Retrofitting Effect Handlers onto OCaml

“KC” Sivaramakrishnan

IIT MADRAS

Retrofitting Effect Handlers onto OCaml

“KC” Sivaramakrishnan

IIT MADRAS

See PLDI’21 paper

Concurrent Programming

- Computations may be *suspended* and *resumed* later

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
  - ✦ async/await — JavaScript, Python, Rust, C# 5.0, F#, …
  - ✦ generators — Python, Javascript, …
  - ✦ coroutines — C++, Kotlin, Lua, …
  - ✦ futures & promises — JavaScript, Swift, …

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
  - ✦ async/await — JavaScript, Python, Rust, C# 5.0, F#, …
  - ✦ generators — Python, Javascript, …
  - ✦ coroutines — C++, Kotlin, Lua, …
  - ✦ futures & promises — JavaScript, Swift, …
- Often include different primitives for concurrent programming
  - ✦ JavaScript has async/await, generators, promises, and callbacks!!

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
  - ✦ **Lwt** and **Async** - concurrent programming libraries
  - ✦ Callback-oriented programming with monadic syntax **>>=**

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
  - ✦ **Lwt** and **Async** - concurrent programming libraries
  - ✦ Callback-oriented programming with monadic syntax **>>=**
- Suffers many pitfalls of *callback-oriented programming*
  - ✦ No backtraces, no exceptions, more closures
  - ✦ Monads split the ecosystem into *Asynchronous* and *Synchronous*
    - ✤ Bob Nystrom, *"What colour is your function?"*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
  - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*
- Effect handlers ~= *first-class, restartable exceptions*
  - ✦ Structured programming with *delimited continuations*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control flow mechanisms
  - ✦ Exceptions, generators, lightweight thre IO, coroutines as *libraries*
- Effect handlers ~= *first-class, restartab*
  - ✦ Structured programming with *delimited*

- Direct-style asynchronous I/O
- Generators
- Resumable parsers
- Probabilistic Programming
- Reactive UIs
- ....

