CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compiling for Concise Code and Efficient I/O

Sebastian Ertel, Andrés Goens, Justus Adam and Jeronimo Castrillon

Chair for Compiler Construction
TU Dresden

27th International Conference on Compiler Construction
Vienna, 25.2.2018

Microservice-based Architectures

- Backend (server) system requirements: scalable, flexible, fault-tolerant …

Microservice-based Architectures

- Backend (server) system requirements: scalable, flexible, fault-tolerant …

[http://magnasoma.com/#/monolith-3]

- Monolithic server program

Microservice-based Architectures

- Backend (server) system requirements: scalable, flexible, fault-tolerant …

[http://magnasoma.com/#/monolith-3]

- Monolithic server program

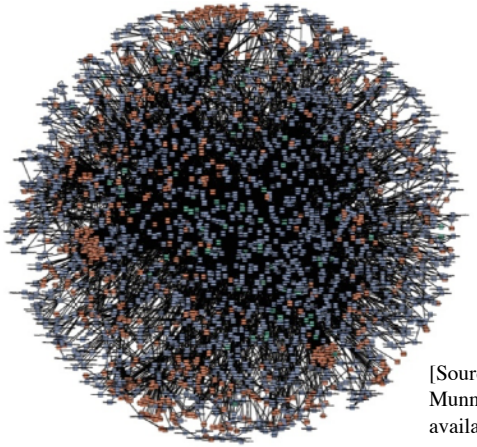
- Microservice: independent function
- Microservices compose to larger services

Microservice-based Architectures

- Backend (server) system requirements: scalable, flexible, fault-tolerant …

[http://magnasoma.com/#/monolith-3]

- Monolithic server program



http://amazon.com

[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: h p://bit.ly/2zboHTK]

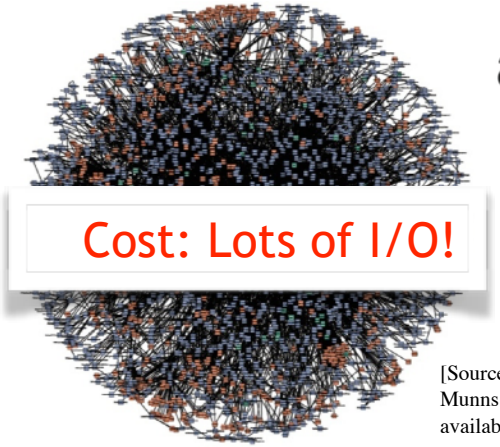
- Microservice: independent function
- Microservices compose to larger services

Microservice-based Architectures

- Backend (server) system requirements: scalable, flexible, fault-tolerant …

[http://magnasoma.com/#/monolith-3]

- Monolithic server program



amazon

http://amazon.com

[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at: h p://bit.ly/2zboHTK]

- Microservice: independent function
- Microservices compose to larger services

Fighting I/O

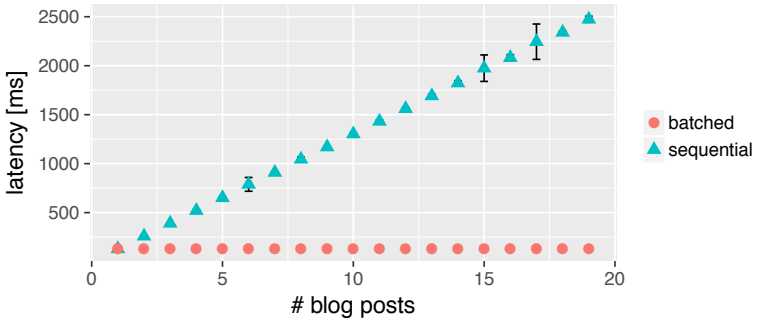
Efficient I/O

Concise Code

Fighting I/O

Efficient I/O

- Batching (& deduplication):
  1 I/O call vs. n I/O calls



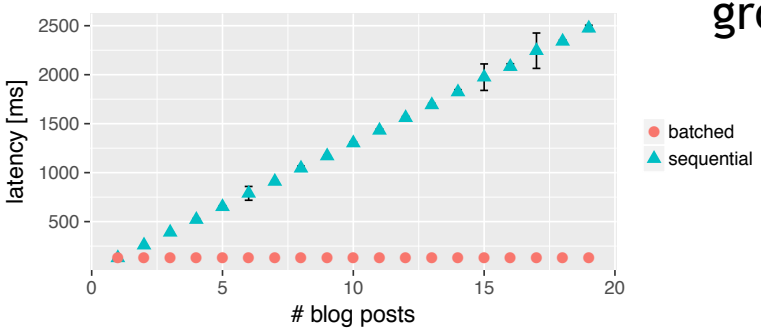
- Concurrency:
  Computation—IO—Computation
- (Caching)

Concise Code

Fighting I/O

Efficient I/O

- Batching (& deduplication): 1 I/O call vs. n I/O calls



- Concurrency: Computation—IO—Computation
- (Caching)

group I/O calls

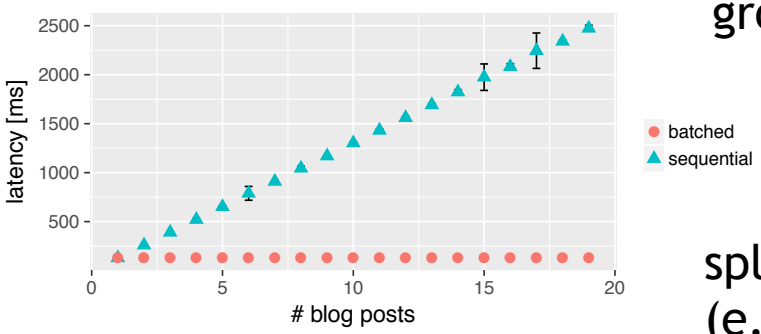
Concise Code

- Breaks modularity!

