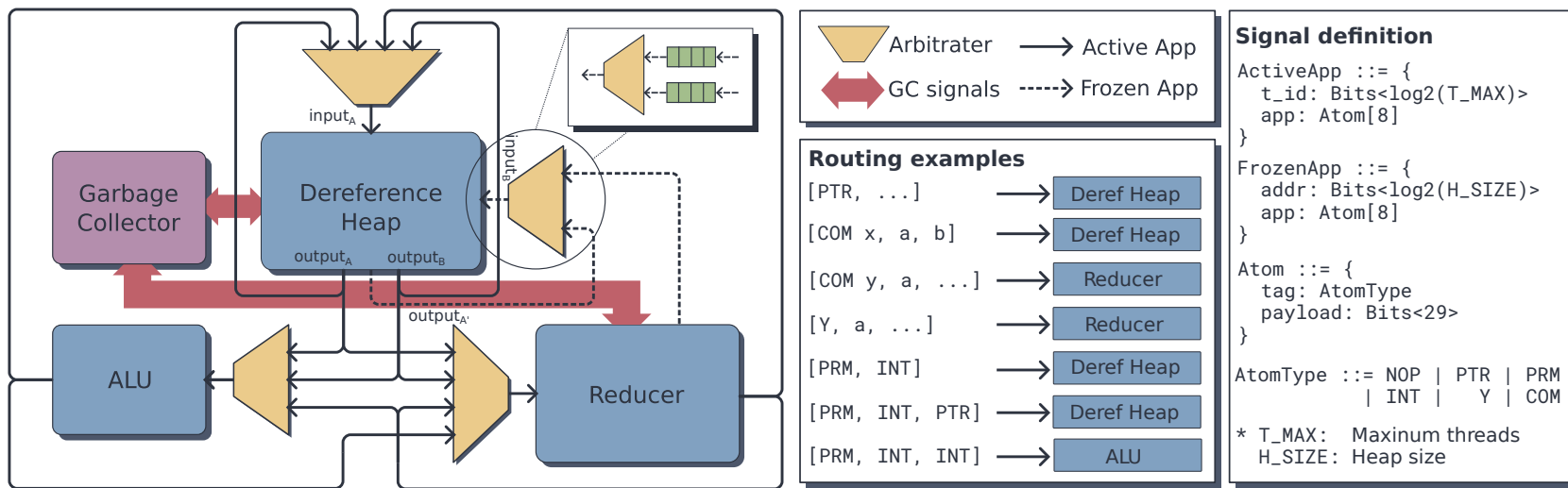


# Specialised processors for FP

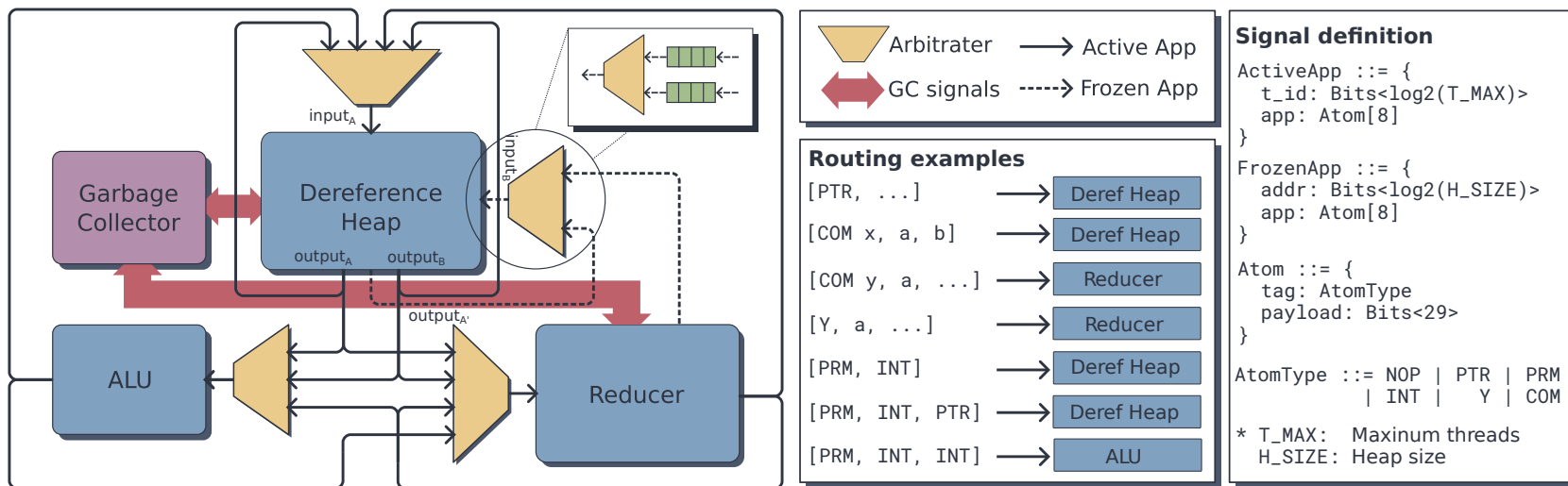
---



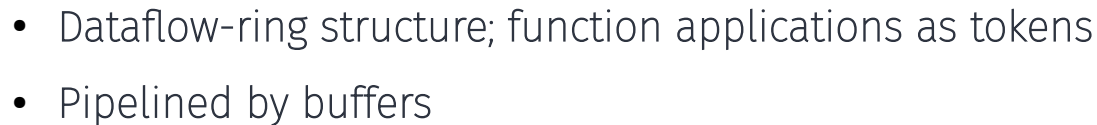
# Ouros: Overview



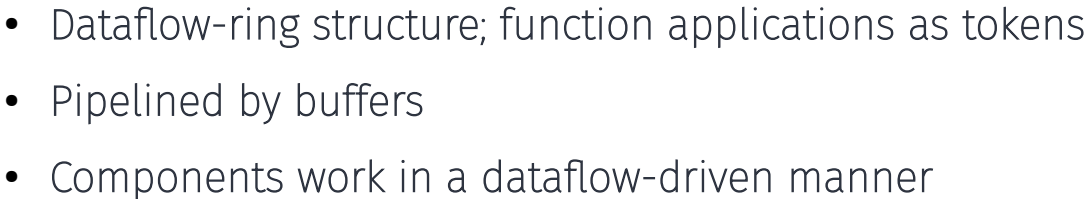
# Ouros: Overview



- Dataflow-ring structure; function applications as tokens

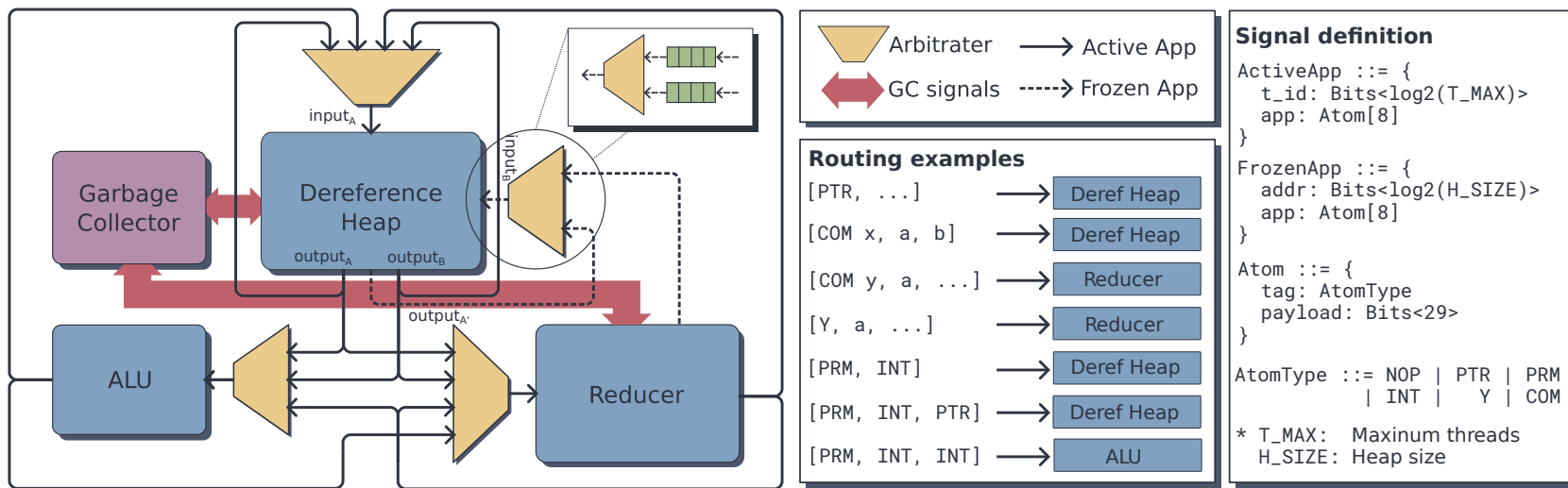


- Dataflow-ring structure; function applications as tokens
- Pipelined by buffers



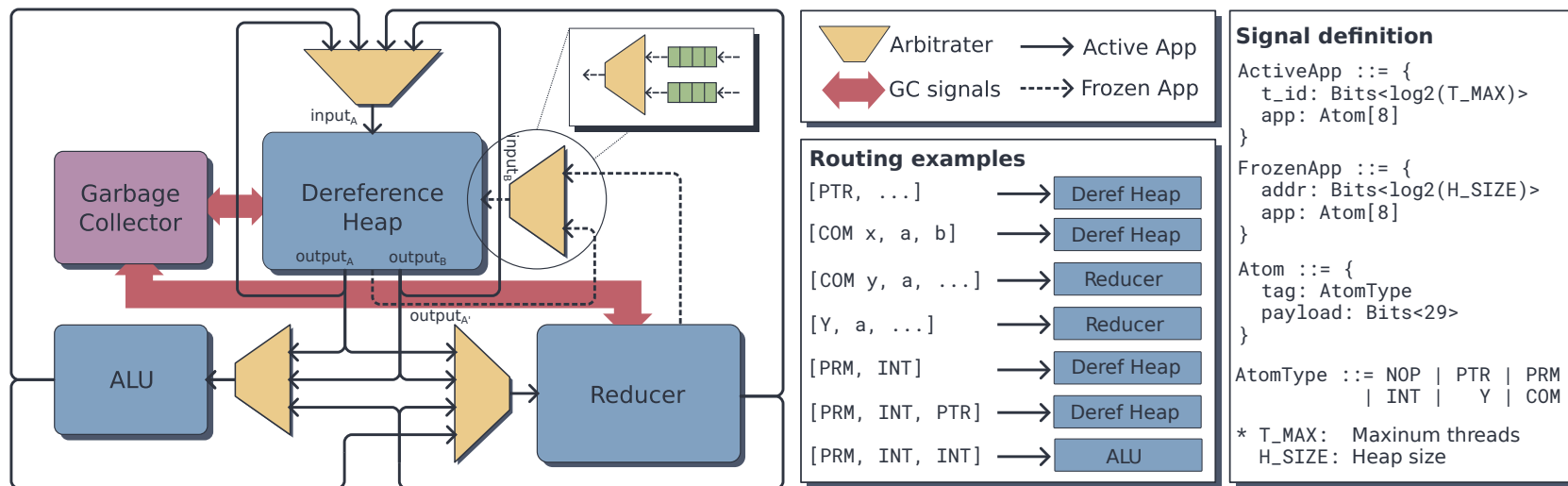
