Fiber Resource Management and Deadlocks in Multicore OCaml

Surendar
cs24m050@smail.iitm.ac.in

Dr. K. C. Sivaramakrishnan
kcsrk@cse.iitm.ac.in

Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Introduction

OCaml 5 introduces algebraic effect handlers with multicore support, enabling fiber-based concurrency via captured continuations. This thesis extends the OCaml runtime to track continuations and suspended fibers, and studies their reachability and lifetime. In addition, the thesis reviews existing runtime-level techniques for deadlock detection, particularly those used in the Go runtime. Several motivating examples are constructed to illustrate memory leaks, resource leaks, and deadlock scenarios in effect-based concurrent systems.

Effect Handler

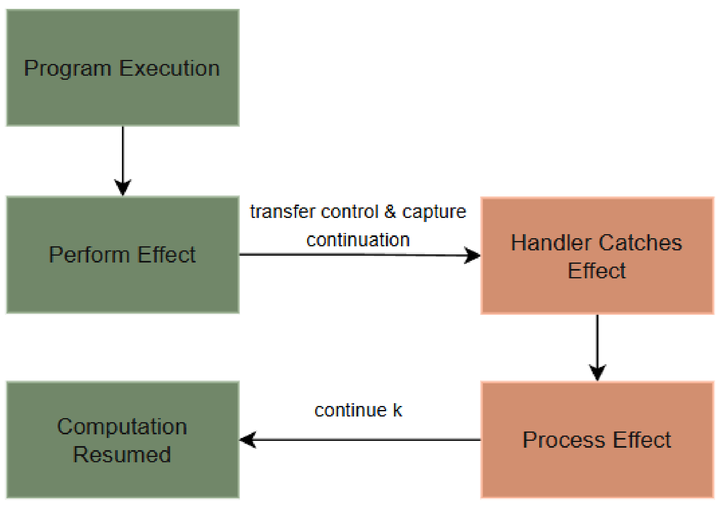

Fig. 1: Effect handling workflow

When an effect is triggered, execution is suspended and control is transferred to the nearest handler, along with the current continuation. The handler may choose to resume the suspended computation, delay its resumption, or discard it, allowing expressive and lightweight control flow mechanisms.

Memory & Resource Leak Issue

```
open Effect

type _ Effect.t += Pause : unit t

let use_resource path =
  let fd = Unix.openfile path [Unix.O_RDONLY
      ] 0 in
  let buf = Bytes.create 1024 in
  leaked_ref := Some buf;
  match perform Pause with
  | () ->
      Unix.close fd
  | effect Pause k ->
      ()
```

The code intentionally suspends execution after acquiring a resource and allocating memory, preventing the cleanup and release steps from running. This serves as a motivating example of how abandoned continuations can lead to both memory and resource leaks in effect-based programs.

Garbage Collector

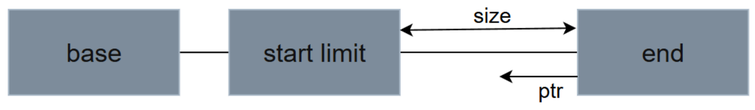

Fig. 2: Minor heap layout and allocation pointers

The OCaml runtime uses a generational, incremental garbage collector to reclaim objects no longer reachable from program roots. Short-lived values are allocated in the minor heap using a bump-pointer strategy and collected via fast copying, while long-lived objects are managed incrementally in the major heap across multiple domains in Multicore OCaml for scalable concurrency.