Fighting I/O

Efficient I/O

- Batching (& deduplication): 1 I/O call vs. n I/O calls



- Concurrency: Computation—IO—Computation
- (Caching)

group I/O calls

split up code (e.g. into threads)

Concise Code

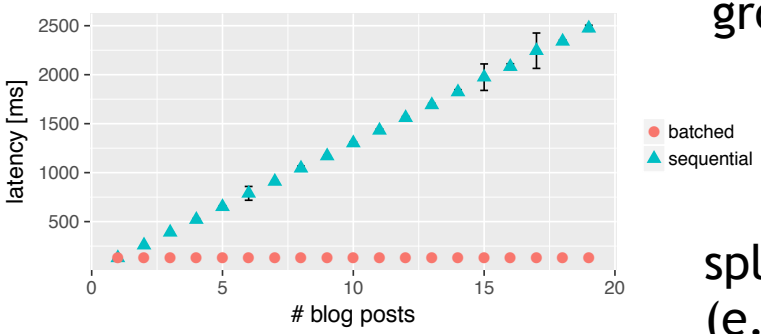
- Breaks modularity!

- Introduces optimization aspect (concurrency) into the algorithm!

Fighting I/O

Efficient I/O

- Batching (& deduplication): 1 I/O call vs. n I/O calls



- Concurrency: Computation—IO—Computation
- (Caching)

group I/O calls

Concise Code

- Breaks modularity!

- Batching and concurrency do not easily compose!

split up code (e.g. into threads)

- Introduces optimization aspect (concurrency) into the algorithm!

Fighting I/O

Efficient I/O

- Batching (& deduplication):
  1 I/O call vs. n I/O calls

- Concurrency:
  Computation—IO—Computation
- (Caching)

group I/O calls

split up code
(e.g. into threads)

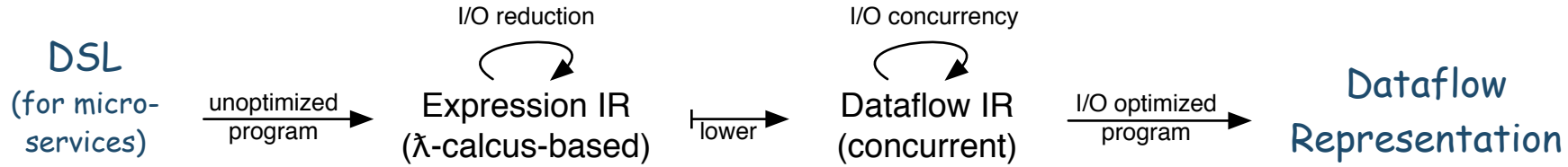
Concise Code

- Breaks modularity!

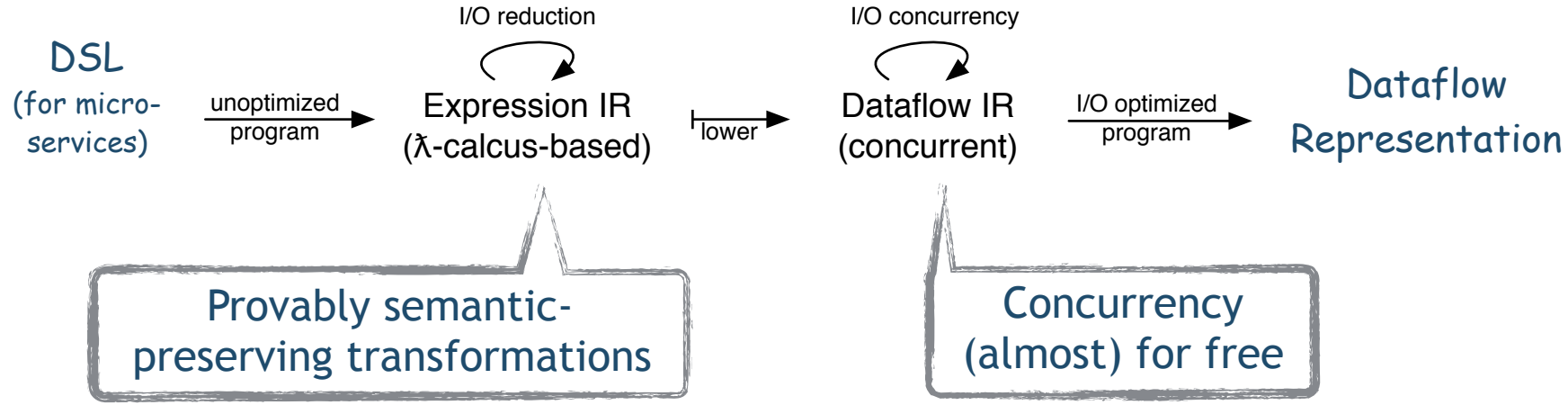
- Introduces optimization aspect (concurrency) into the algorithm!

Release the developer from this burden → Compiler-based Approach!

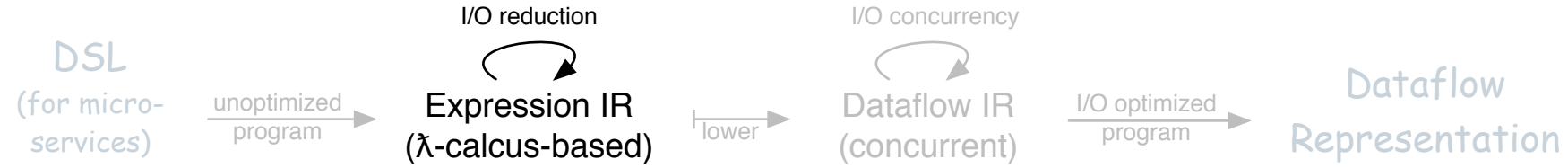
Ÿauhau - A compiler framework for microservices