- Dataflow-ring structure; function applications as tokens
- Pipelined by buffers
- Components work in a dataflow-driven manner

# Ouros: Overview

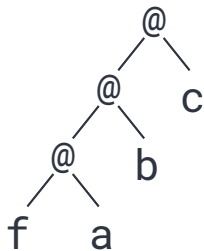


- Dataflow-ring structure; function applications as tokens
- Pipelined by buffers
- Components work in a dataflow-driven manner
- Execution guided by expressions' shape

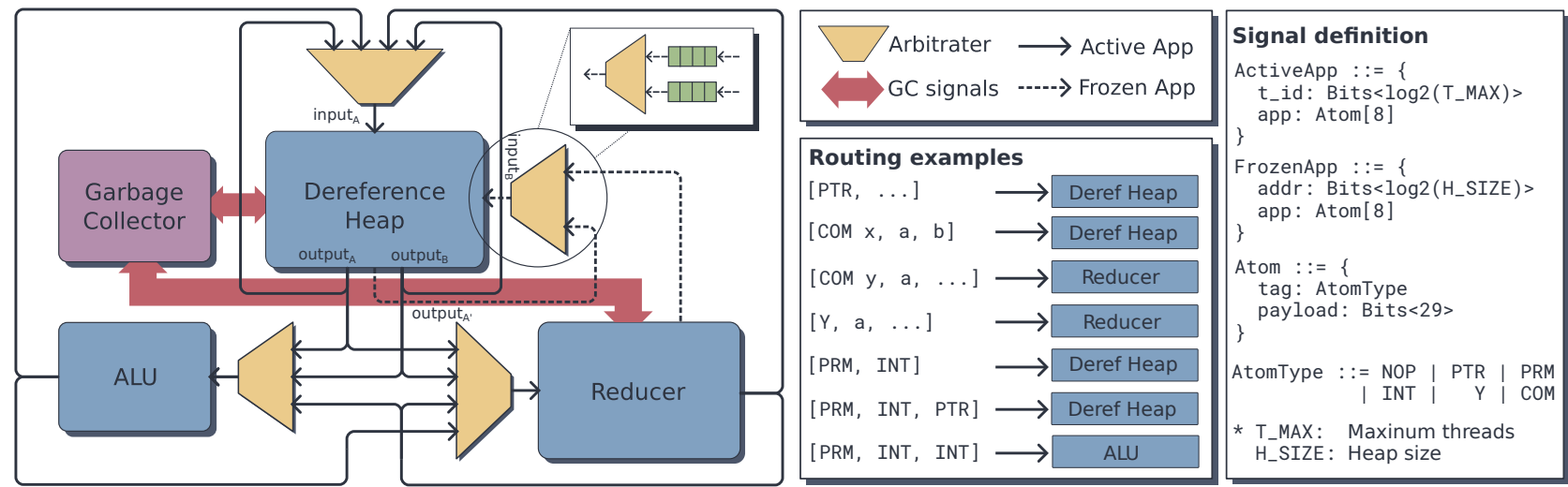
# Ouros: Overview



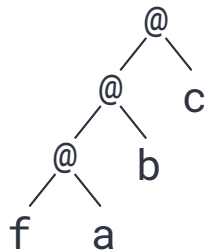
(f a b c in Haskell)



