This work is supported by

THE UNIVERSITY *of* EDINBURGH

School of informatics

EPSRC Centre for Doctoral Training in Pervasive Parallelism

EPSRC

Engineering and Physical Sciences Research Council

UNIVERSITY OF CAMBRIDGE

OCaml Labs

Asynchronous Effect-based Input and Output

Daniel Hillerström
daniel.hillerstrom@ed.ac.uk
http://homepages.inf.ed.ac.uk/s1467124

CDT Pervasive Parallelism
School of Informatics
The University of Edinburgh, UK

June 14, 2017

CDT Pervasive Parallelism Student Showcase

(based on joint work with Stephen Dolan, Spiros Eliopoulos, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White)

Multicore OCaml

Multicore OCaml adds shared-memory parallelism to OCaml.

Stephen Dolan

Leo White

KC Sivaramakrishnan

Jeremy Yallop

Anil Madhavapeddy

In addition, it adds *effect handlers* as the primary means for concurrency [1]

- enabling schedulers to be implemented as user-level libraries,
- providing fine-grained control over scheduling,
- while retaining direct-style programming.

For more information regarding the Multicore OCaml project see

`http://ocamllabs.io/doc/multicore.html`

Effect Handlers

Effect handlers provide a generalisation of exception handlers

Exceptions

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   raise (Fork task);
   do_something_else ()
 with
  | Fork task ->
    Queue.enqueue task
  | Yield ->
    Queue.enqueue
      (fun () -> ???);
    let task = Queue.dequeue () in
    task ()
```

Effects

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   perform (Fork task);
   do_something_else ()
 with
  | effect (Fork task) comp ->
    Queue.enqueue task;
    continue comp ()
  | effect Yield comp ->
    Queue.enqueue
      (fun () -> continue comp ());
    let task = Queue.dequeue () in
    task ()
```

Intuition Effect handlers are *exception handlers* + *resumable exceptions*

Effect Handlers

Effect handlers provide a generalisation of exception handlers

Exceptions

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   raise (Fork task);
   do_something_else ()
 with
  | Fork task ->
    Queue.enqueue task
  | Yield ->
    Queue.enqueue
      (fun () -> ???);
    let task = Queue.dequeue () in
    task ()
```

Effects

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   perform (Fork task);
   do_something_else ()
 with
  | effect (Fork task) comp ->
    Queue.enqueue task;
    continue comp ()
  | effect Yield comp ->
    Queue.enqueue
      (fun () -> continue comp ());
    let task = Queue.dequeue () in
    task ()
```

Intuition Effect handlers are *exception handlers* + *resumable exceptions*

Effect Handlers

Effect handlers provide a generalisation of exception handlers

Exceptions

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   raise (Fork task);
   do_something_else ()
 with
  | Fork task ->
    Queue.enqueue task
  | Yield ->
    Queue.enqueue
      (fun () -> ???);
    let task = Queue.dequeue () in
    task ()
```

Effects

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   perform (Fork task);
   do_something_else ()
 with
  | effect (Fork task) comp ->
    Queue.enqueue task;
    continue comp ()
  | effect Yield comp ->
    Queue.enqueue
      (fun () -> continue comp ());
    let task = Queue.dequeue () in
    task ()
```

Intuition Effect handlers are *exception handlers* + *resumable exceptions*

Effect Handlers

Effect handlers provide a generalisation of exception handlers

Exceptions

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   raise (Fork task);
   do_something_else ()
 with
  | Fork task ->
    Queue.enqueue task
  | Yield ->
    Queue.enqueue
      (fun () -> ???);
    let task = Queue.dequeue () in
    task ()
```

Effects

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   perform (Fork task);
   do_something_else ()
 with
  | effect (Fork task) comp ->
    Queue.enqueue task;
    continue comp ()
  | effect Yield comp ->
    Queue.enqueue
      (fun () -> continue comp ());
    let task = Queue.dequeue () in
    task ()
```

Intuition Effect handlers are *exception handlers* + *resumable exceptions*

Effect Handlers

Effect handlers provide a generalisation of exception handlers

Exceptions

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   raise (Fork task);
   do_something_else ()
 with
  | Fork task ->
    Queue.enqueue task
  | Yield ->
    Queue.enqueue
      (fun () -> ???);
    let task = Queue.dequeue () in
    task ()
```

Effects