Ÿauhau - A compiler framework for microservices



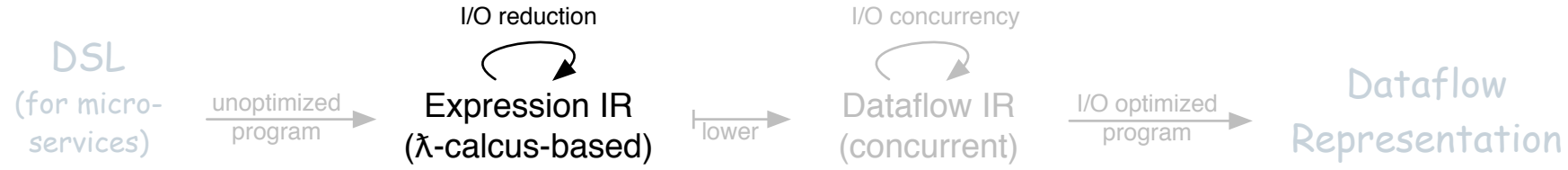
A minimalistic DSL for microservices

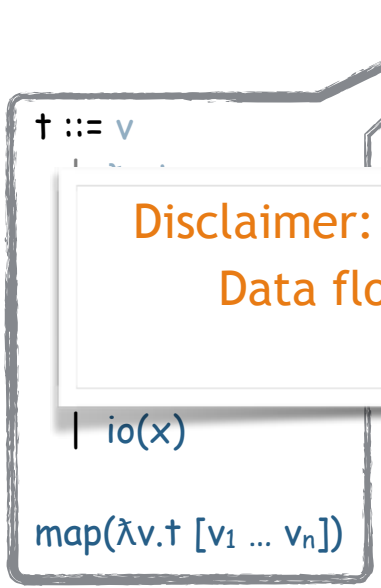



```
t ::= v
    | λv.t
    | t t
    | let v=t in t
    | if (t t t)
    | ff_f(x_1 ... x_n)
    | io(x)

map(λv.t [v_1 ... v_n])
```

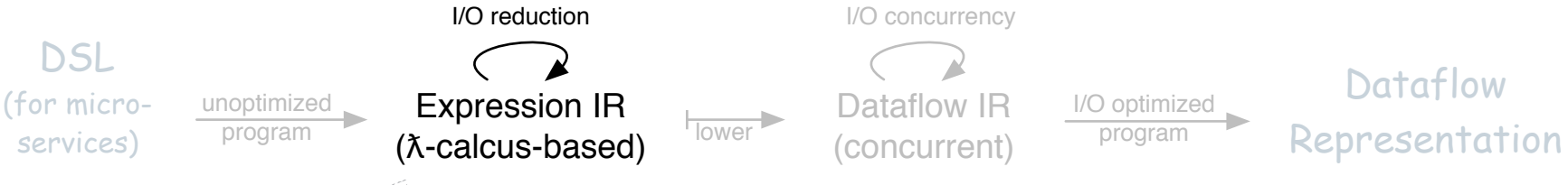
Semantic-preserving Transformations





Disclaimer: Presentation diverges from paper for visualization purposes. Data flow graphs before and after the transformations instead of lambda expressions.

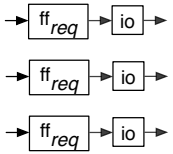
Semantic-preserving Transformations



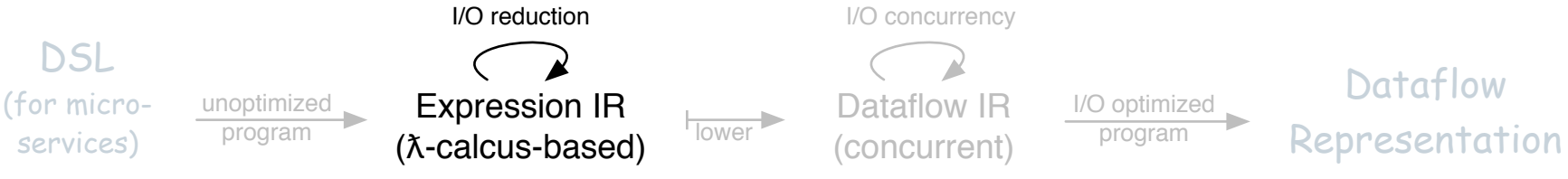
t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

map(λv.t [$v_1 \dots v_n$])

- Grouping <u>independent</u> I/O calls:



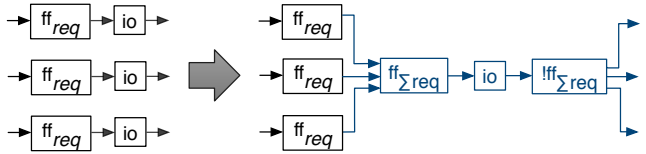
Semantic-preserving Transformations



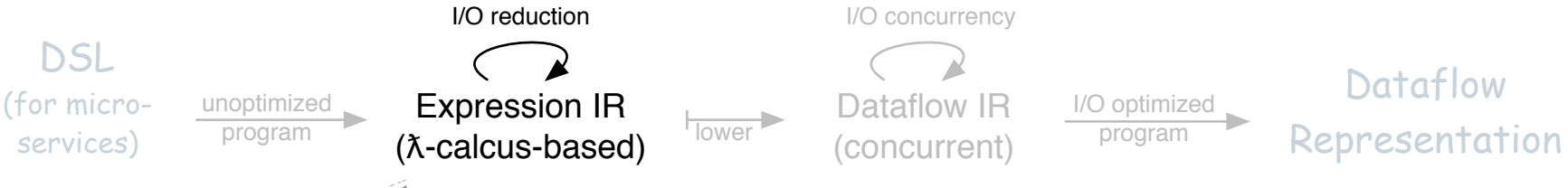
t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

map(λv.t [$v_1 \dots v_n$])

- Grouping <u>independent</u> I/O calls:



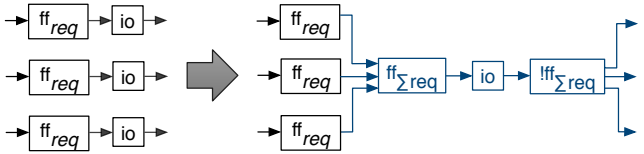
Semantic-preserving Transformations



t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

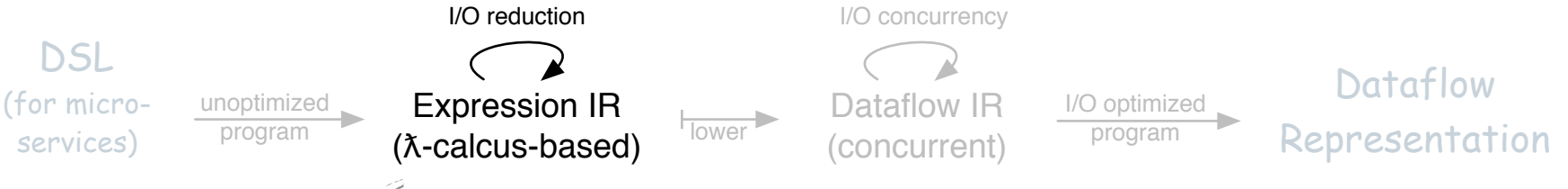
map(λv.t $[v_1 \dots v_n]$)

