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 ...

- Monolithic server program
  - Microservice: independent function
  - Microservices compose to larger services

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

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

- Monolithic server program

- Microservice: independent function

- Microservices compose to larger services

Microservice-based Architectures

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

- Monolithic server program
- Microservice: independent function
- Microservices compose to larger services

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

Fighting I/O

Efficient I/O

Concise Code
Efficient I/O

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

Concise Code

- Concurrency:
  Computation–IO–Computation

- (Caching)
Fight I/O

Efficient I/O

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

Concise Code

- Breaks modularity!

group I/O calls

- Concurrency:
  Computation–IO–Computation

- (Caching)
Fighting I/O

Efficient I/O

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

<table>
<thead>
<tr>
<th># blog posts</th>
<th>latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
</tr>
<tr>
<td>10</td>
<td>1500</td>
</tr>
<tr>
<td>15</td>
<td>2000</td>
</tr>
<tr>
<td>20</td>
<td>2500</td>
</tr>
</tbody>
</table>

● batched
● sequential

Concise Code

- Breaks modularity!

group I/O calls

- Concurrency: Computation–IO–Computation

- (Caching)

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

Concise Code

- Breaks modularity!

- Batching and concurrency do not easily compose!

- Introduces optimization aspect (concurrency) into the algorithm!

- Concurrency: Computation—I/O—Computation

- (Caching)

- Batched & sequential

- Efficient I/O

- Concurrency: Computation—I/O—Computation

- Batching and concurrency do not easily compose!

- Introduces optimization aspect (concurrency) into the algorithm!
Fighting I/O

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

Concise Code
- Breaks modularity!

Concurrency:
- Computation—IO—Computation

Batching and concurrency do not easily compose!
- Introduces optimization aspect (concurrency) into the algorithm!

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

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

Split up code (e.g. into threads)
- (Caching)
Šauhau - A compiler framework for microservices

**DSL** (for micro-services) → **I/O reduction** → **Expression IR** (ƛ-calculus-based) → **I/O concurrency** → **I/O optimized program** → **Dataflow Representation**
Yauhau - A compiler framework for microservices

- DSL (for microservices)
- I/O reduction
- Expression IR (\(\lambda\)-calculus-based)
- I/O concurrency
- Dataflow IR (concurrent)
- Dataflow Representation

- Provably semantic-preserving transformations
- Concurrency (almost) for free
A minimalistic DSL for microservices

DSL (for micro-services) → unoptimized program → I/O reduction

Expression IR (\(\lambda\)-calculus-based) → I/O lower

I/O optimized program → I/O concurrency

Dataflow IR (concurrent) → Dataflow Representation

\[ t ::= v \\
| \lambda v. t \\
| t \ t \\
| let v=t in t \\
| if (t \ t \ t) \\
| ff_f(x_1 \ldots x_n) \\
| io(x) \\
| map(\lambda v. t [v_1 \ldots v_n]) \]
Semantic-preserving Transformations

Configuration:

1. **DSL** (for micro-services) → **unoptimized program** → **Expression IR** (λ-calculus-based) → **I/O reduction**
2. **Expression IR** → **I/O concurrency** → **Dataflow IR** (concurrent) → **I/O optimized program** → **Dataflow Representation**

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

**Dataflow IR (concurrent)**

- **Expression IR (λ-calculus-based)**
- **I/O reduction**
- **I/O concurrency**

### DSL (for microservices)

- Unoptimized program

### I/O reduction

- Grouping **independent** I/O calls:
  - `ff` \[ `req` \[ `io` \]
  - `ff` \[ `req` \[ `io` \]
  - `ff` \[ `req` \[ `io` \]

### I/O optimized program

© S. Ertel, Chair for Compiler Construction. 2018
Semantic-preserving Transformations

DSL (for micro-services) → unoptimized program → Expression IR (λ-calculus-based) → I/O reduction → I/O optimized program → Dataflow IR (concurrent) → I/O concurrency → Dataflow Representation

- Grouping independent I/O calls:

  \[
  \begin{align*}
  \mathbb{I} & ::= v \\
  & \mid \lambda v. t \\
  & \mid t \ t \\
  & \mid \text{let } v = t \text{ in } t \\
  & \mid \text{if } (t \ t \ t) \\
  & \mid \text{ff}_f(x_1 \ldots x_n) \\
  & \mid \text{io}(x) \\
  \text{map}(\lambda v. t \ [v_1 \ldots v_n])
  \end{align*}
  \]
Semantic-preserving Transformations

### DSL (for micro-services)

- Expression IR ($\lambda$-calculus-based)

- I/O reduction

- I/O concurrency

- Dataflow IR (concurrent)

- I/O optimized program

#### Dataflow Representation

- Grouping independent I/O calls:

- Lifting I/O calls:

  \[
  t ::= v \\
  \quad | \quad \lambda v.t \\
  \quad | \quad t \; t \\
  \quad | \quad let \; v=t \; in \; t \\
  \quad | \quad if \; (t \; t \; t) \\
  \quad | \quad ff_f(x_1 \; \ldots \; x_n) \\
  \quad | \quad io(x)
  \]

  \[
  map(\lambda v.t \; [v_1 \; \ldots \; v_n])
  \]
Semantic-preserving Transformations

**DSL** (for micro-services) | I/O reduction | I/O concurrency | Dataflow Representation
---|---|---|---
unoptimized program | Expression IR (λ-calculus-based) | Dataflow IR (concurrent) |  