https://github.com/ocaml-multicore/effects-examples

An example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

An example

effect declaration

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

An example

effect declaration

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

computation

An example

effect declaration

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

computation

handler

An example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

handler

An example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

An example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

resume suspended computation

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

Stepping through the example

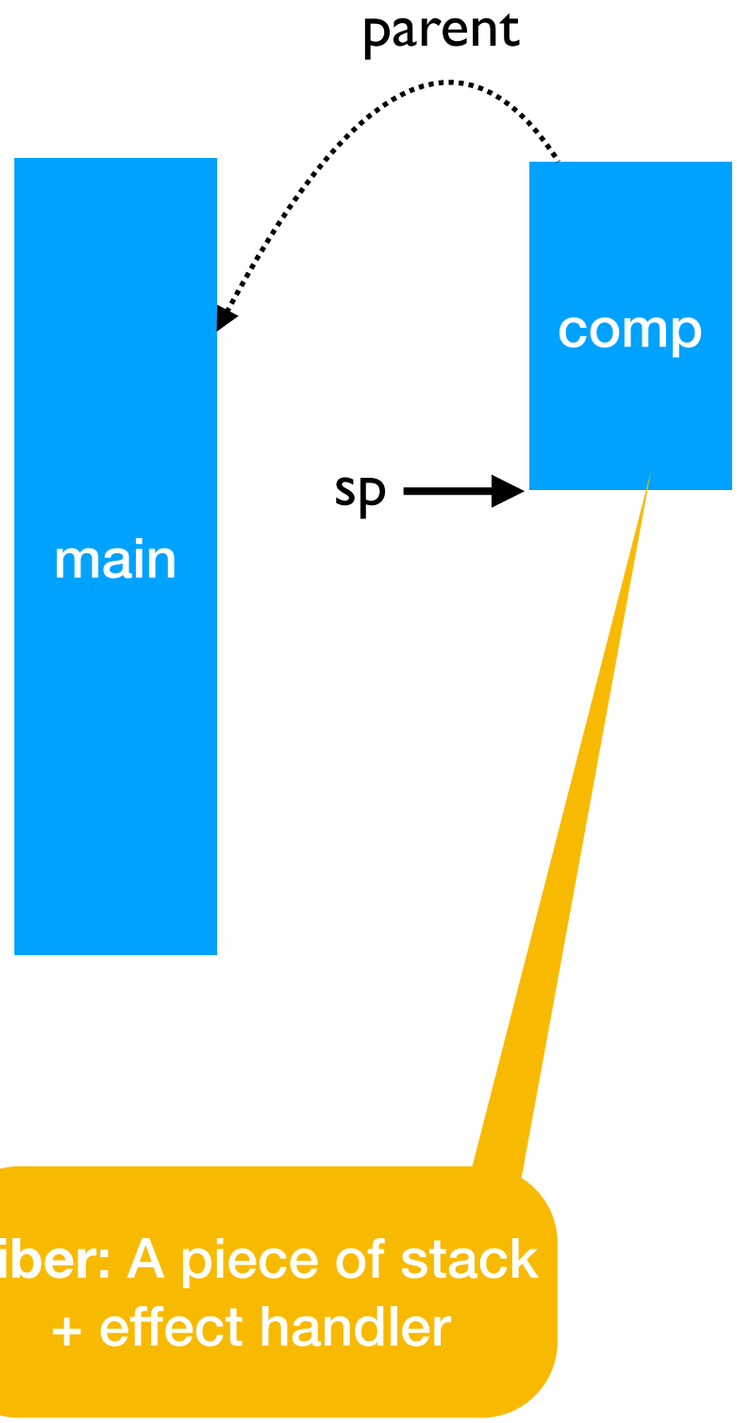
```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

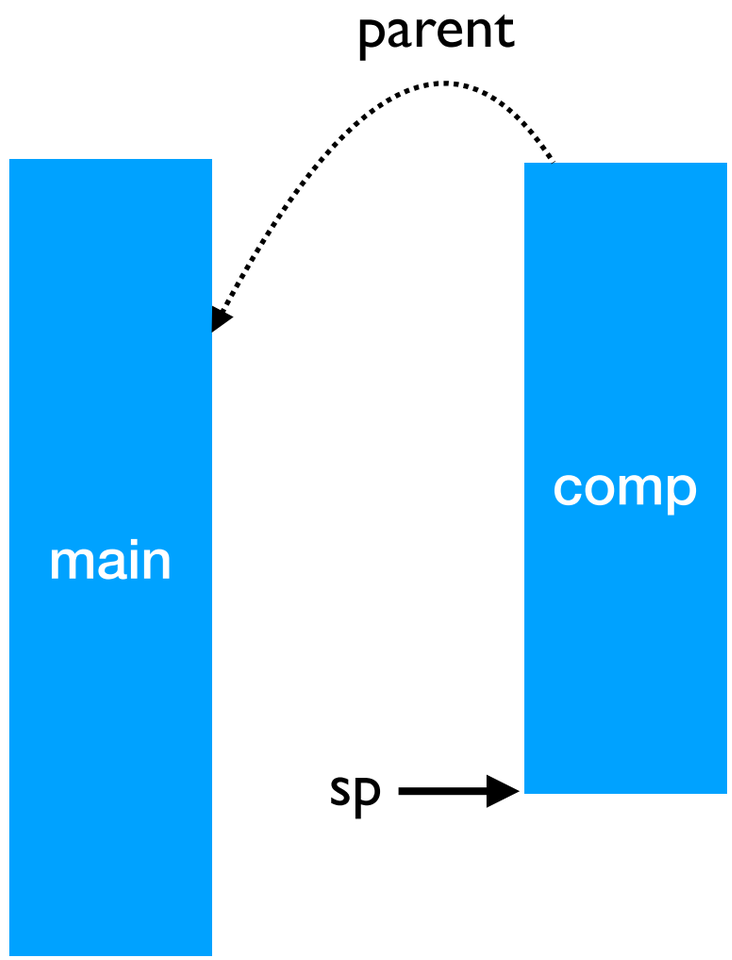
pc →



Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

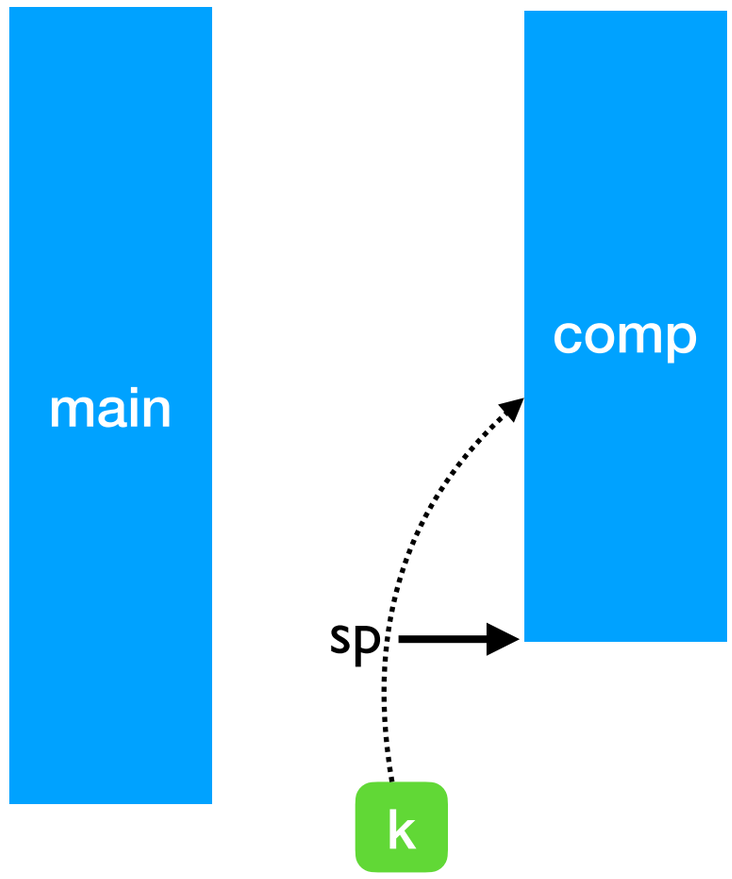


0

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

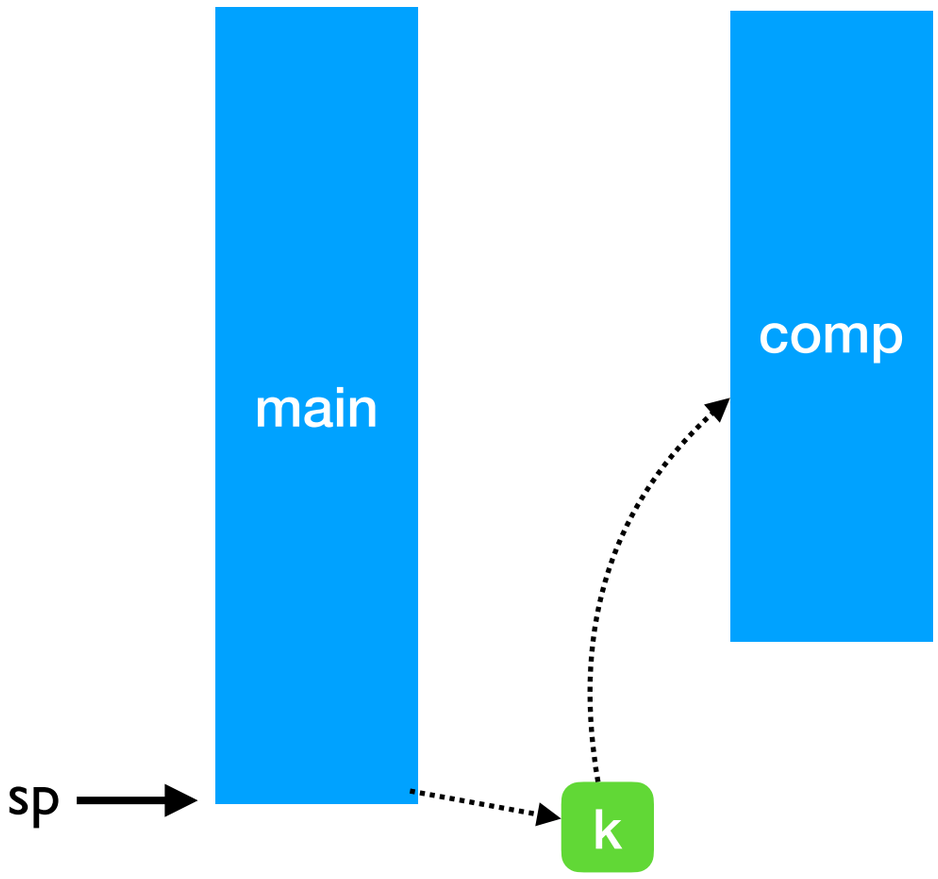


0

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

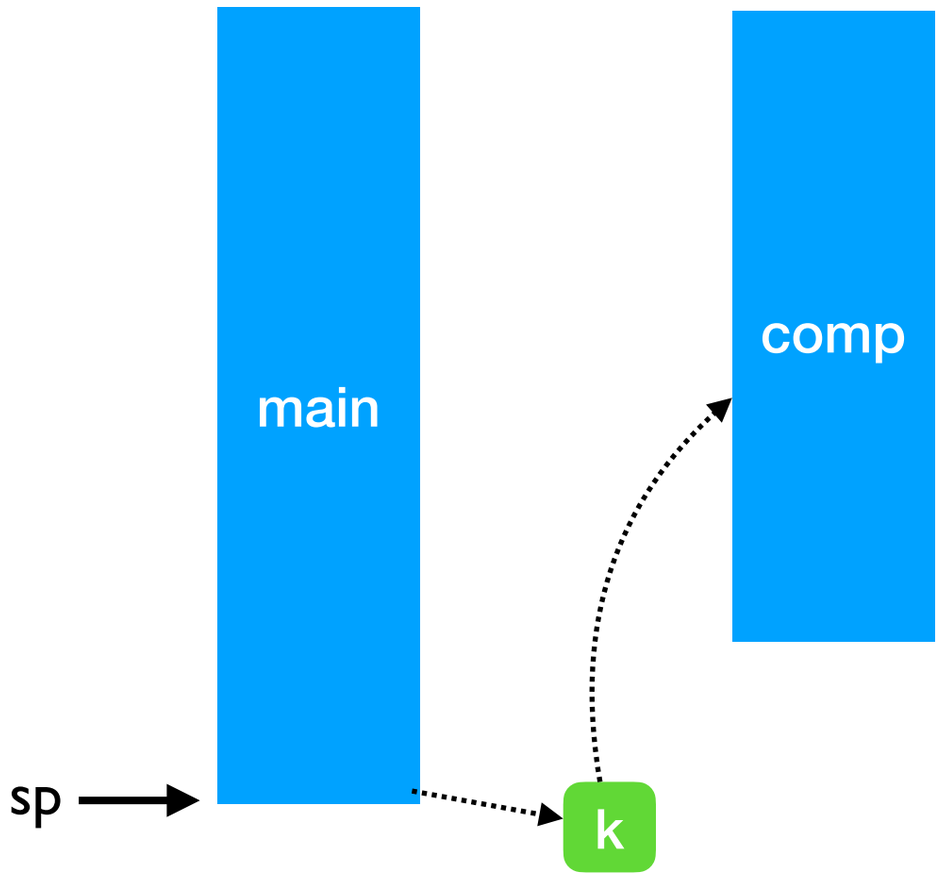


0

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

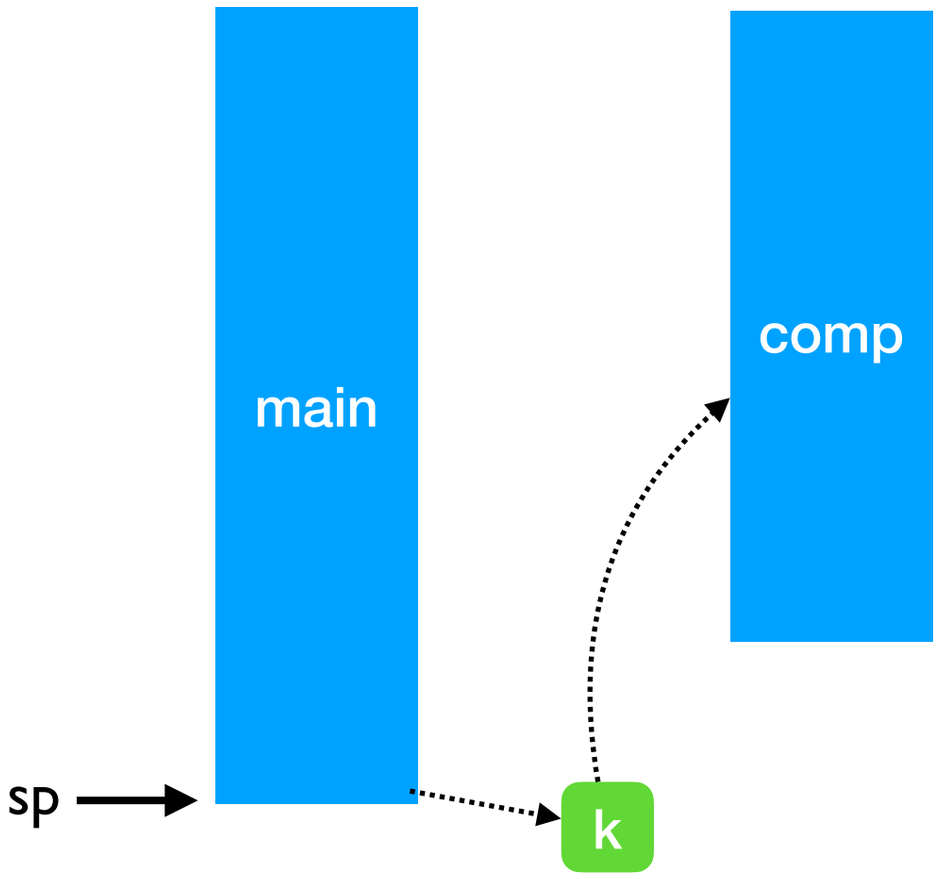


0

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

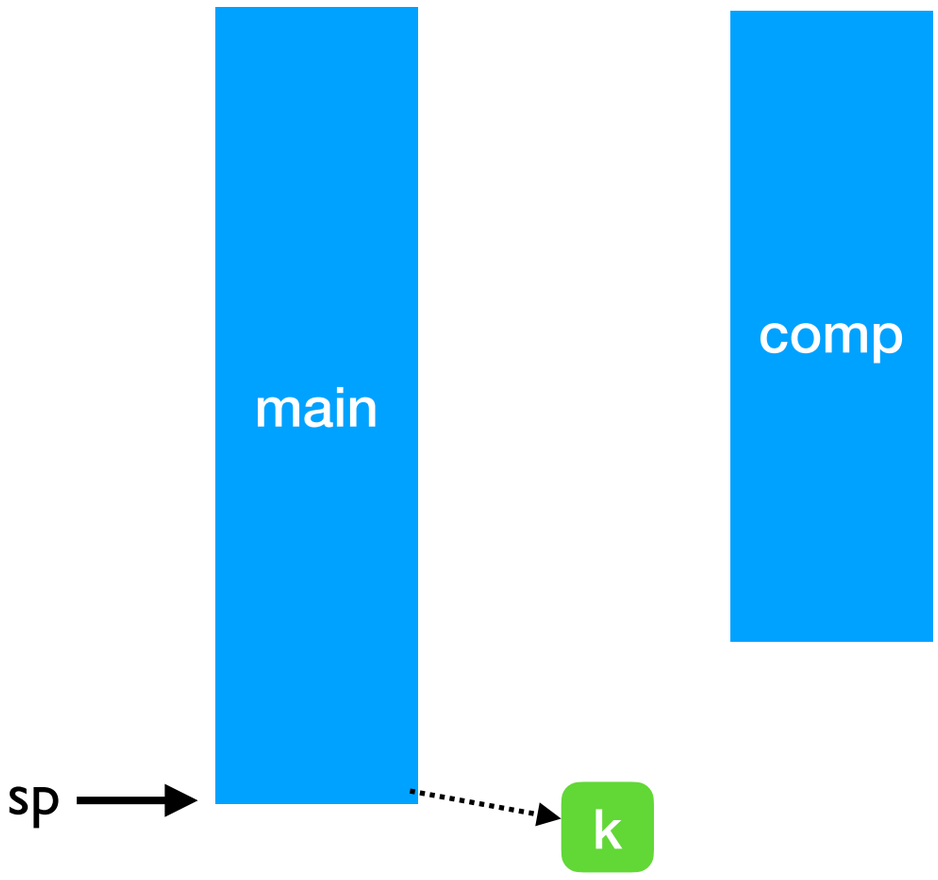


0 1

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

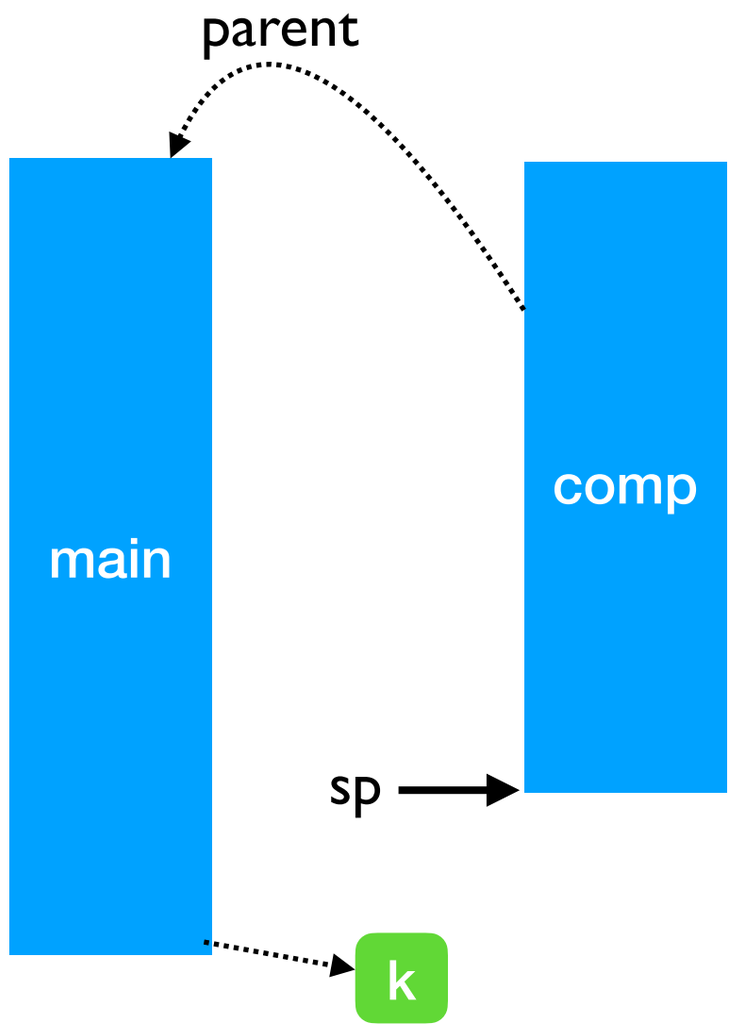


0 1

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

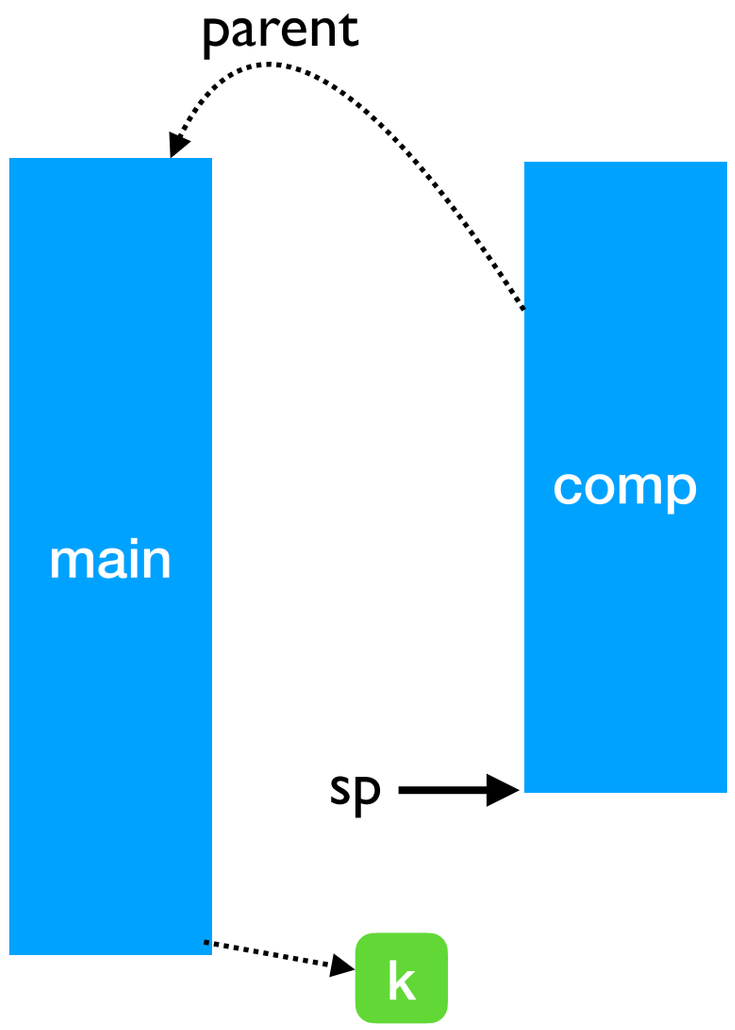


0 1

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

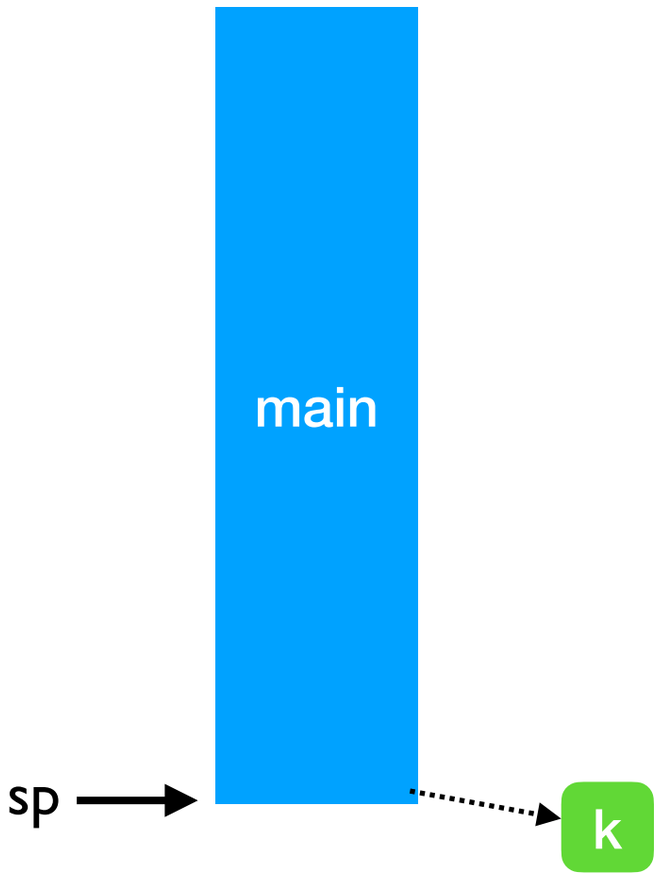


0 1 2

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →

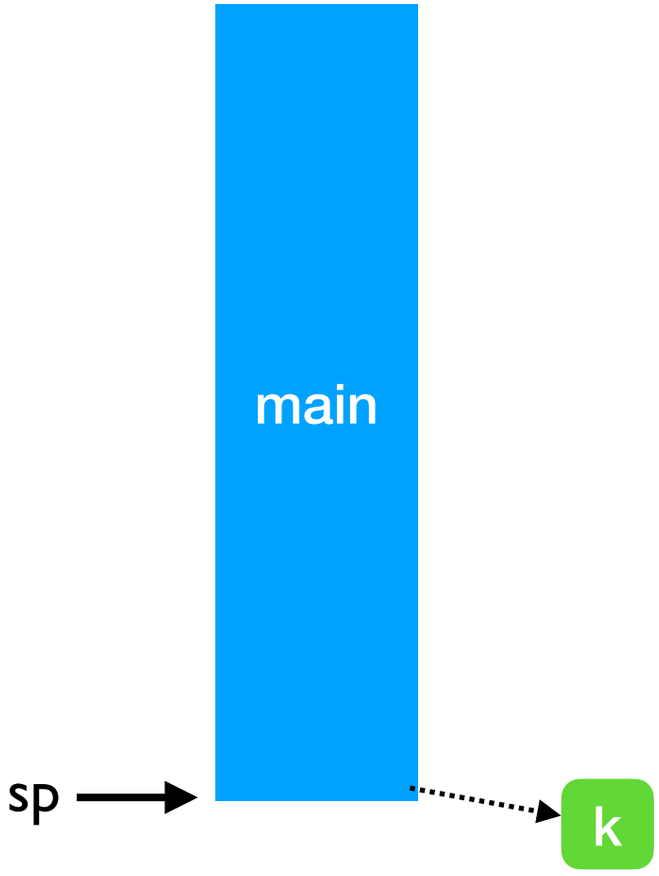


0 1 2 3

Stepping through the example

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

pc →



0 1 2 3 4

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit
```