- Grouping independent I/O calls:



- Lifting I/O calls:

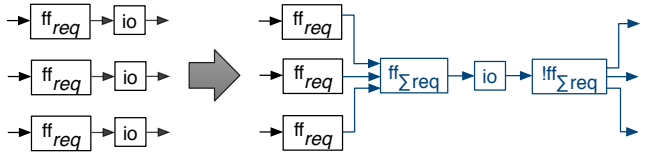
Semantic-preserving Transformations



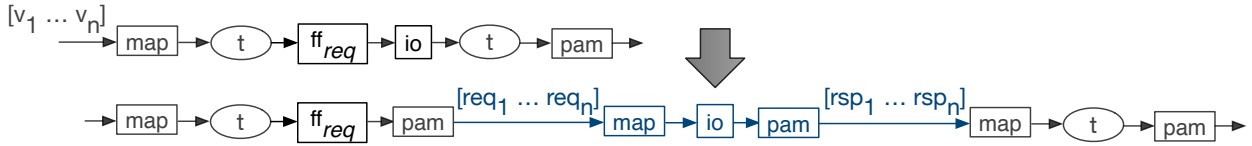
t ::= v
| λv.t
| t t
| let v=t in t
| if (t t t)
| $ff_f(x_1 \dots x_n)$
| io(x)

map(λv.t $[v_1 \dots v_n]$)

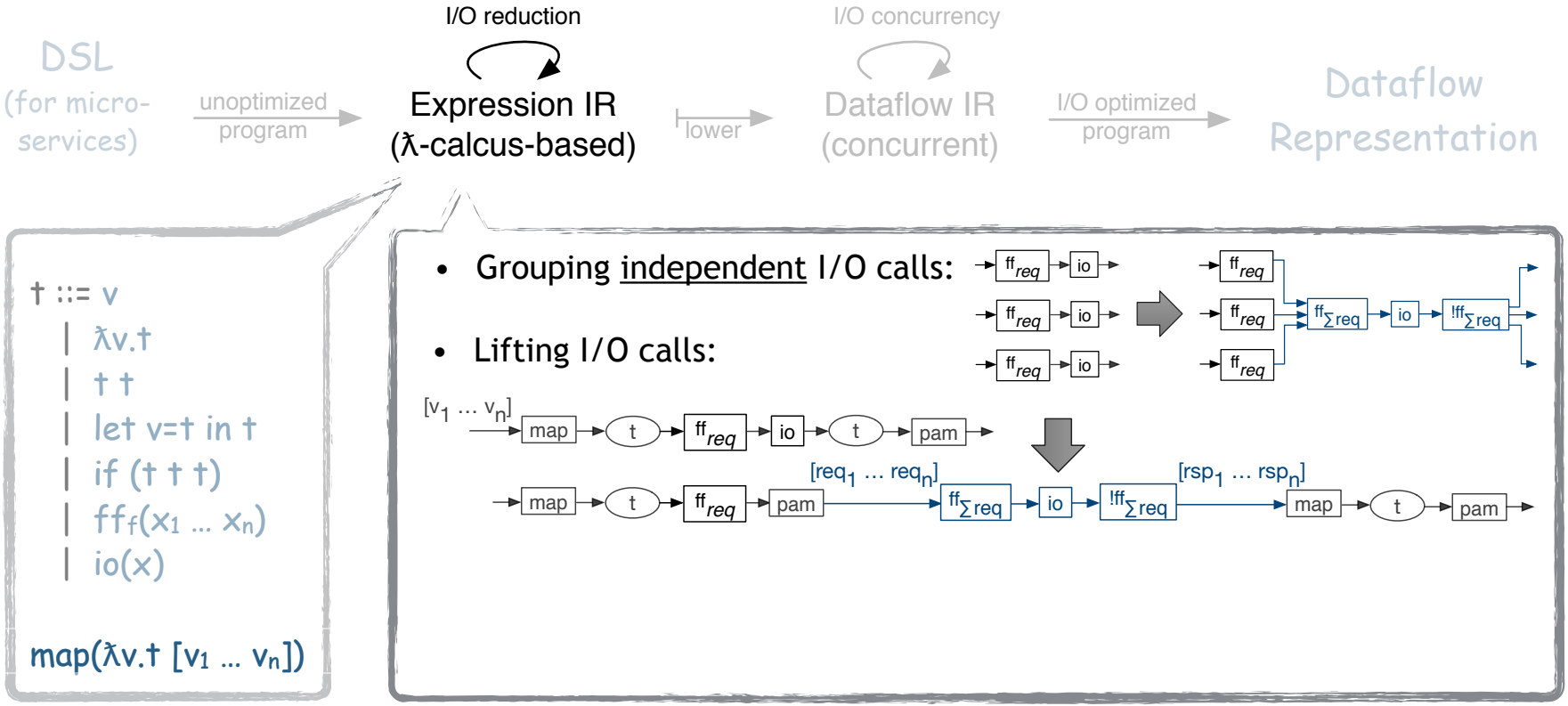
- Grouping independent I/O calls:



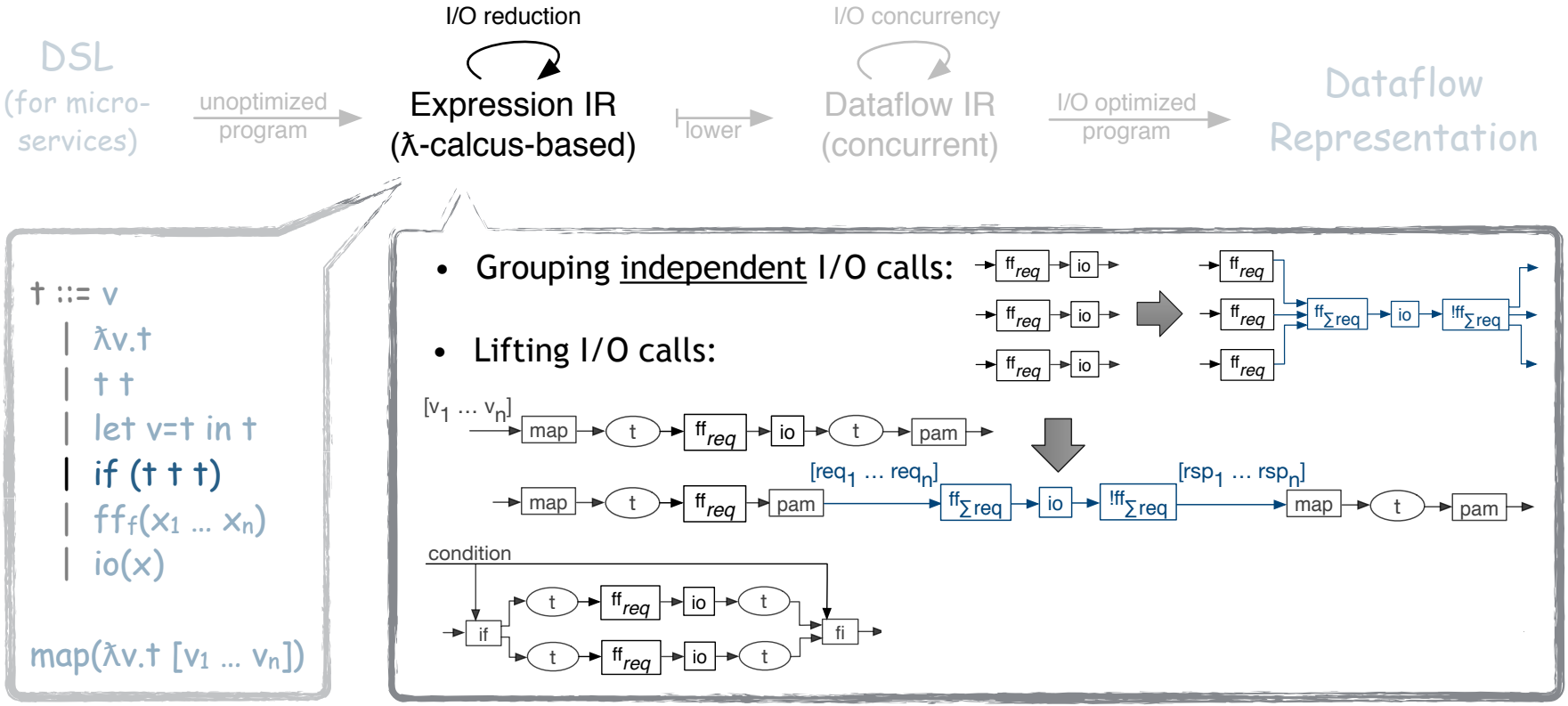
- Lifting I/O calls:



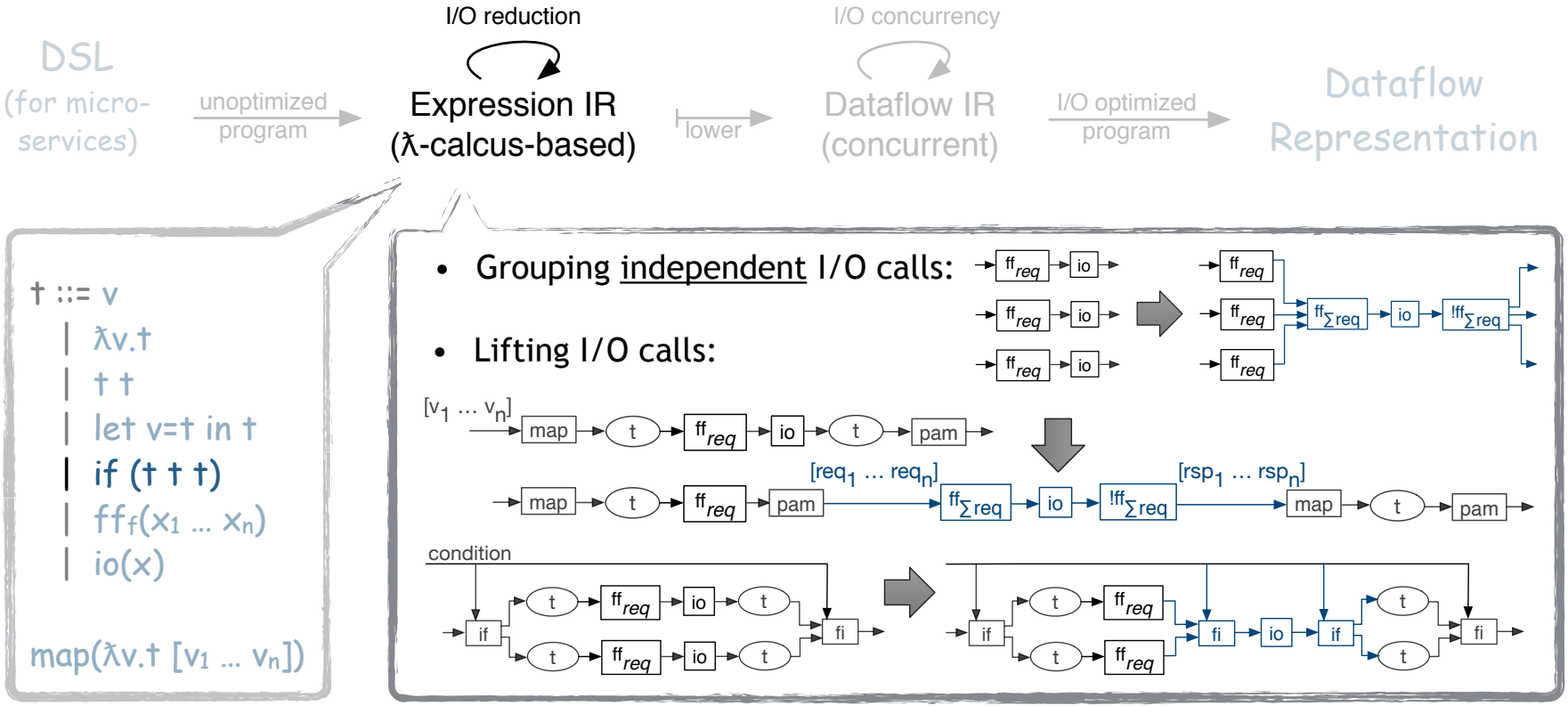
Semantic-preserving Transformations



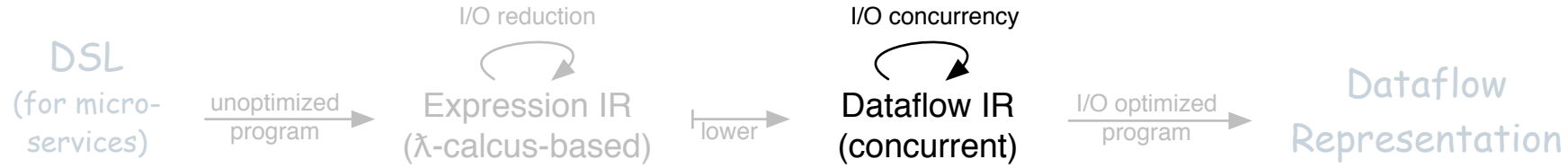
Semantic-preserving Transformations



Semantic-preserving Transformations

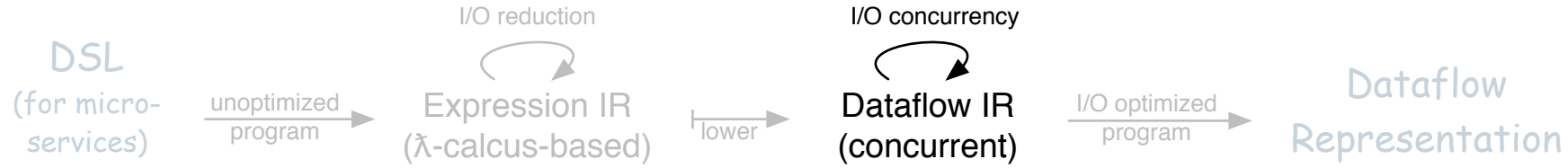


Bringing back Concurrency

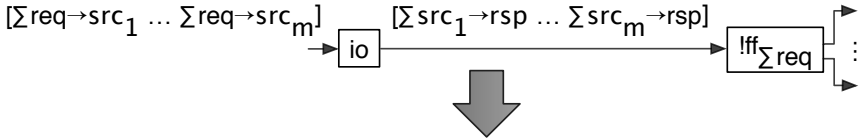


- Computation—IO ✅
- IO—IO
- IO—Computation

Bringing back Concurrency

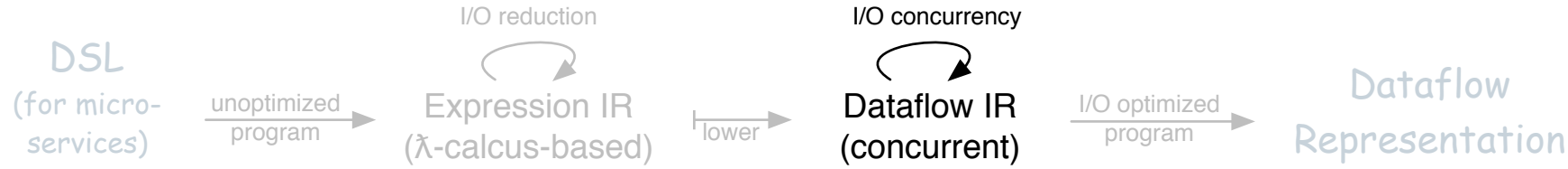


- Computation—IO ✅

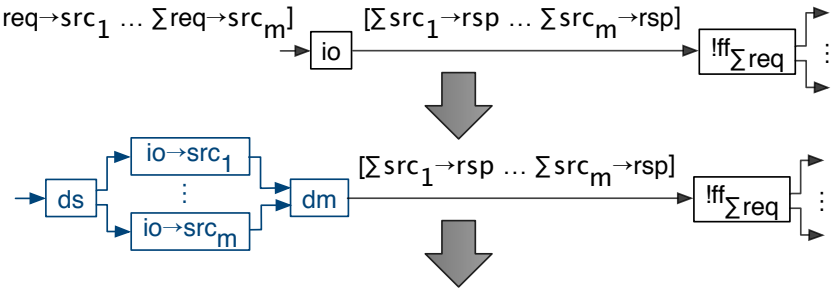


- IO—IO
- IO—Computation

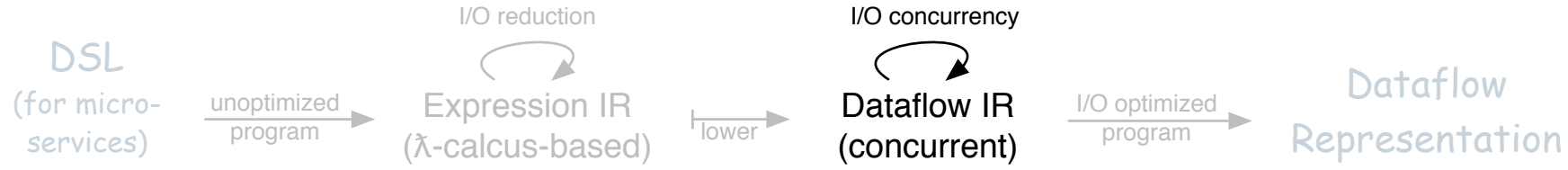
Bringing back Concurrency



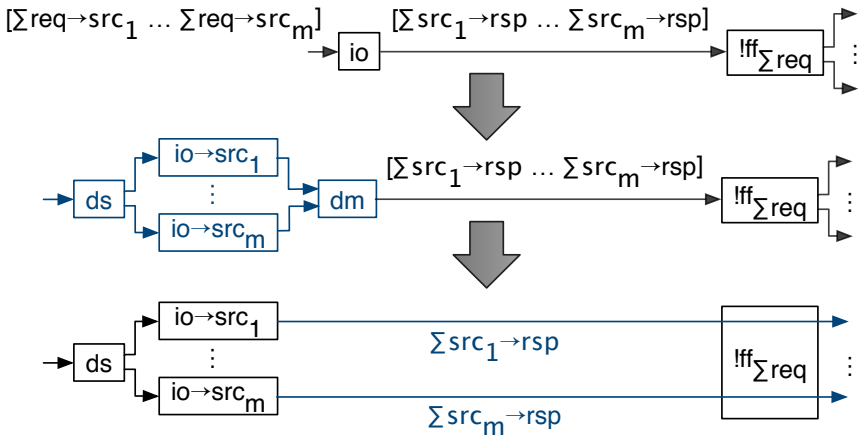
- Computation—IO ✅
- IO—IO ✅
- IO—Computation



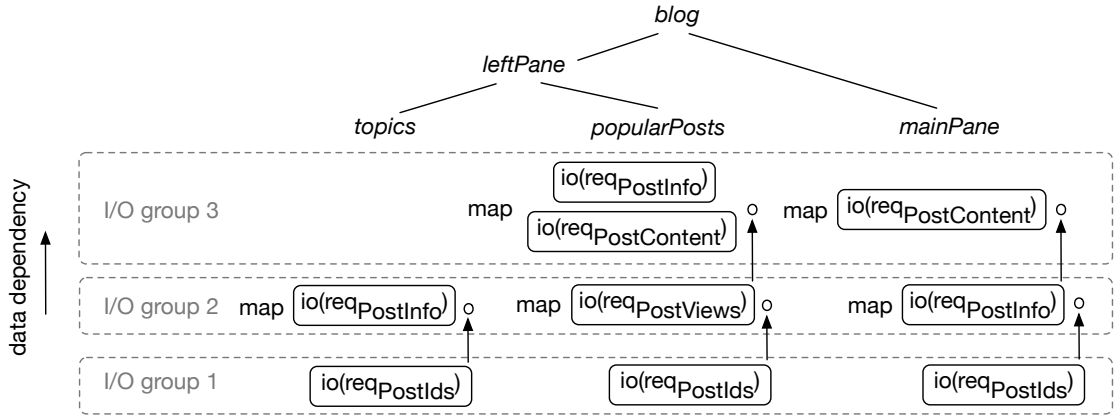
Bringing back Concurrency



- Computation—IO ✅
- IO—IO ✅
- IO—Computation ✅



Use Case: Blog



Version	seq	base	batch	full
Time [ms]	275 ± 25	292 ± 19	79 ± 1	66.5 ± 5.2

4x latency improvement!

Evaluation against State-of-the-art

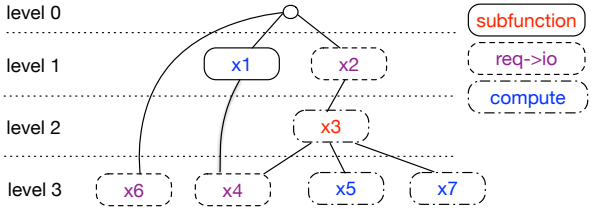