# Ouros: Overview



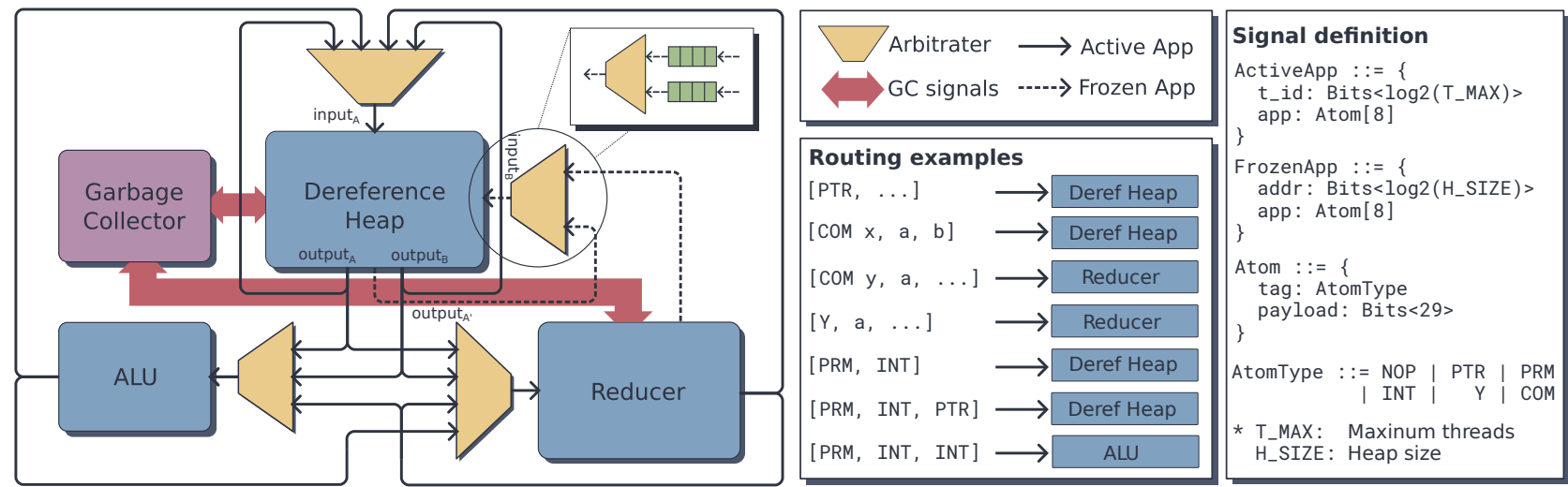
(f a b c in Haskell)



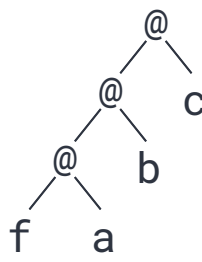
is represented as

[f, a, b, c, NOP, NOP, NOP, NOP]

# Ouros: Overview



(f a b c in Haskell)



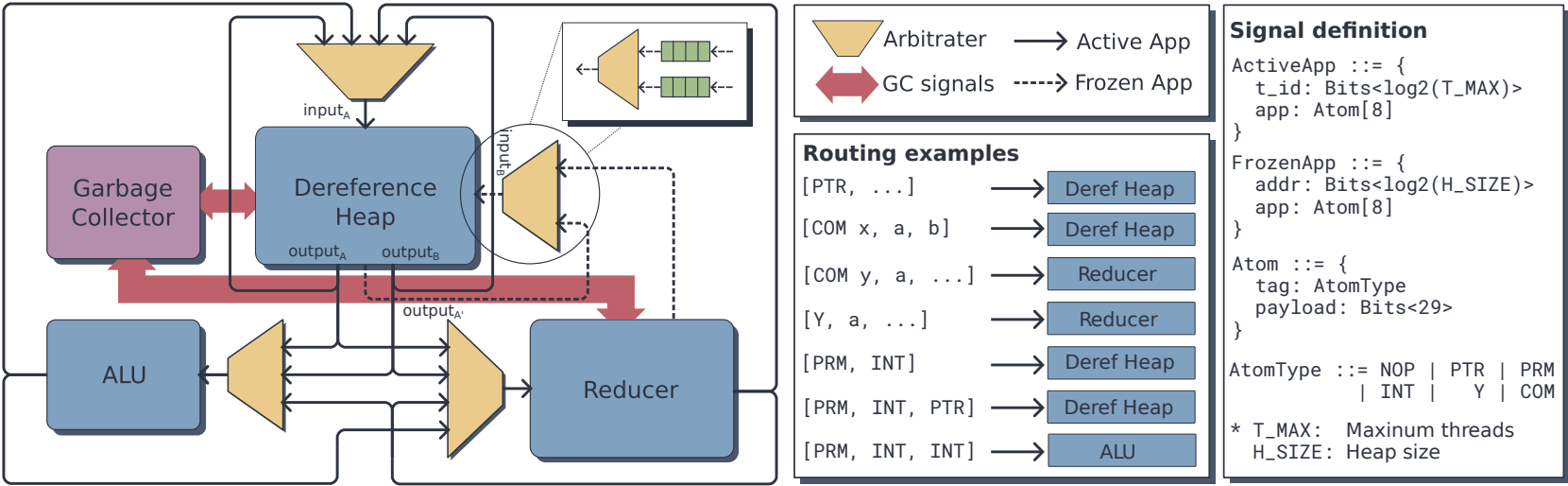
is represented as

[f, a, b, c, NOP, NOP, NOP, NOP]

abbreviated as

[f, a, b, c]

# Ouros: Overview



$1 + (2 - 3)$

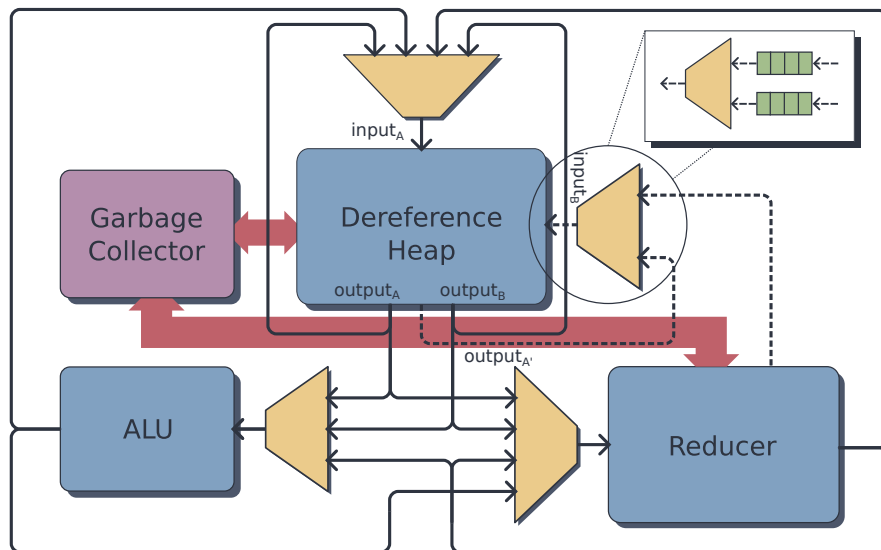
is represented as

[PRM +, INT 1, PTR •]

[PRM -, INT 2, INT 3]