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit

let run main =
  ... (* assume queue of continuations *)
  let run_next () =
    match dequeue () with
    | Some k -> continue k ()
    | None -> ()
in
let rec spawn f =
  match f () with
  | () -> run_next () (* value case *)
  | effect Yield k -> enqueue k; run_next ()
  | effect (Fork f) k -> enqueue k; spawn f
in
spawn main
```

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit
effect Yield : unit

let run main =
  ... (* assume queue of continuations *)
  let run_next () =
    match dequeue () with
    | Some k -> continue k ()
    | None -> ()
  in
  let rec spawn f =
    match f () with
    | () -> run_next () (* value case *)
    | effect Yield k -> enqueue k; run_next ()
    | effect (Fork f) k -> enqueue k; spawn f
  in
  spawn main

let fork f = perform (Fork f)
let yield () = perform Yield
```

Lightweight threading

```
let main () =
  fork (fun _ -> print_endline "1.a"; yield (); print_endline "1.b");
  fork (fun _ -> print_endline "2.a"; yield (); print_endline “2.b")
;;
run main
```

Lightweight threading

```
let main () =
  fork (fun _ -> print_endline "1.a"; yield (); print_endline "1.b");
  fork (fun _ -> print_endline "2.a"; yield (); print_endline “2.b")
;;
run main
```

```
1.a
2.a
1.b
2.b
```

Lightweight threading

```
let main () =
  fork (fun _ -> print_endline "1.a"; yield (); print_endline "1.b");
  fork (fun _ -> print_endline "2.a"; yield (); print_endline “2.b")
;;
run main
```

- Direct-style (no monads)
- User-code need not be aware of effects
- No Async vs Sync distinction

```
1.a
2.a
1.b
2.b
```