- Haxl → Facebook's Haskell-based approach on the abstraction of applicative functors.
- Muse → Similar to Twitter's Stitch, Clojure-based implementation of an AST interpretation.

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. ICFP '14.

Alexey Kachayev. 2015. Reinventing Haxl: Efficient, Concurrent and Concise Data Access. Presentation at EuroClojure 2015.

Got “microservices for evaluation”?

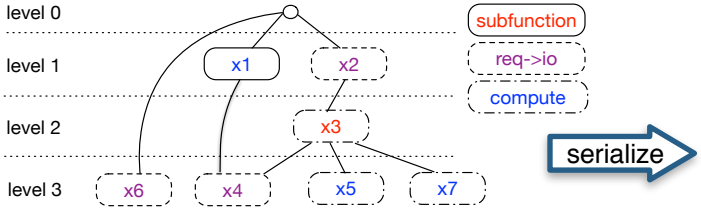
→ Level-Graphs:



Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers. MULTIPROG’18.

Got “microservices for evaluation”?

→ Level-Graphs:



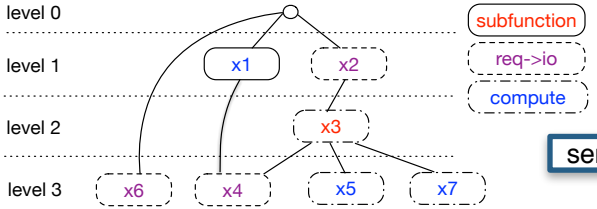
→ Programs:

- Haxl
- Muse
- Seq
- Ÿauhau

Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers. MULTIPROG’18.

Ÿauhau outperforms State-of-the-art

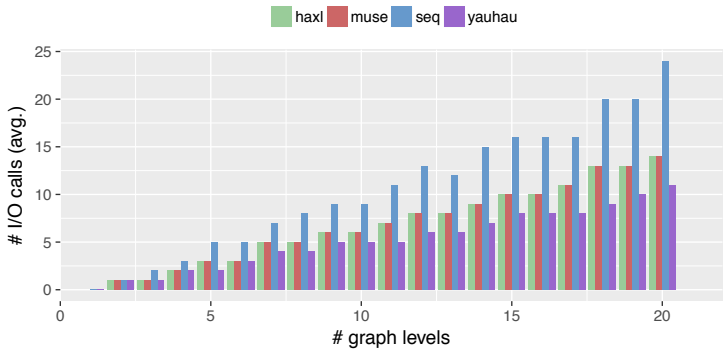
→ Level-Graphs:



serialize

→ Programs:

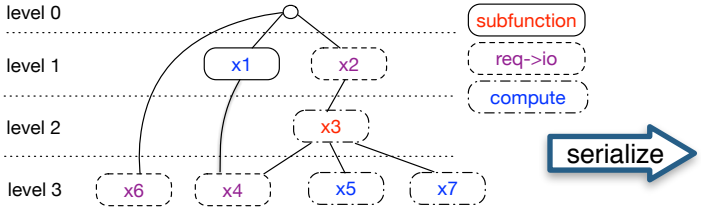
- Haxl
- Muse
- Seq
- Ÿauhau



Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers. MULTIPROG'18.