- **Grouping independent I/O calls:**
- **Lifting I/O calls:**

\[ t ::= v \mid \lambda v.t \mid t \mid t \mid \text{let } v = t \text{ in } t \mid \text{if } (t \quad t \quad t) \mid \text{ff}_{f}(x_{1} \ldots x_{n}) \mid \text{io}(x) \]

\[ \text{map}(\lambda v.t \ [v_{1} \ldots v_{n}]) \]
Semantic-preserving Transformations

**DSL** (for micro-services) → unoptimized program → Expression IR (λ-calculus-based) → I/O reduction

Expression IR (λ-calculus-based) → Dataflow IR (concurrent) → I/O optimized program → Dataflow Representation

- Grouping **independent** I/O calls:
- Lifting I/O calls:

$t ::= v$
- $\lambda v.t$
- $t \cdot t$
- let $v = t$ in $t$
- if $(t \cdot t \cdot t)$
- $ff_f(x_1 ... x_n)$
- $io(x)$
- $map(\lambda v.t [v_1 ... v_n])$

$\Sigma_{req}$
Semantic-preserving Transformations

DSL (for micro-services) → unoptimized program → Expression IR (λ-calculus-based) → lower → Dataflow IR (concurrent) → I/O optimized program → Dataflow Representation

• Grouping independent I/O calls:
• Lifting I/O calls:

<table>
<thead>
<tr>
<th>t ::= v</th>
<th>λv.t</th>
<th>t t</th>
<th>let v=t in t</th>
<th>if (t t t)</th>
<th>ff_f(x_1 ... x_n)</th>
<th>io(x)</th>
</tr>
</thead>
</table>

map(λv.t [v_1 ... v_n])
Semantic-preserving Transformations

DSL (for micro-services) → unoptimized program → I/O reduction → Expression IR (λ-calculus-based) → I/O lower → I/O concurrency → Dataflow IR (concurrent) → I/O optimized program → Dataflow Representation

- Grouping independent I/O calls:

- Lifting I/O calls:

 conditioned I/O calls:

\[ t ::= v \]
\[ \lambda v.t \]
\[ t \cdot t \]
\[ \text{let } v=t \text{ in } t \]
\[ \text{if } (t \cdot t \cdot t) \]
\[ \text{ff}_{f}(x_1 \ldots x_n) \]
\[ \text{io}(x) \]
\[ \text{map}(\lambda v.t[v_1 \ldots v_n]) \]
Bringing back Concurrency

DSL (for micro-services) → unoptimized program → Expression IR (λ-calculus-based) → I/O reduction → Lower → Dataflow IR (concurrent) → I/O optimized program → Dataflow Representation

- Computation–IO
- IO–IO
- IO–Computation
Bringing back Concurrency

DSL (for micro-services) → unoptimized program → Expression IR (λ-calculus-based) → I/O reduction → I/O optimized program → Dataflow IR (concurrent) → I/O optimized program → Dataflow Representation

- Computation–IO
- IO–IO
- IO–Computation
Bringing back Concurrency

- **Computation–IO**
- **IO–IO**
- **IO–Computation**
Bringing back Concurrency

- **DSL** (for micro-services)
- **Expression IR** (λ-calculus-based)
- **Dataflow IR** (concurrent)

- **I/O reduction**
- **I/O concurrency**

• Computation–IO

• IO–IO

• IO–Computation

\[
\sum_{\text{req} \rightarrow \text{src}_1} \ldots \sum_{\text{req} \rightarrow \text{src}_m} \quad [\sum_{\text{src}_1 \rightarrow \text{rsp}} \ldots \sum_{\text{src}_m \rightarrow \text{rsp}}]
\]

\[
\sum_{\text{req} \rightarrow \text{src}_1} \ldots \sum_{\text{req} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]

\[
\sum_{\text{io} \rightarrow \text{src}_1} \ldots \sum_{\text{io} \rightarrow \text{src}_m}
\]
Use Case: Blog

<table>
<thead>
<tr>
<th>I/O group 1</th>
<th>I/O group 2</th>
<th>I/O group 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>io(reqPostIds)</td>
<td>io(reqPostIds)</td>
<td>io(reqPostIds)</td>
</tr>
<tr>
<td>map io(reqPostInfo)</td>
<td>map io(reqPostViews)</td>
<td>map io(reqPostInfo)</td>
</tr>
<tr>
<td>data dependency</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Got “microservices for evaluation”?

→ Level-Graphs:

Got “microservices for evaluation”?

→ Level-Graphs:

→ Programs:

Haxl
Muse
Seq
Ýauhau

Yauhau outperforms State-of-the-art

→ Level-Graphs:  

→ Programs:

Programs:

- Haxl
- Muse
- Seq
- Yauhau

Yauhau outperforms State-of-the-art

→ Level-Graphs: → Programs:

<table>
<thead>
<tr>
<th>Level-Graphs</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 0</td>
<td>Haxl</td>
</tr>
<tr>
<td>level 1</td>
<td>Muse</td>
</tr>
<tr>
<td>level 2</td>
<td>Seq</td>
</tr>
<tr>
<td>level 3</td>
<td>Yauhau</td>
</tr>
</tbody>
</table>

Haxl
Muse
Seq
Yauhau

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://tudccc.github.io/yauhau-doc/
(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)))
(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)))

(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))))

Haxl (< GHC 8)        Muse (with Cats)        Ŷauhau