Retrofitting Challenges

- Millions of lines of legacy code
  - ✦ Written without *non-local control-flow* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*

Retrofitting Challenges

- Millions of lines of legacy code
  - ✦ Written without *non-local control-flow* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- OCaml uses the same system stack for both OCaml and C
  - ✦ Fast exceptions and FFI between C and OCaml
  - ✦ No stack overflow checks needed
  - ✦ Excellent compatibility with debugging (gdb) and profiling (perf) tools

Retrofitting Challenges

- Millions of lines of legacy code
  - ✦ Written without *non-local control-flow* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- OCaml uses the same system stack for both OCaml and C
  - ✦ Fast exceptions and FFI between C and OCaml
  - ✦ No stack overflow checks needed
  - ✦ Excellent compatibility with debugging (gdb) and profiling (perf) tools

Must preserve
feature, tooling, performance
compatibility

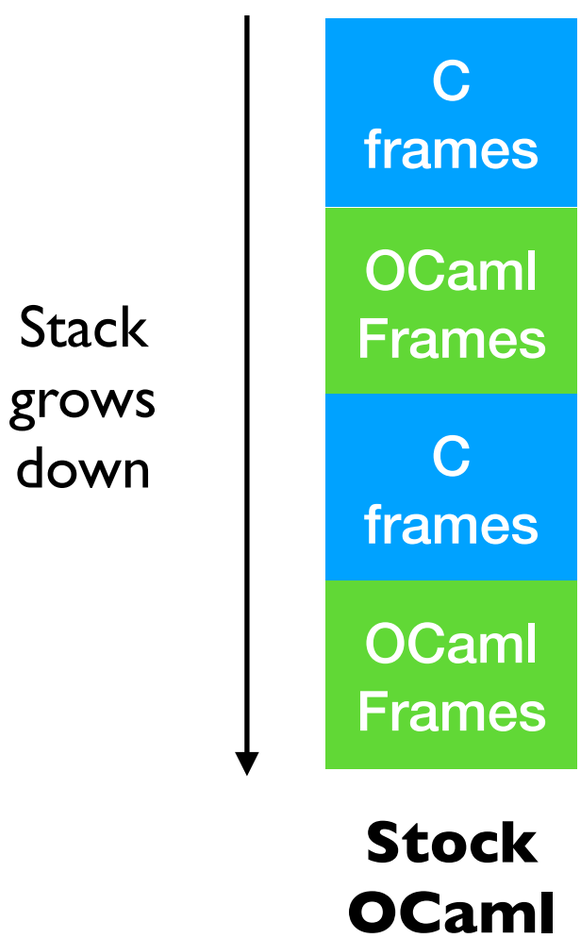
Representing Stacks & Continuations

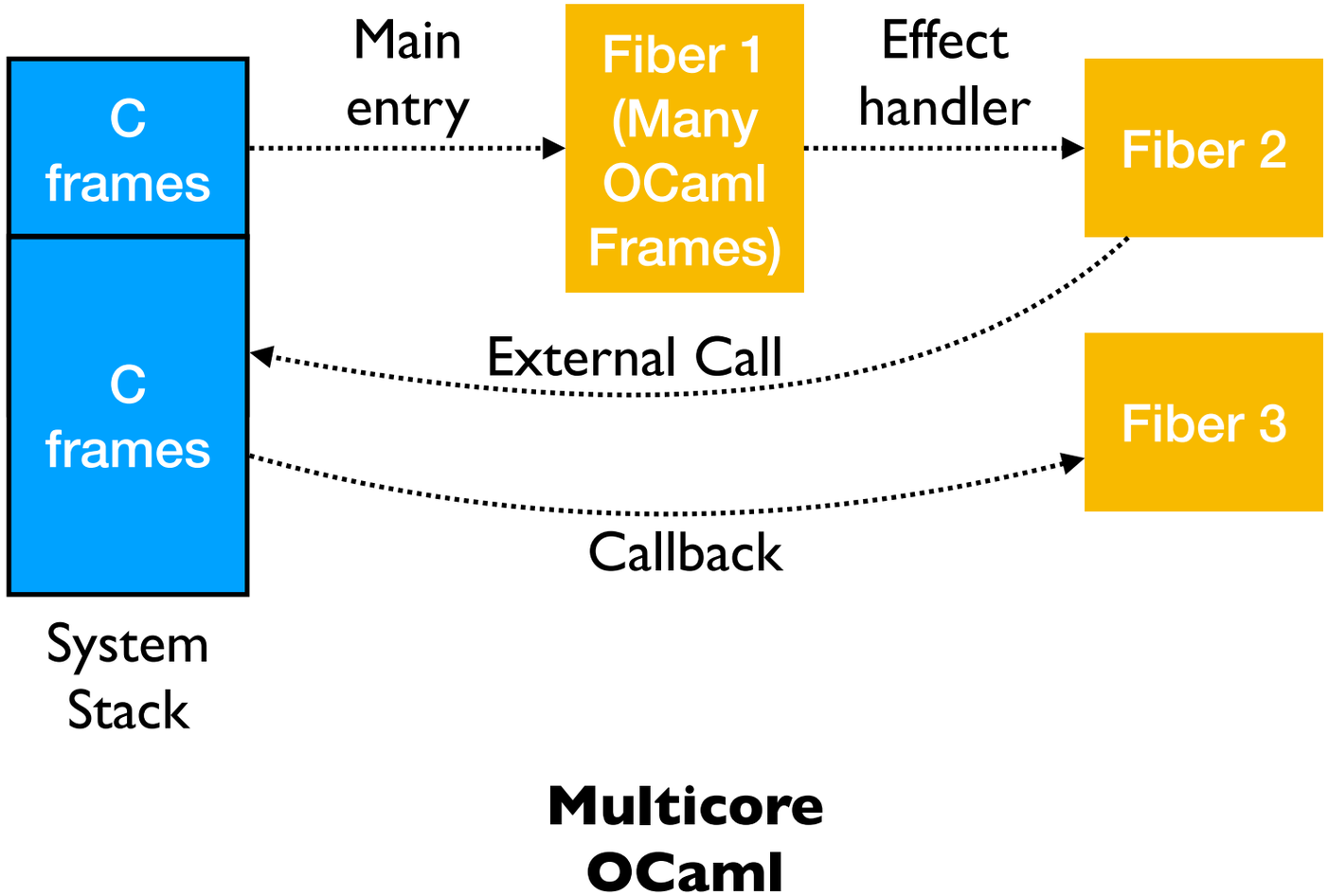
Representing Stacks & Continuations

- A stack of runtime-managed, *dynamically growing* stack segments
  - ✦ No pointers into OCaml stack
  - ✦ Need stack overflow checks for OCaml code

Representing Stacks & Continuations

- A stack of runtime-managed, *dynamically growing* stack segments
  - ✦ No pointers into OCaml stack
  - ✦ Need stack overflow checks for OCaml code
- Switch to system stack for C calls