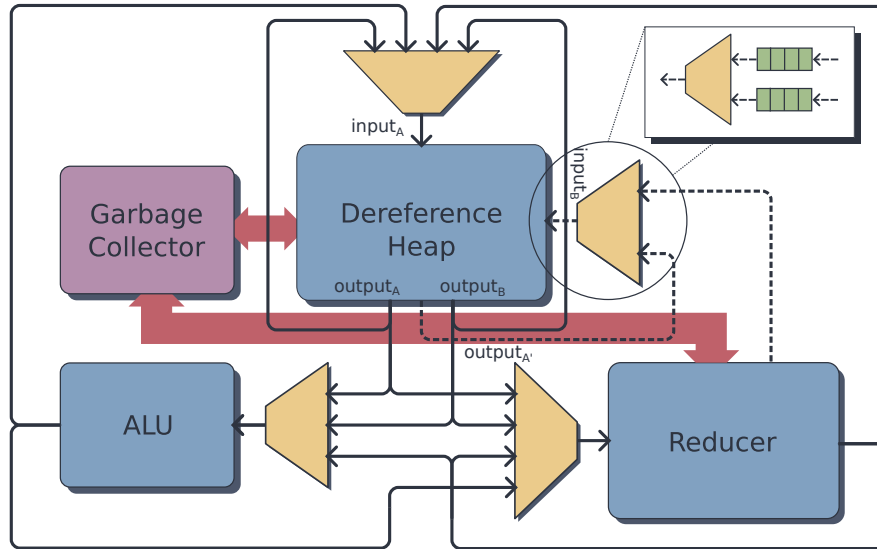
# Latency hiding

---



# Latency hiding

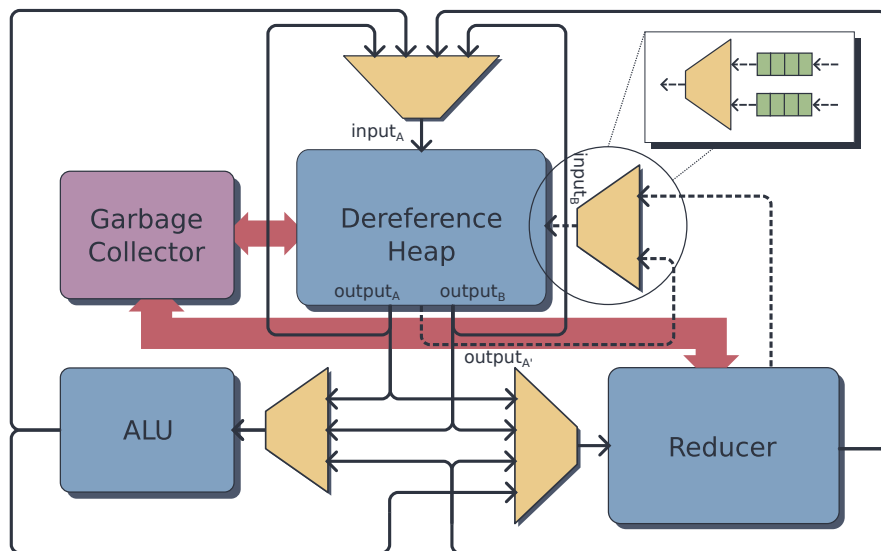
---



- Ouros supports larger combinators that requires **multiple cycles** to reduce

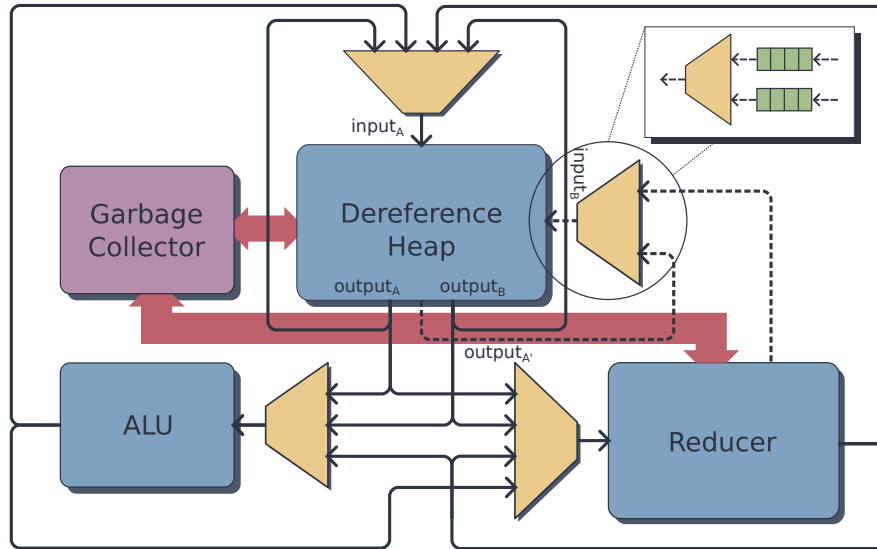
# Latency hiding

---



- Ouros supports larger combinators that requires **multiple cycles** to reduce
- This latency can be partially hidden by dataflow-driven execution

# Latency hiding

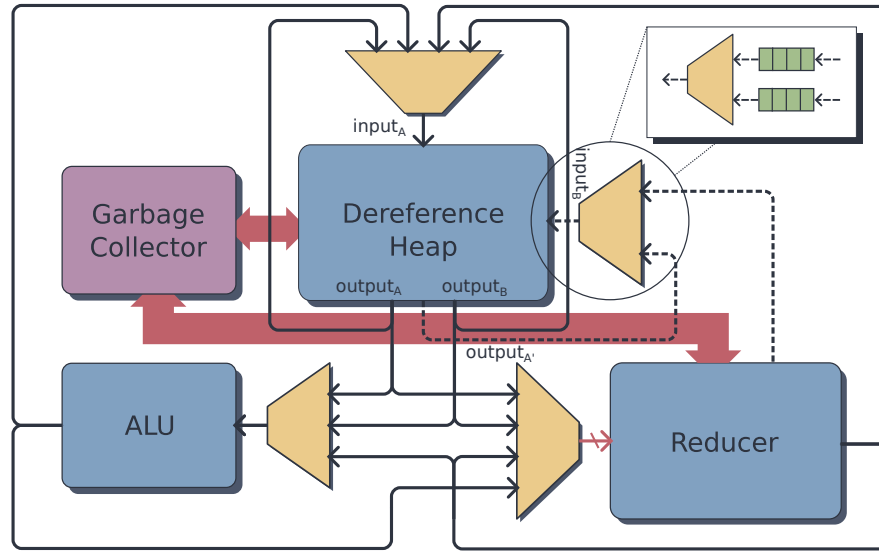


$$f \ x \ y = g \ (h \ x) \ (y + 1)$$

arity of  $g > 2$

- Ouros supports larger combinators that requires **multiple cycles** to reduce
- This latency can be partially hidden by dataflow-driven execution

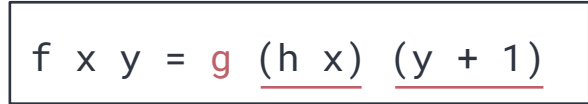
# Latency hiding



$$f \times y = g (h \ x) (y + 1)$$

arity of  $g > 2$

- Ouros supports larger combinators that requires **multiple cycles** to reduce
- This latency can be partially hidden by dataflow-driven execution

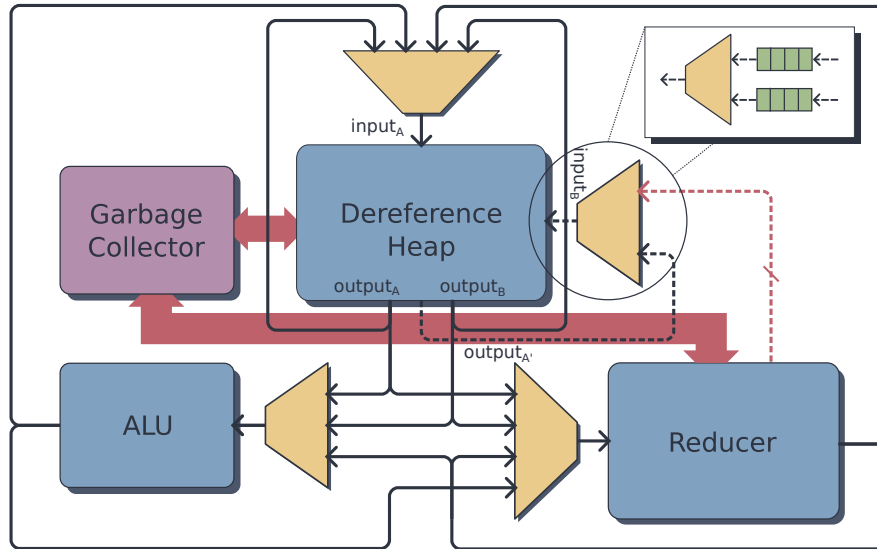


[COM g, PTR p, PTR q]