```
let _ =
 let run_q = Queue.empty () in
 try
   let task = do_something () in
   perform (Fork task);
   do_something_else ()
 with
  | effect (Fork task) comp ->
    Queue.enqueue task;
    continue comp ()
  | effect Yield comp ->
    Queue.enqueue
      (fun () -> continue comp ());
    let task = Queue.dequeue () in
    task ()
```

Intuition Effect handlers are *exception handlers* + *resumable exceptions*

Systems Programming with Effect Handlers?

Concurrent systems programming today (in functional languages)

- Callback style (e.g. Node.Js)
- Monadic style (e.g. Haskell, OCaml)

Research Hypothesis

Effect handlers provide a compelling direct-style abstraction suitable for systems programming

Evaluation We have built an *Asynchronous Effect-based IO* (AEIO) library for overlapping IO operations which we put to use in a web server.

Experimental Setup

We evaluate the performance of three web servers

1. OCaml state-of-art: `httpaf` with `Async` 113.33.03[1] (vanilla OCaml)
2. Effect-based: `httpaf` with `aeio`[1] (Multicore OCaml)
3. Go 1.6.3 using `net/http` constrained to a single core

The workload was generated by `wrk2`

`https://github.com/giltene/wrk2`

The experiments were conducted on a standard machine

- 3 Ghz Intel Core i7
- 16 GB main memory
- 64-bit Ubuntu 16.10

[1] uses libev event loop (using *epoll*)

Results I

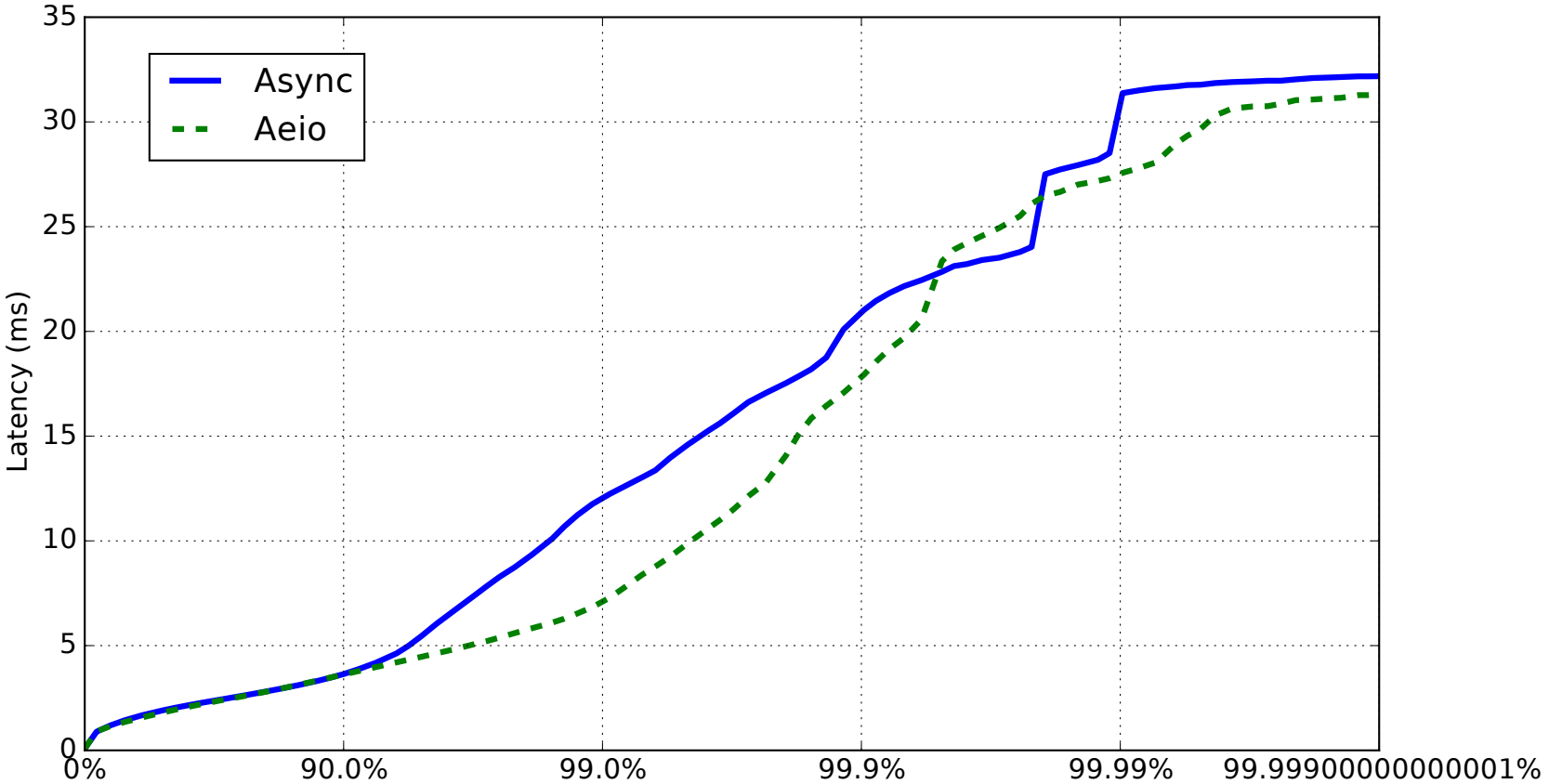

Figure: Medium contention 1000 connections, 10000 requests/sec

Results II

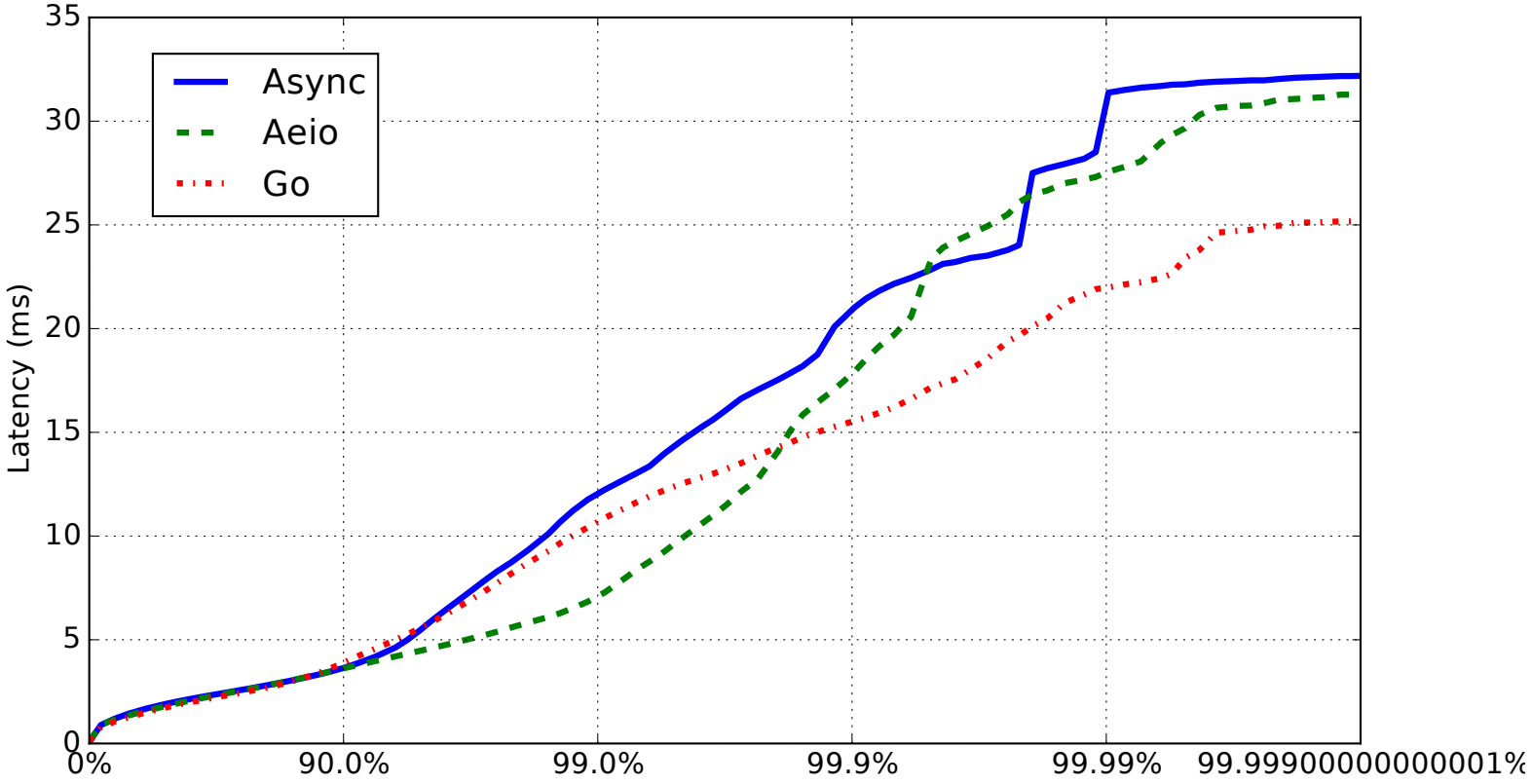

Figure: Medium contention 1000 connections, 10000 requests/sec (incl. Go)

Conclusions

In summary

- Handlers provide composable concurrency in direct-style
- Performs on par with the state of the art in OCaml
- Multicore OCaml is young, yet promising results

The Multicore OCaml compiler

`https://github.com/ocamllabs/ocaml-multicore`

Asynchronous Effect-based IO library for Multicore OCaml

`https://github.com/kayceesrk/ocaml-aeio`

Full details are available in our paper [2]

`http://kcsrk.info/papers/system_effects_may_17.pdf`

References

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy.
Effective concurrency through algebraic effects.
OCaml Workshop, 2015.

Stephen Dolan andSpiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White.
Concurrent system programming with effect handlers.
Trends in Functional Programming, 2017.