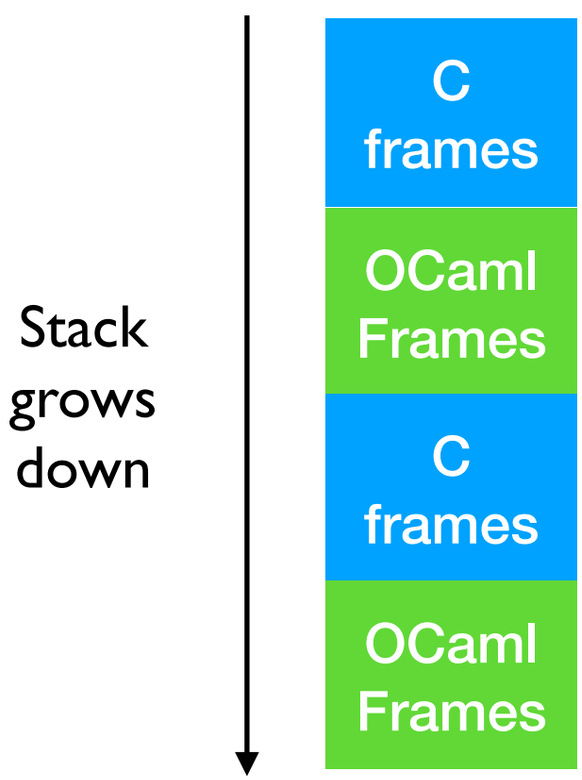




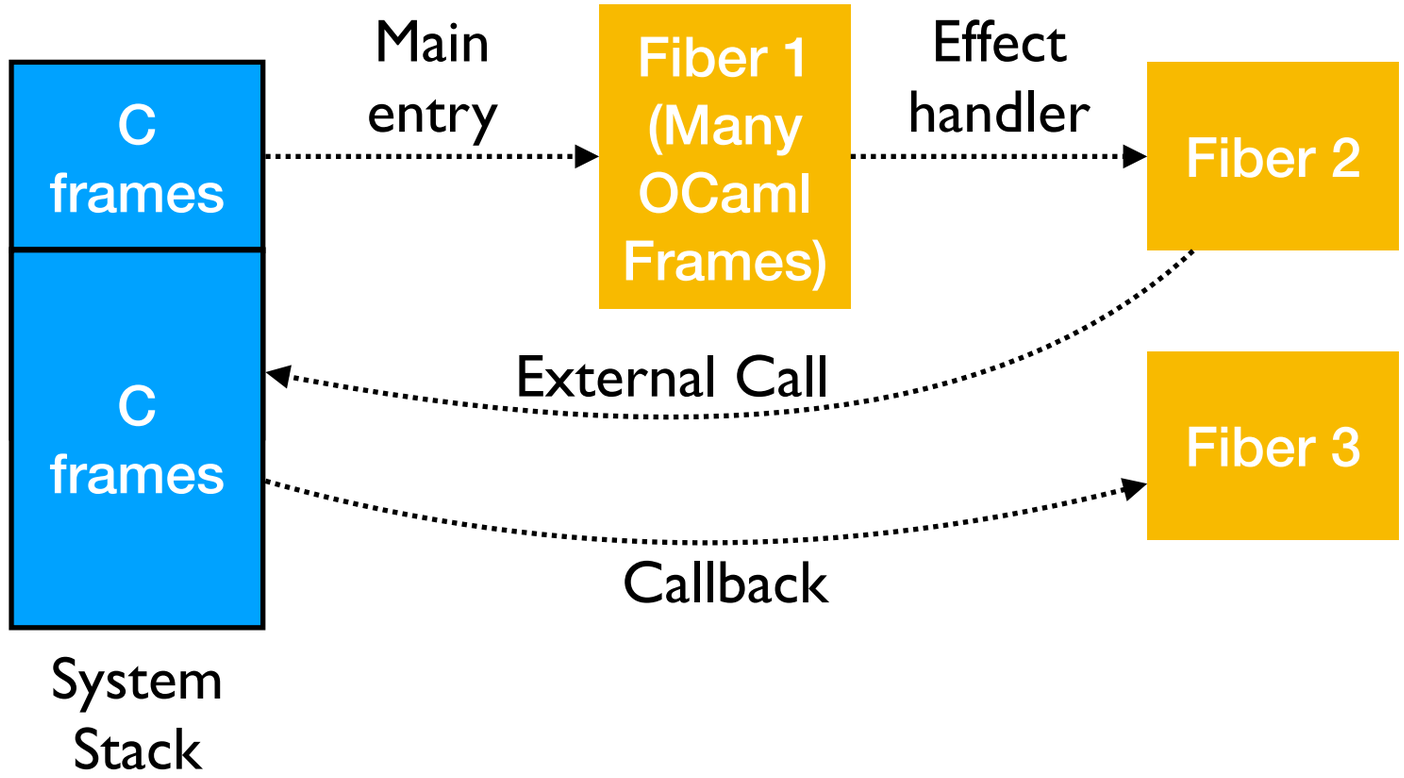
Representing Stacks & Continuations

- A stack of runtime-managed, *dynamically growing* stack segments
  - ✦ No pointers into OCaml stack
  - ✦ Need stack overflow checks for OCaml code
- Switch to system stack for C calls

Formal Semantics in PLDI'21 paper



Stock OCaml



Multicore OCaml

Switching stacks fast

- *One-shot* — capture and resumption does not involve copying frames

Switching stacks fast

- *One-shot* — capture and resumption does not involve copying frames
- No callee-saved registers in OCaml
  - ✦ Switching between stacks need not save & restore register state

Performance

```
let foo () =
    (* a *)
  try
    (* b *)
    perform E
    (* d *)
  with effect E k ->
    (* c *)
    continue k ()
(* e *)
```

Performance

```
let foo () =
    (* a *)
  try
    (* b *)
    perform E
    (* d *)
  with effect E k ->
    (* c *)
    continue k ()
(* e *)
```

Instruction Sequence	Significance
a to b	Create a new stack & run the computation
b to c	Performing & handling an effect
c to d	Resuming a continuation
d to e	Returning from a computation & free the stack

- Each of the instruction sequences involves a stack switch

Performance

```
let foo () =
    (* a *)
  try
    (* b *)
    perform E
    (* d *)
  with effect E k ->
    (* c *)
    continue k ()
(* e *)
```

Instruction Sequence	Significance
a to b	Create a new stack & run the computation
b to c	Performing & handling an effect
c to d	Resuming a continuation
d to e	Returning from a computation & free the stack

- Each of the instruction sequences involves a stack switch
- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ✦ Cost measured using Intel PT's cycle accurate tracing
  - ✦ For calibration, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

Performance

```
let foo () =
    (* a *)
  try
    (* b *)
    perform E
    (* d *)
  with effect E k ->
    (* c *)
    continue k ()
(* e *)
```

Instruction Sequence	Significance	Time (ns)
a to b	Create a new stack & run the computation	23
b to c	Performing & handling an effect	5
c to d	Resuming a continuation	11
d to e	Returning from a computation & free the stack	7

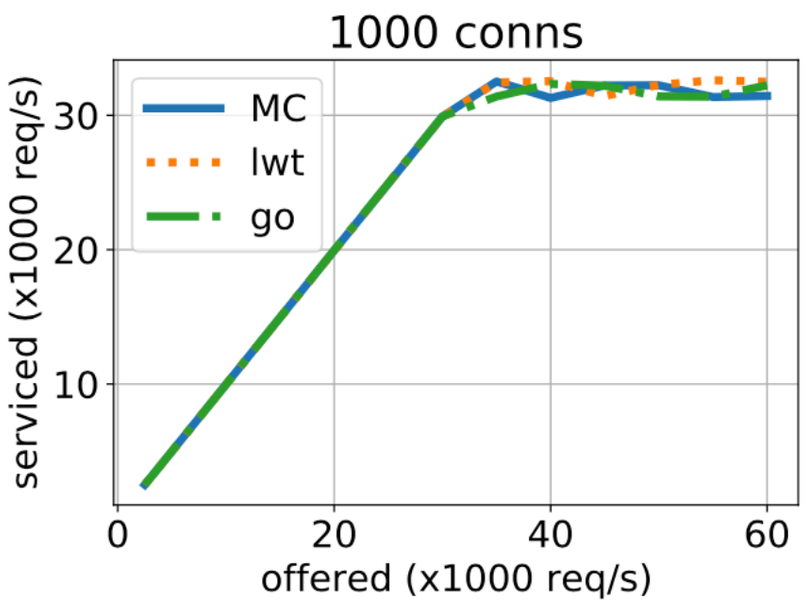
- Each of the instruction sequences involves a stack switch
- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ✦ Cost measured using Intel PT's cycle accurate tracing
  - ✦ For calibration, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

Performance: WebServer

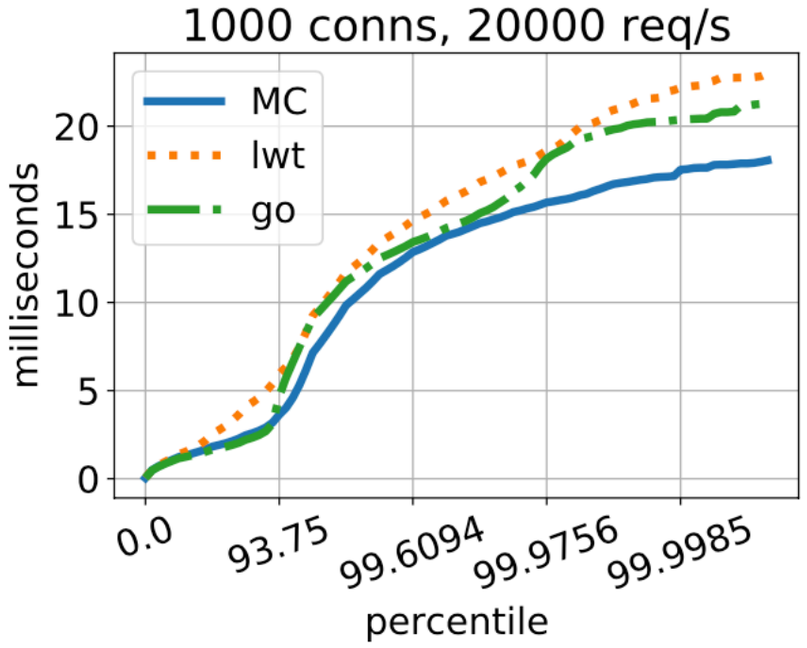
- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)
- Performance measured using wrk2

Performance: WebServer

- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)
- Performance measured using wrk2



(a) Throughput



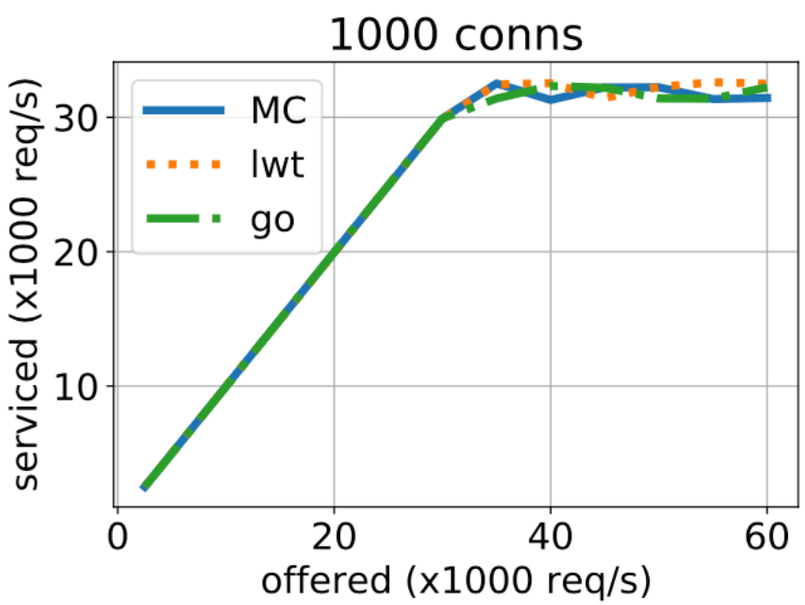
(b) Tail latency

Performance: WebServer

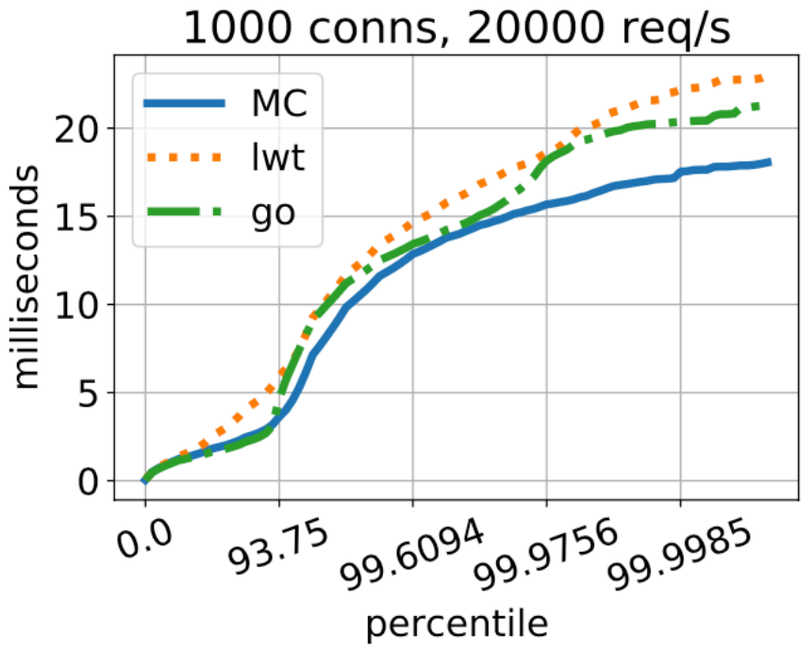
- Effect handlers for asynchronous I/O in direct-style
  - ✦ https://github.com/kayceesrk/ocaml-aeio
- Variants
  - ✦ **Go** + net/http (GOMAXPROCS=1)
  - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
  - ✦ OCaml + http/af + Effect handlers (**MC**)

- Direct style (no monadic syntax)
- Can use OCaml exceptions!
- Backtrace per thread (request)
- gdb & perf work!