- Ouros supports larger combinators that requires **multiple cycles** to reduce
- This latency can be partially hidden by dataflow-driven execution



# Latency hiding



Cycle 3

$$f \ x \ y = g \ (h \ x) \ (y + 1)$$

arity of  $g > 2$

[PRM +, INT b, INT 1]

- Ouros supports larger combinators that requires **multiple cycles** to reduce
- This latency can be partially hidden by dataflow-driven execution

# Automatic-multithreading

---

# Automatic-multithreading

---

- In a lazy language, some functions are still strict on their arguments (e.g., arithmetic op)

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

# Automatic-multithreading

---

- In a lazy language, some functions are still strict on their arguments (e.g., arithmetic op)

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

- The purity of FP means it's safe to evaluate these arguments in parallel (implicit parallelism)

# Automatic-multithreading

---

- In a lazy language, some functions are still strict on their arguments (e.g., arithmetic op)

```
fib n = if n <= 1
      then 1
      else fib (n - 2) + fib (n - 1)
```

- The purity of FP means it's safe to evaluate these arguments in parallel (implicit parallelism)
- The granularity problem makes it hard to adopt implicit parallelism in a software runtime; but we can do this in Ouro!

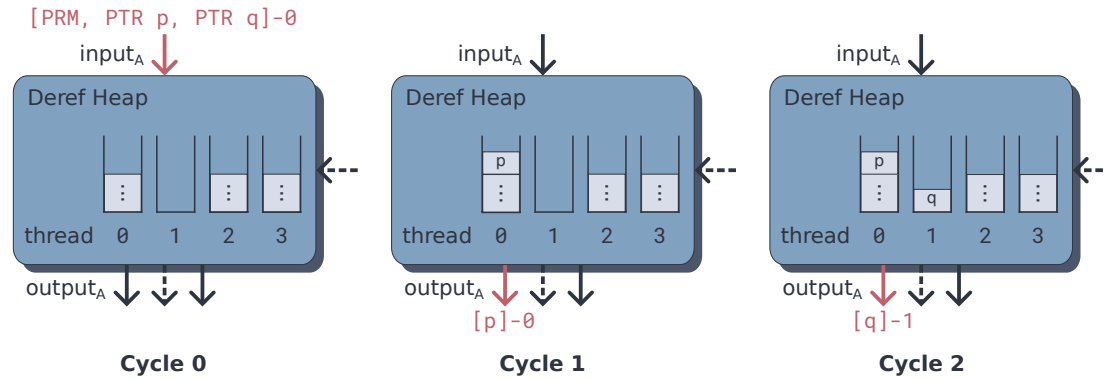
# Automatic-multithreading

---

# Automatic-multithreading

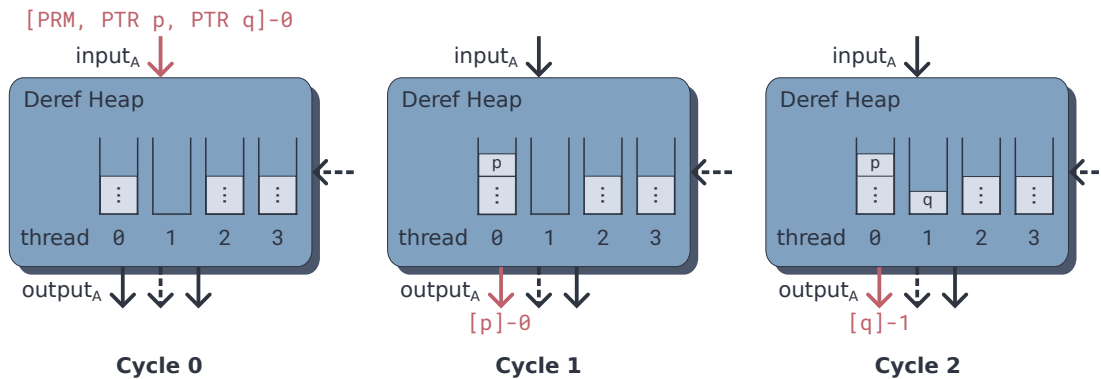
---

Thread creation:

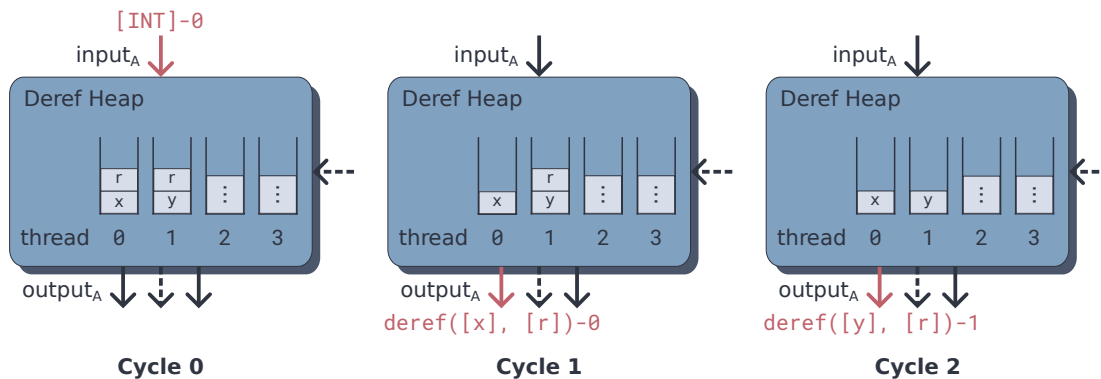


# Automatic-multithreading

Thread creation:



Thread synchronisation:



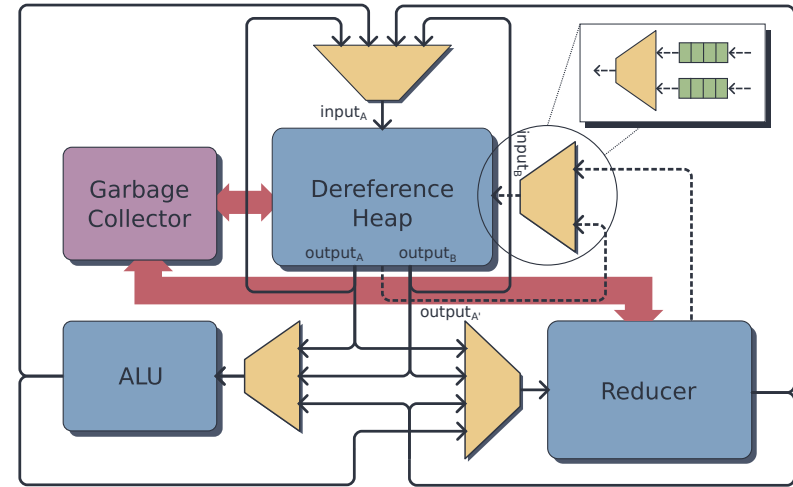
# Automatic-multithreading

---

# Automatic-multithreading

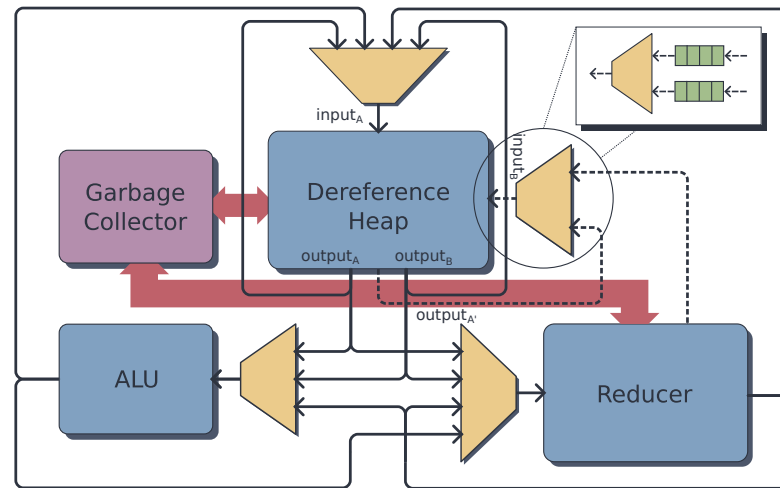
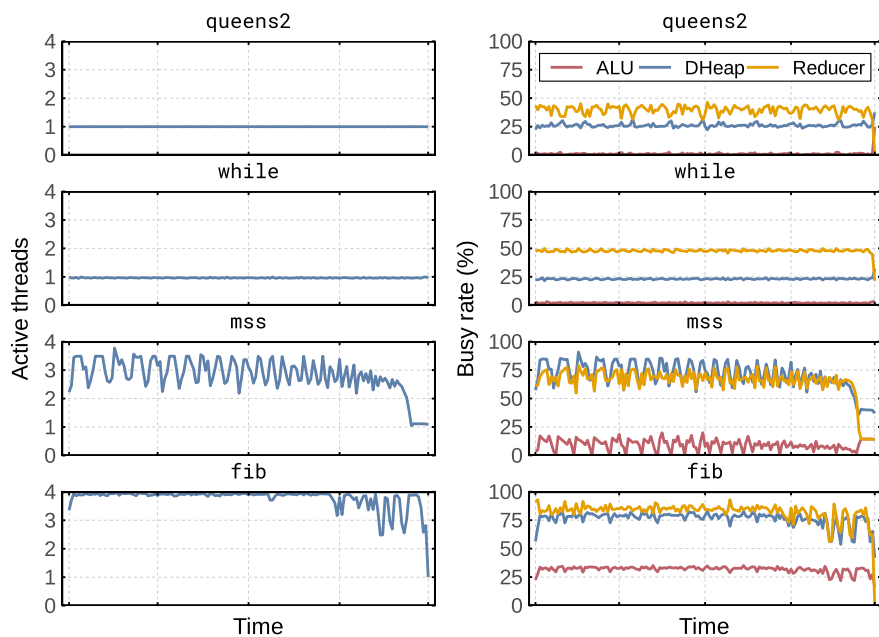
---

Threading behavior varies between different programs.



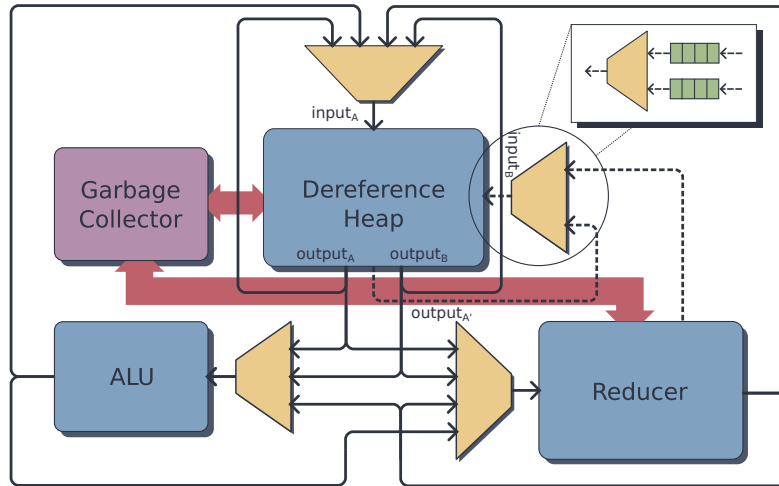
# Automatic-multithreading

Threading behavior varies between different programs.