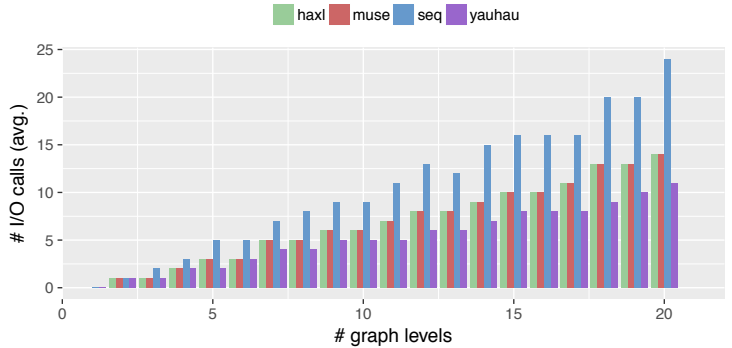
Ÿauhau outperforms State-of-the-art

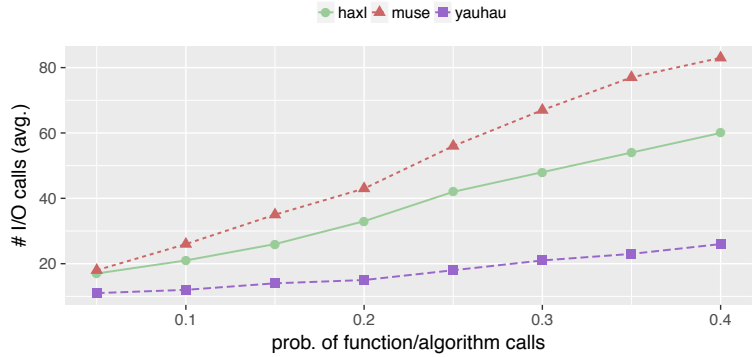
→ Level-Graphs:

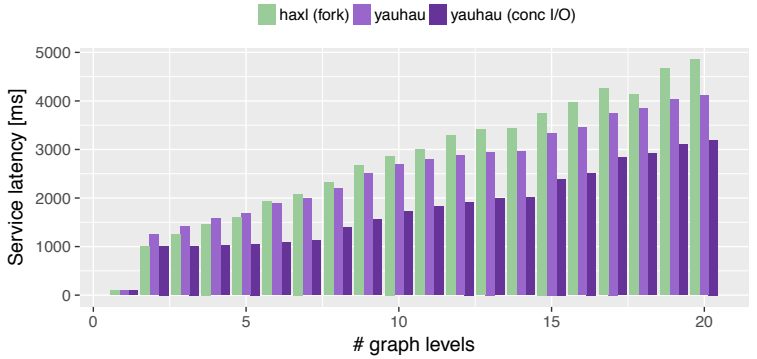


→ Programs:

Haxl
Muse
Seq
Ÿauhau







Andrés Goens, Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Level Graphs: Generating Benchmarks for Concurrency Optimizations in Compilers. MULTIPROG'18.

Conclusion

- I/O optimizations performed at the compiler level
  - Efficient I/O ✅
  - Concise Code ✅
- Use lambda calculus to provide semantic-preserving transformations!
- Use dataflow for concurrency/parallel execution!

Try it out:
https://tud-ccc.github.io/yauhau-doc/

Ÿauhau Code - Blog Use Case

```
(defn ohua-batching []
  (<-ohua
    (time-end (blog (time-begin three)))
    :compile-with-config {:df-transformations yauhau.ir-transform/transformations}))

                                                    (defalgo blog [pid]
                                                      (let [pop1 (popular-posts pid)
                                                            top (topics pid)
                                                            spane (render-side-pane pop1 top)
                                                            mpane (main-pane pid)]
                                                      (render-page spane mpane)))

          (defalgo topics [pid]
            (let [ids (fetch-ids pid)
                  topics2 (smap (algo [pid] (let [info (fetch-info pid)] (:topic info))) ids)
                  concatenated (concat topics2)
                  freqs (frequencies concatenated)]
              (render-topics freqs)))
```

Level Graphs Code

```
do
  -- level 3
 (local4, local5, local6, local7) <- (,,,)
   <$> getData "source" [100] <*> compute [100]
   <*> getData "source" [100] <*> compute [100]
 -- level 2
 local3 <- subfun3 [100, local4, local5, local7]
 -- level 1
 (local1, local2) <- (,)
    <$> compute [100, local4]
    <*> getData "source" [100, local3]
 -- level 0
 getData "source" [100,local1,local2,local6]
```

Haxl (< GHC 8)

```
(cats/mlet
   [; level 3
    [local-4 local-5 local-6 local-7]
      (cats/<$>
        clojure.core/vector
        (get-data "source" 100)
        (cats/<$> (compute (cats/return 100)))
        (get-data "source" 100)
        (cats/<$> (compute (cats/return 100))))
    ; level 2
    local-3
      (subfun-3 local-4 local-5 local-7)
    ; level 1
    [local-1 local-2]
      (cats/<$>
           clojure.core/vector
           (cats/<$>
             (compute (cats/return 100)
                      (cats/return local-4)))
           (get-data "source" 100 local-3))]
   ; level 0
   (get-data "source" 100 local-1 local-2
             local-6))))
```

Muse (with Cats)

```
(let
   [; level 3
    [local-4 local-5 local-6 local-7]
      (vector (get-data "source" 100)
              (compute 100)
              (get-data "source" 100)
              (compute 100))
    ; level 2
    local-3 (subfun-3 local-4 local-5 local-7)
    ; level 1
    [local-1 local-2]
      (vector (compute 100 local-4)
              (get-data "source" 100 local-3))]
   ; level 0
   (get-data "source" 100 local-1 local-2
             local-6))))
```

Ÿauhau