- Performance measured using wrk2



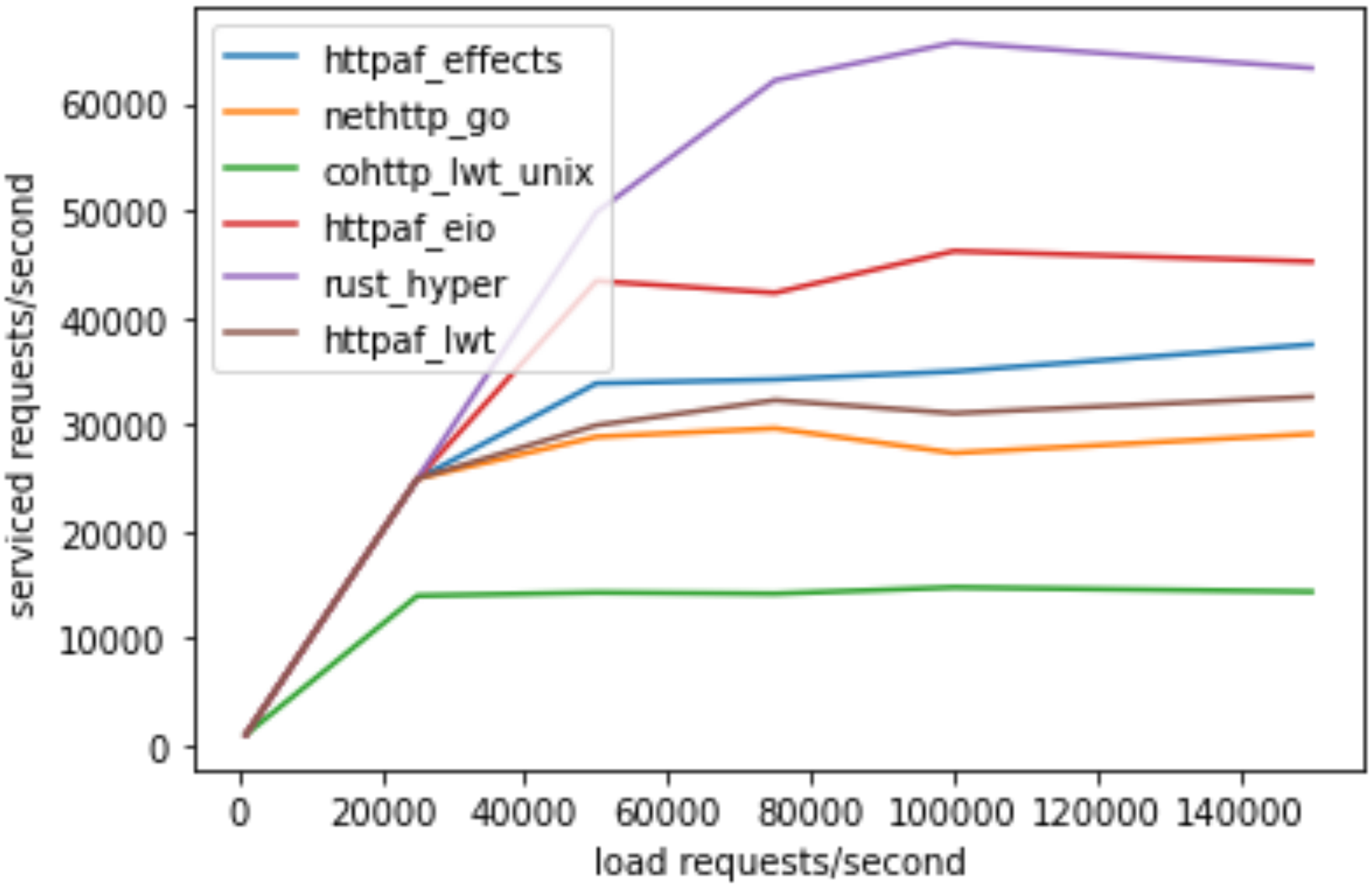
(a) Throughput



(b) Tail latency

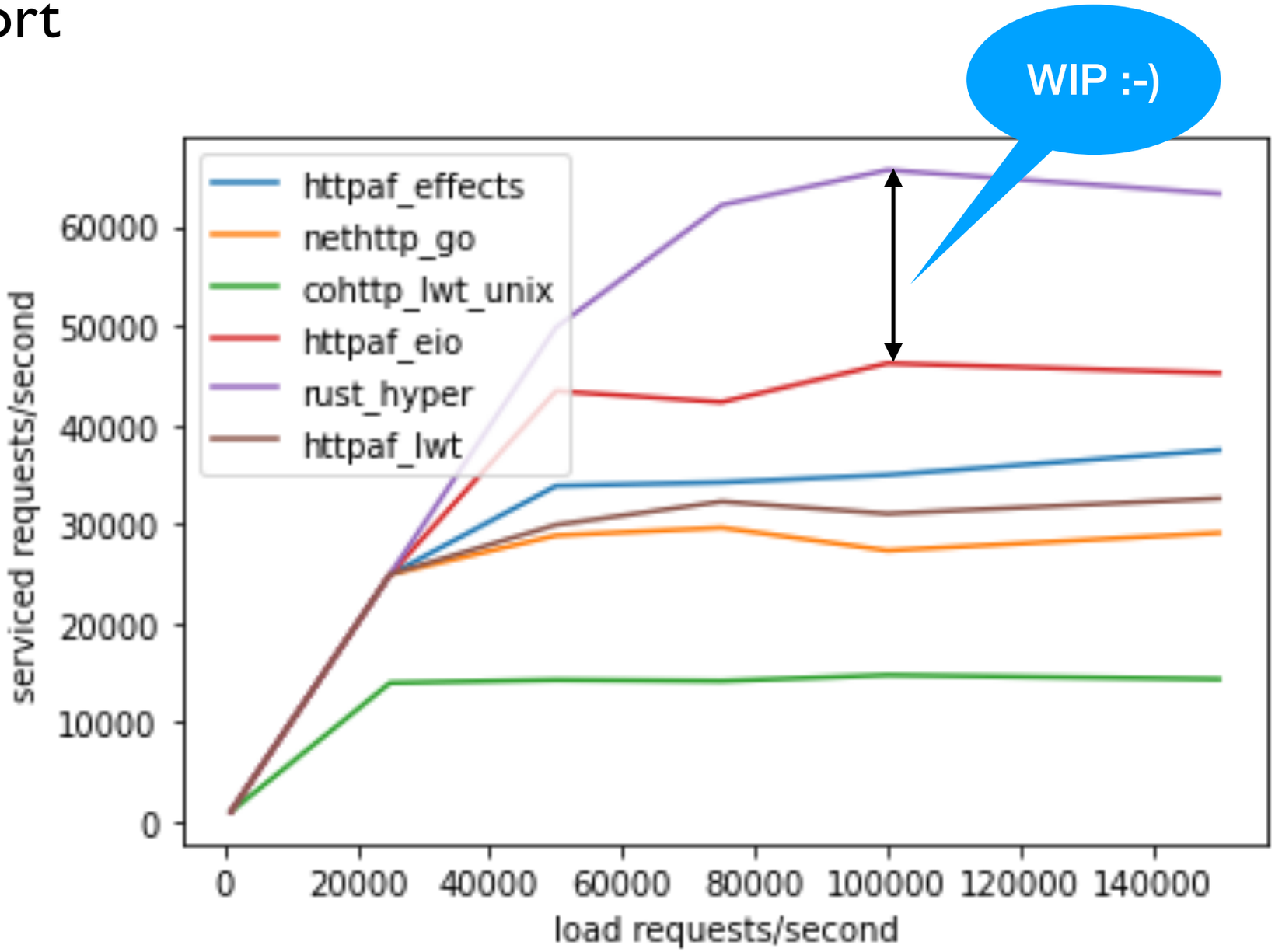
Performance: WebServer

- **eio**: effects-based I/O over Linux kernel's new `io_uring` support



Performance: WebServer

- **eio**: effects-based I/O over Linux kernel's new `io_uring` support



Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.

Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources

Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources

```
let copy ic oc =
  let rec loop () =
    let l = input_line ic in
    output_string oc (l ^ "\n");
    loop ()
  in
  try loop () with
  | End_of_file  -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources

raises `End_of_file` at the end

```
let copy ic oc =
  let rec loop () =
    let l = input_line ic in
    output_string oc (l ^ "\n");
    loop ()
in
try loop () with
| End_of_file  -> close_in ic; close_out oc
| e -> close_in ic; close_out oc; raise e
```

Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources

raises `End_of_file` at the end

raise `Sys_error` when channel is closed

```
let copy ic oc =
  let rec loop () =
    let l = input_line ic in
    output_string oc (l ^ "\n");
    loop ()
  in
  try loop () with
  | End_of_file  -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

Backwards Compatibility

- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources

raises `End_of_file` at the end

raise `Sys_error` when channel is closed

```
let copy ic oc =
  let rec loop () =
    let l = input_line ic in
    output_string oc (l ^ "\n");
    loop ()
  in
  try loop () with
  | End_of_file  -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

We would like to make this code transparently asynchronous

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit
```