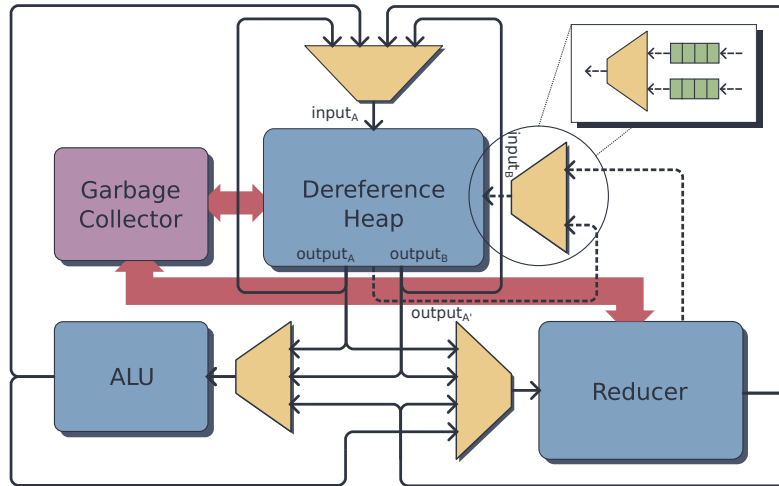
# Garbage collector

---



# Garbage collector

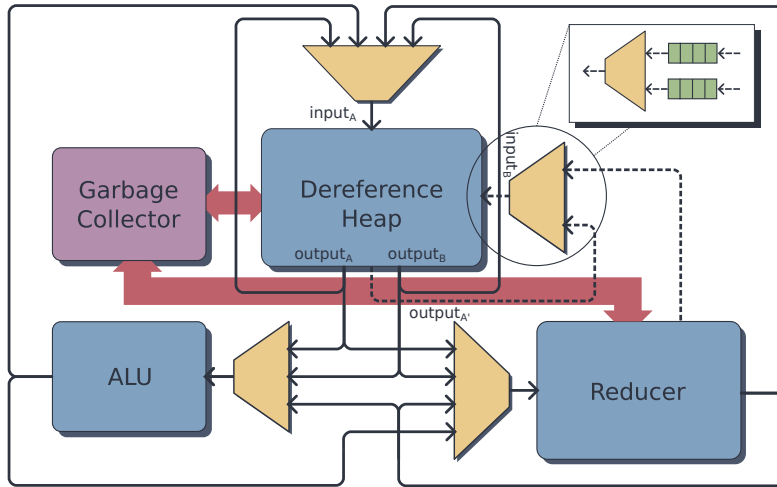
---



- GC is essential for FP runtime

# Garbage collector

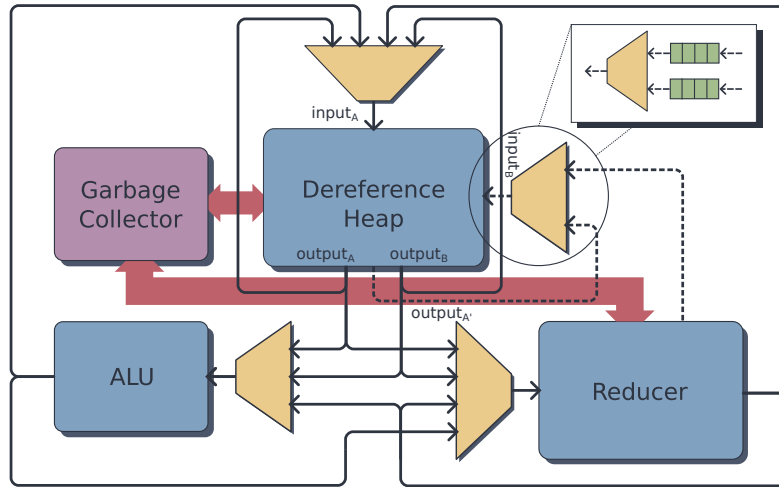
---



- GC is essential for FP runtime
- Maintains a free address list for new alloc

# Garbage collector

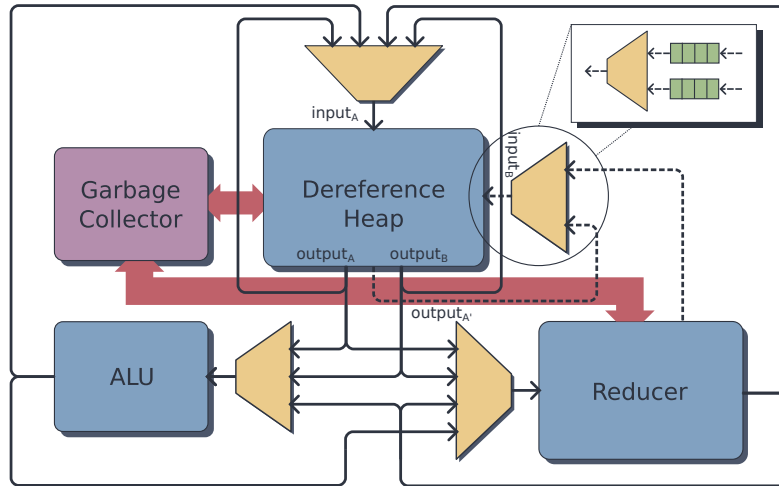
---



- GC is essential for FP runtime
- Maintains a free address list for new alloc
- Works in parallel with the mutator

# Garbage collector

---



- GC is essential for FP runtime
- Maintains a free address list for new alloc
- Works in parallel with the mutator
- The execution only stalls when supply of free address cannot keep up

# GC Algorithm

---

# GC Algorithm

---

- **Algorithm:** a concurrent mark-and-sweep variant

# GC Algorithm

---

- **Algorithm:** a concurrent mark-and-sweep variant
- Expensive write barriers are usually needed in concurrent GC (SATB, incremental update..)

# GC Algorithm

---

- **Algorithm:** a concurrent mark-and-sweep variant
- Expensive write barriers are usually needed in concurrent GC (SATB, incremental update..)
- However, we don't need to account for arbitrary heap mutation in graph reduction!

---

**Algorithm 2:** Mutator Abstraction (Non-deterministic)

---

**Global:** Heap  $H$ , root set  $R$ , free list  $F$ , work list  $W$

**Mutator loop:**

**while** program is running **do**

**Non-deterministically choose from**

**Allocate cell:**

$n \leftarrow \text{pop from } F$

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Insert PTR  $n$  into  $a$

**foreach** PTR  $p \in a$  **do**

                | Optionally insert PTR  $p$  into  $H[n]$

**if** Collector is in mark phase **then**

                | Push  $n$  onto  $W$

**else if**  $n$  in unswept region **then**

                | SETTAG( $n$ , Marked)

**else**

                | SETTAG( $n$ , Unmarked)

**Dereference:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Take alive App  $b$  from  $H$  s.t.  $b \in \text{WHNF}$

**foreach** PTR  $p \in b$  **do**

                | Insert PTR  $p$  into  $a$

**Delete pointer:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

**foreach** PTR  $p \in a$  **do**

                | Optionally remove PTR  $p$  from  $a$

---

# GC Algorithm

---

- **Algorithm:** a concurrent mark-and-sweep variant
- Expensive write barriers are usually needed in concurrent GC (SATB, incremental update..)
- **However, we don't need to account for arbitrary heap mutation in graph reduction!**
- Specifically, a heap cell becomes **immutable** after it reaches WHNF, and may only be dereferenced in WHNF

---

**Algorithm 2:** Mutator Abstraction (Non-deterministic)

---

**Global:** Heap  $H$ , root set  $R$ , free list  $F$ , work list  $W$

**Mutator loop:**

**while** program is running **do**

**Non-deterministically choose from**

**Allocate cell:**

$n \leftarrow \text{pop from } F$

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Insert PTR  $n$  into  $a$

**foreach** PTR  $p \in a$  **do**

                | Optionally insert PTR  $p$  into  $H[n]$

**if** Collector is in mark phase **then**

                | Push  $n$  onto  $W$

**else if**  $n$  in unswept region **then**

                | SETAG( $n$ , Marked)

**else**

                | SETAG( $n$ , Unmarked)

**Dereference:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Take alive App  $b$  from  $H$  s.t.  $b \in \text{WHNF}$

**foreach** PTR  $p \in b$  **do**

                | Insert PTR  $p$  into  $a$

**Delete pointer:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

**foreach** PTR  $p \in a$  **do**

                | Optionally remove PTR  $p$  from  $a$

---

# GC Algorithm

---

- **Algorithm:** a concurrent mark-and-sweep variant
- Expensive write barriers are usually needed in concurrent GC (SATB, incremental update..)
- **However, we don't need to account for arbitrary heap mutation in graph reduction!**
- Specifically, a heap cell becomes **immutable** after it reaches WHNF, and may only be dereferenced in WHNF
- A write barrier is only needed for allocation

---

**Algorithm 2: Mutator Abstraction (Non-deterministic)**

---

**Global:** Heap  $H$ , root set  $R$ , free list  $F$ , work list  $W$

**Mutator loop:**

**while** program is running **do**

**Non-deterministically choose from**

**Allocate cell:**

$n \leftarrow \text{pop from } F$

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Insert PTR  $n$  into  $a$

**foreach** PTR  $p \in a$  **do**

                | Optionally insert PTR  $p$  into  $H[n]$

**if** Collector is in mark phase **then**

 Push  $n$  onto  $W$

**else if**  $n$  in unswept region **then**

                | SETAG( $n$ , Marked)

**else**

                | SETAG( $n$ , Unmarked)

**Dereference:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

            Take alive App  $b$  from  $H$  s.t.  $b \in \text{WHNF}$

**foreach** PTR  $p \in b$  **do**

                | Insert PTR  $p$  into  $a$

**Delete pointer:**

            Take alive App  $a$  from  $H$  s.t.  $a \notin \text{WHNF}$

**foreach** PTR  $p \in a$  **do**

                | Optionally remove PTR  $p$  from  $a$

---

# Verify the algorithm with model checking

---

# Verify the algorithm with model checking

---

- Bounded model checking of the GC algorithm using Kind2

# Verify the algorithm with model checking

---

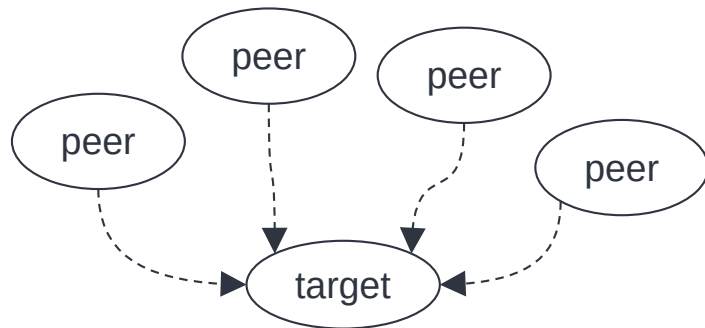
- **Bounded model checking** of the GC algorithm using Kind2
- *Safety*: no live object will be collected
- *Completeness*: all garbage will eventually be collected