Asynchronous IO

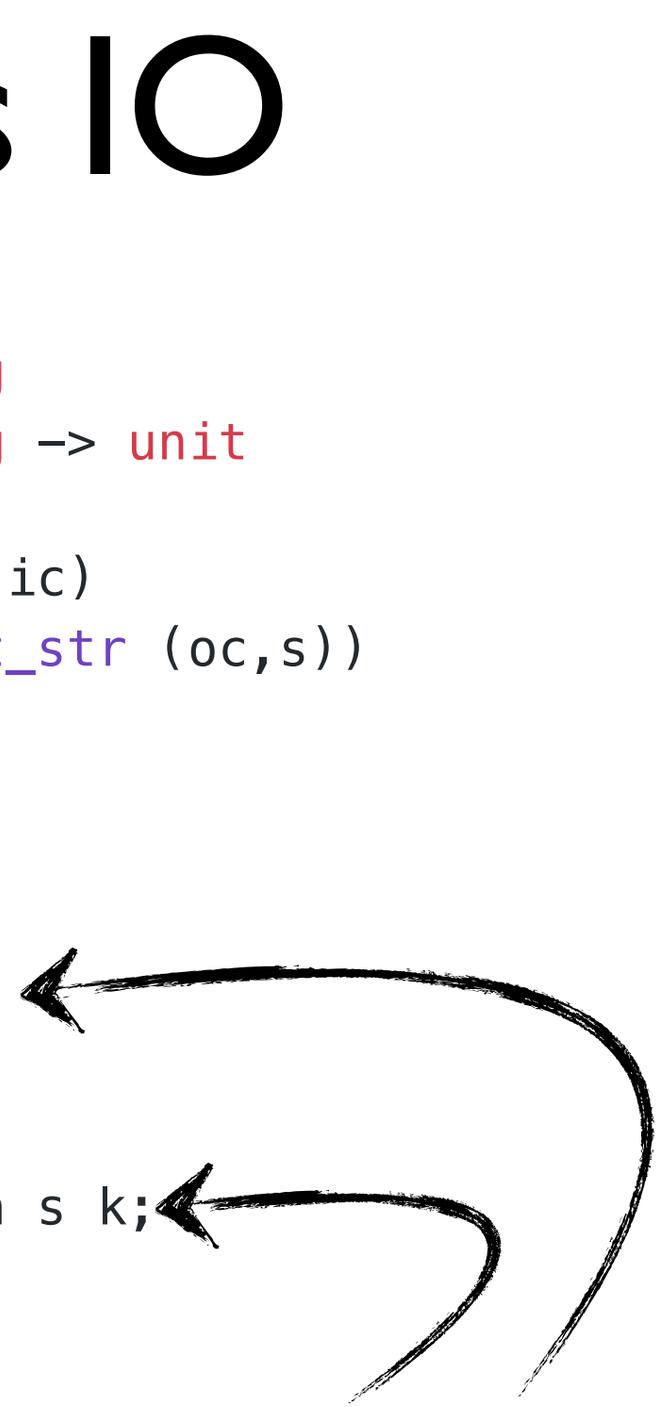
```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc,s))
```

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc,s))

let run_aio f = match f () with
| v -> v
| effect (In_line chan) k ->
    register_async_input_line chan k;
    run_next ()
| effect (Out_str (chan, s)) k ->
    register_async_output_string chan s k;
    run_next ()
```

Asynchronous IO

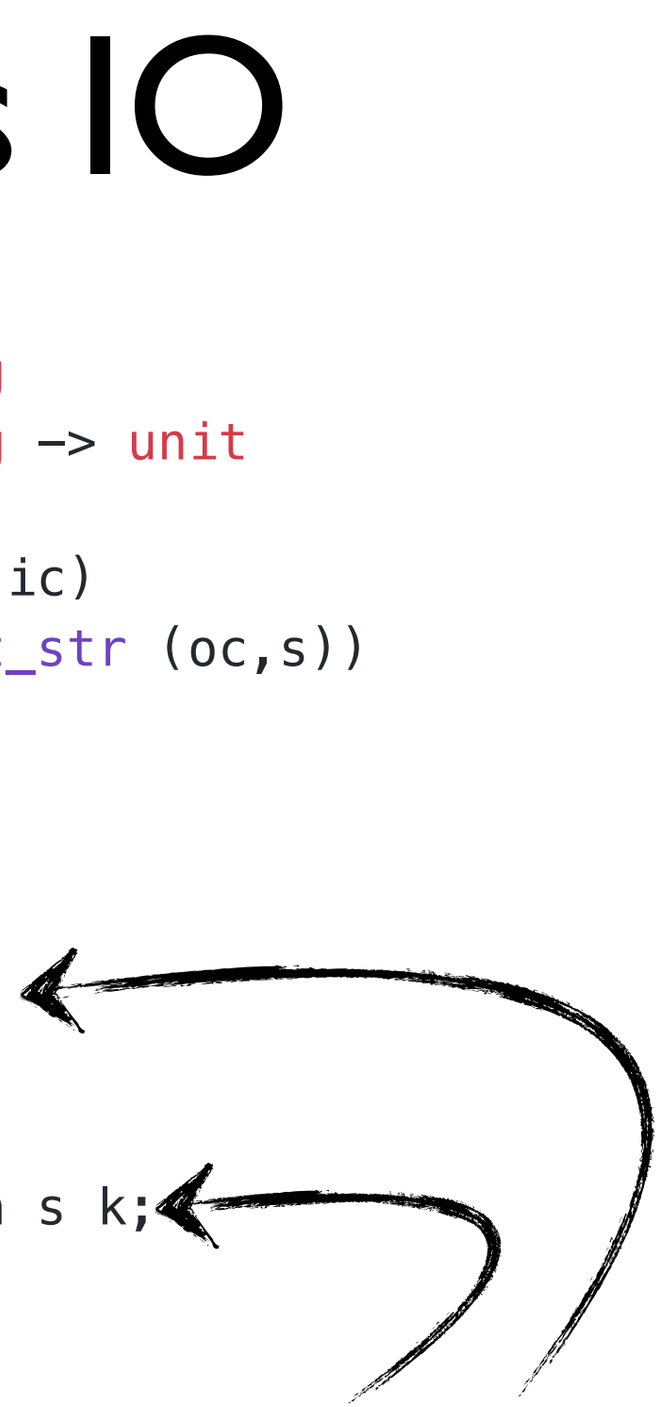
```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc,s))

let run_aio f = match f () with
| v -> v
| effect (In_line chan) k ->
    register_async_input_line chan k;
    run_next ()
| effect (Out_str (chan, s)) k ->
    register_async_output_string chan s k;
    run_next ()
```



- Continue with appropriate *value* when the asynchronous IO call returns

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc,s))

let run_aio f = match f () with
| v -> v
| effect (In_line chan) k ->
    register_async_input_line chan k;
    run_next ()
| effect (Out_str (chan, s)) k ->
    register_async_output_string chan s k;
    run_next ()
```



- Continue with appropriate *value* when the asynchronous IO call returns
- But what about termination? — `End_of_file` and `Sys_error` *exceptional* cases.

Discontinue

```
discontinue k End_of_file
```

- We add a discontinue primitive to resume a continuation by raising an exception
- On `End_of_file` and `Sys_error`, the asynchronous IO scheduler uses discontinue to raise the appropriate exception

Linearity

- Resources such as sockets, file descriptors, channels and buffers are *linear* resources
  - ✦ Created and destroyed *exactly once*

Linearity

- Resources such as sockets, file descriptors, channels and buffers are *linear* resources
  - ✦ Created and destroyed *exactly once*
- OCaml functions return *exactly once* with *value* or *exception*
  - ✦ Defensive programming already guards against exceptional return cases

Linearity

- Resources such as sockets, file descriptors, channels and buffers are *linear* resources
  - ✦ Created and destroyed *exactly once*
- OCaml functions return *exactly once* with *value* or *exception*
  - ✦ Defensive programming already guards against exceptional return cases
- With effect handlers, functions may return *at-most once* if continuation not resumed
  - ✦ This breaks resource-safe legacy code

Linearity

```
effect E : unit
let foo () = perform E
```

Linearity

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e -> close_in ic; raise e
```

Linearity

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e -> close_in ic; raise e
```

```
let baz () =
  try bar () with
  | effect E _ -> () (* leaks ic *)
```

Linearity

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e -> close_in ic; raise e
```

```
let baz () =
  try bar () with
  | effect E _ -> () (* leaks ic *)
```

We *assume* that captured continuations are resumed *exactly once*
either using `continue` or `discontinue`

Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
  - ✦ DWARF stack unwinding support

Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
  - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

Backtraces

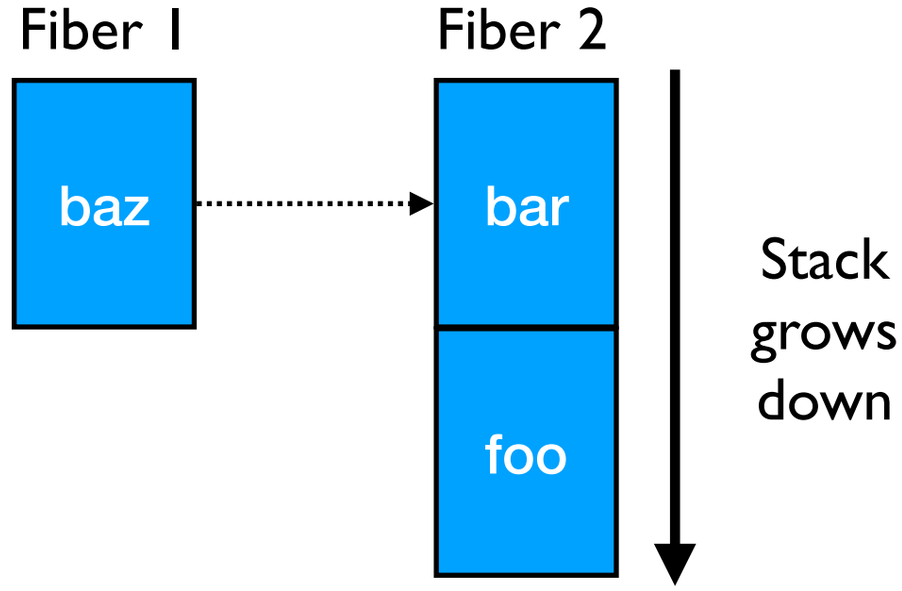
- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
  - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e ->
      close_in ic; raise e

let baz () =
  try bar () with
  | effect E _ -> () (* leak *)
```

Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
  - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

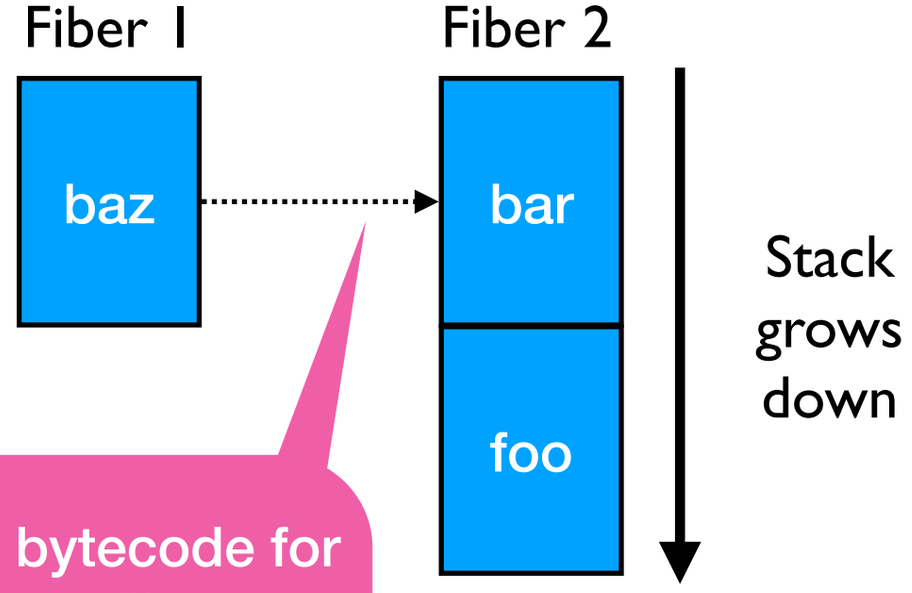
```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e ->
      close_in ic; raise e

let baz () =
  try bar () with
  | effect E _ -> () (* leak *)
```



Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
  - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e ->
      close_in ic; raise e

let baz () =
  try bar () with
  | effect E _ -> () (* leak *)
```



Backtraces

```
effect E : unit
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e ->
      close_in ic; raise e

let baz () =
  try bar () with
  | effect E _ -> () (* leak *)
```

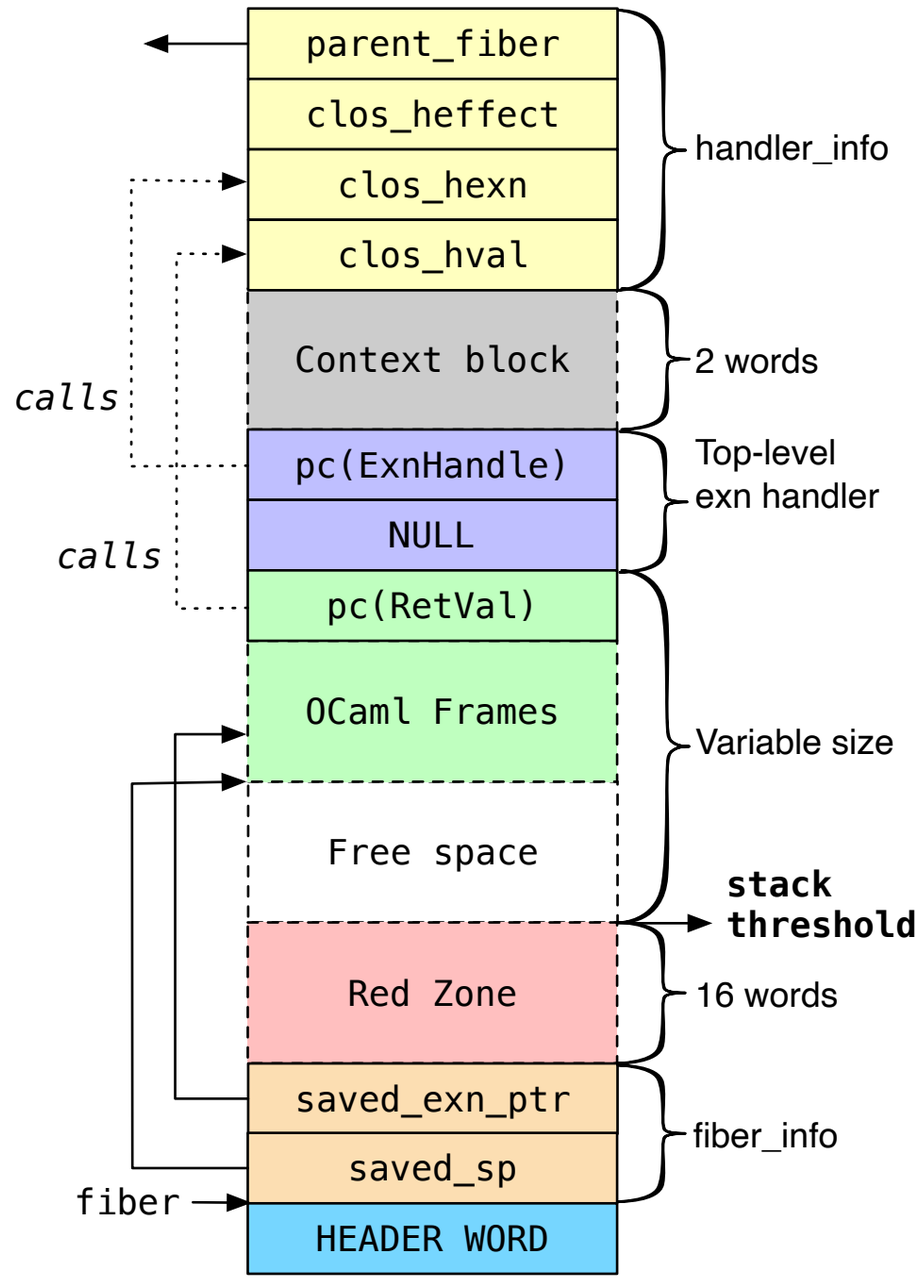
```
(lldb) bt
* thread #1, name = 'a.out', stop reason = …
  * #0: 0x58b208 caml_perform
    #1: 0x56aa5d camlTest__foo_83 at test.ml:4
    #2: 0x56aae2 camlTest__bar_85 at test.ml:9
    #3: 0x56a9fc camlTest__fun_199 at test.ml:14
    #4: 0x58b322 caml_runstack + 70
    #5: 0x56ab99 camlTest__baz_91 at test.ml:14
    #6: 0x56ace6 camlTest__entry at test.ml:21
    #7: 0x56a41c caml_program + 60
    #8: 0x58b0b7 caml_start_program + 135
    #9: …
```

Thanks!

- Multicore OCaml
  - ✦ https://github.com/ocaml-multicore/ocaml-multicore
- Effects Examples
  - ✦ https://github.com/ocaml-multicore/effects-examples
- Sivaramakrishnan et al, "*Retrofitting Effect Handlers onto OCaml*", PLDI 2021
  - ✦ https://arxiv.org/abs/2104.00250

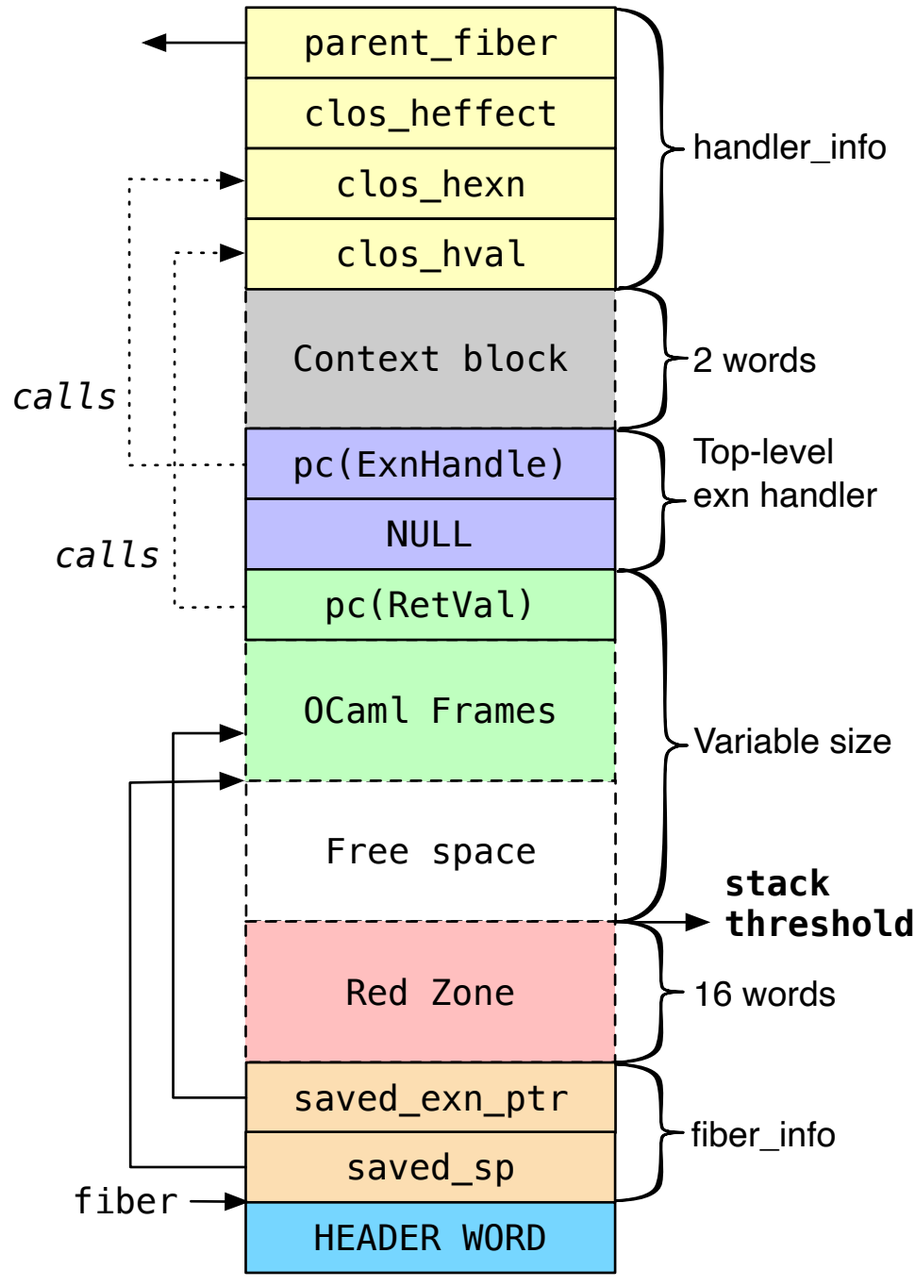
Bonus Slides

Fiber Layout



Fiber Layout

```
match f () with
| v -> ...
| exception X1 -> ...
| exception (X2 v) -> ...
| effect E1 k -> ...
| effect E2 k -> ...
```



“KC” Sivaramakrishnan

- Who am I?
  - ✦ Asst Prof at IIT Madras, India
  - ✦ Lead the development of Multicore OCaml project
- Interested in learning
  - ✦ Compiling effect handlers for uncooperative environments (Wasm, Java, C, JavaScript)
  - ✦ Pragmatic effect systems
  - ✦ New use cases for effects
- Talks
  - ✦ Retrofitting effect handlers onto OCaml (30 minutes)
  - ✦ ParaFuzz: Fuzzing Multicore OCaml programs (15 minutes)