# Verify the algorithm with model checking

---

- **Bounded model checking** of the GC algorithm using Kind2
- **Safety**: no live object will be collected
- **Completeness**: all garbage will eventually be collected
- State transition model, allowing arbitrary interleaving of mutator and collector heap modifications
- **Bounds**: Limited to 20 modifications and 4 peer cells
- Verification completed in ~2 hours on my laptop
- *Not a “full” verification; just increases confidence before hardware implementation*

```
type HeapCell = struct {  
    state: CellState;  
    is_whnf: bool;  
    to_target: bool;  
    reachable: bool  
};
```



```
type CellState = enum {  
    FreeList, WorkList, Marked, Unmarked  
};
```

# Evaluation: FPGA implementaion

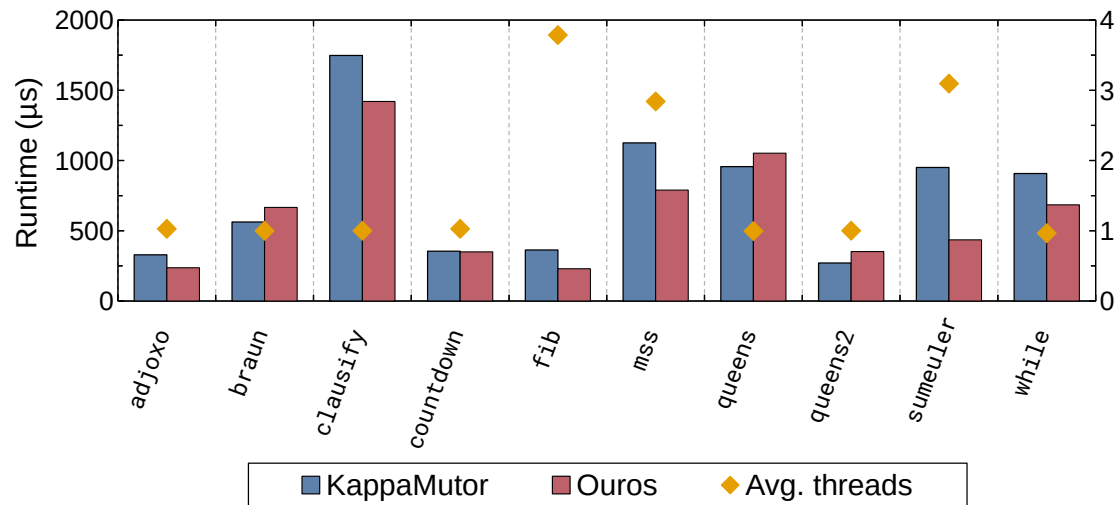
---

| Design      | Mul | GC | LUT<br>(%)   | Register<br>(%) | BRAM<br>(%) | URAM<br>(%) | $f_{\max}$<br>(MHz) |
|-------------|-----|----|--------------|-----------------|-------------|-------------|---------------------|
| Kappa-Mutor | ✗   | ✗  | 6174 (0.71)  | 1185 (0.07)     | 10.5 (0.78) | 4 (0.63)    | 168                 |
|             | ✓   | ✗  | 5992 (0.69)  | 1185 (0.07)     | 10.5 (0.78) | 4 (0.63)    | 128                 |
| Ouros       | ✓   | ✓  | 13753 (1.58) | 3143 (0.18)     | 12 (0.89)   | 4 (0.63)    | 177                 |

- Compare against KappaMutor, a single-cycle combinator processor
- HW is written in Chisel HDL, and implemented on a Xilinx Alveo U50 FPGA
- 4K heap size; maximum 4 threads
- Heap maps to URAM; other memories map to BRAM

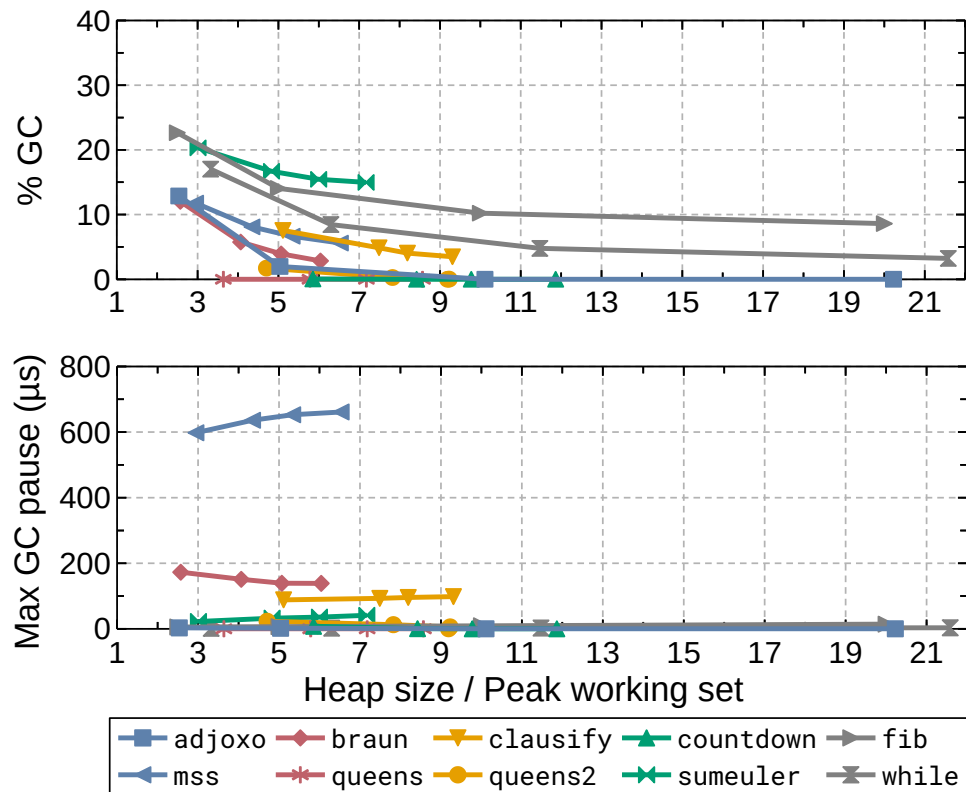
# Evaluation: Runtime comparison

---



- Comparison measured in wall-clock runtime (*lower is better*)
- Small scaled benchmarks, no GC is triggered
- Ouros is on average (geometric mean) **20.9%** faster; up to **118%** faster in richly threaded programs like `sumeuler`

# Evaluation: GC



- Benchmarks in large scale version, up to 150k+ GC rounds
- **GC%:** exposed runtime slow down due to GC
- **Max GC pause:** execution pausing time
- GC overhead is totally hidden for 3 out of the 10 programs

# Conclusion

---

# Conclusion

---

- A pipelined dataflow-driven processor for FP
  - Latency hiding for large combinator reduction
  - Automatic multi-threading

# Conclusion

---

- A pipelined dataflow-driven processor for FP
  - Latency hiding for large combinator reduction
  - Automatic multi-threading
- A simplified GC for graph reduction is included

# Conclusion

---

- A pipelined dataflow-driven processor for FP
  - Latency hiding for large combinator reduction
  - Automatic multi-threading
- A simplified GC for graph reduction is included
- Questions?