Runtime Extensions to the GC

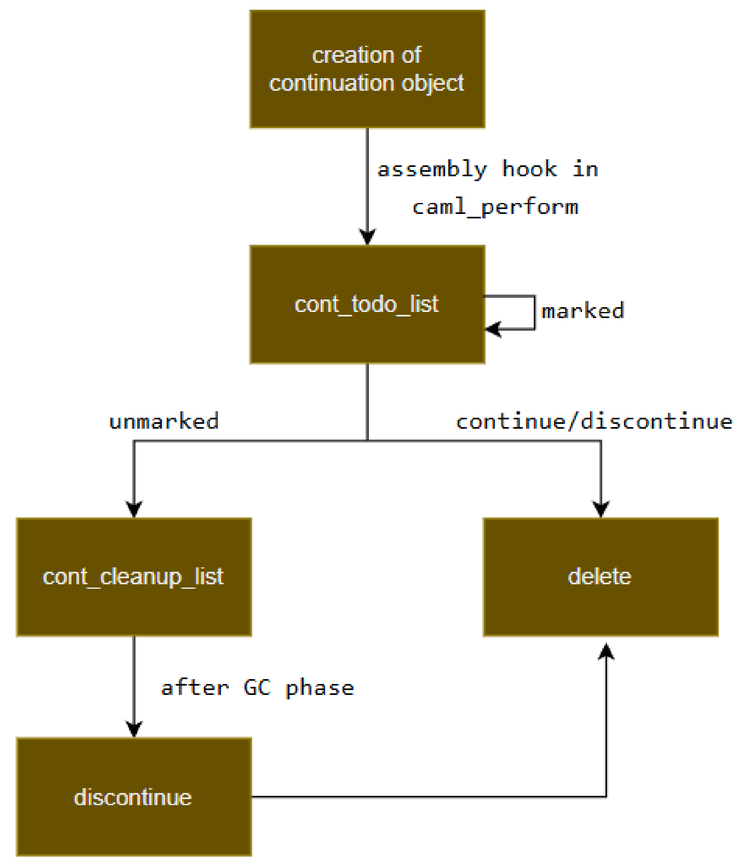

Fig. 3: Continuation tracking lifecycle

Suspended continuations are initially recorded in the `cont_todo_list`. When such continuations become unreachable, they are transferred to the `cont_cleanup_list`, where they are discontinued using `runtime_discontinue` during GC.

Results And Observations

```
[00] cont_ll: Starting mark_and_shift_toclean
[00]   Keeping marked cont in todo: 0x78d7105f8188
[00]   Moving unmarked cont to toclean: 0x78d7105f8168
[00]   Moving unmarked cont to toclean: 0x78d7105f8148
[00]   Removing Used cont: 0x78d7105f8128
[00] cont_ll: Finished mark_and_shift_toclean
[00] cont_ll: Processing toclean list with discontinue
[00]   Discontinuing cont: 0x78d7105f8148
[00]   Discontinuing cont: 0x78d7105f8168
[00] cont_ll: Finished discontinuing toclean list
[00] cont_ll: Starting mark_and_shift_toclean
[00]   Keeping marked cont in todo: 0x78d7105f8188
[00] cont_ll: Finished mark_and_shift_toclean
[00] cont_ll: Processing toclean list with discontinue
[00] cont_ll: Finished discontinuing toclean list
```

Fig. 4: Runtime debug output

The experiment shows that reachable continuations are correctly marked and retained, while resumed or explicitly terminated continuations are removed from tracking. Continuations that become unreachable during the major GC mark phase are automatically identified and safely discontinued after GC completes.

Deadlock Detection In GO

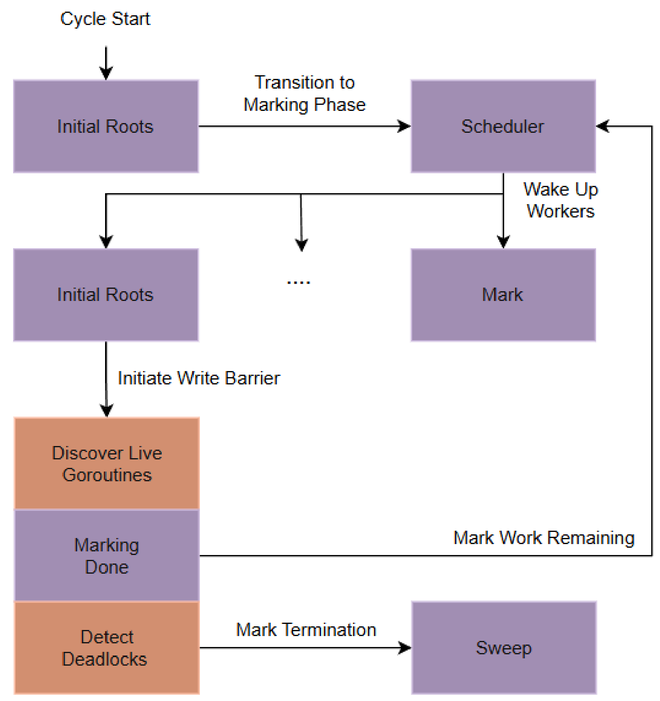

Fig. 5: GC-based deadlock detection workflow

Figure 5 shows how partial deadlock detection is integrated into the Go garbage collection cycle. Alongside standard GC phases such as marking and sweeping, the runtime performs additional analysis to identify goroutines that are blocked.

Fiber-Level Deadlock

```
open Eio.Std
open Effect

type _ Effect.t += E : unit t

let f () =
  Eio_main.run @@ fun _env ->
    Switch.run @@ fun sw ->
      Fiber.fork ~sw (fun () ->
        let lock_a = Mutex.create () in
        Gc.finalise_last
          (fun _ -> Printf.printf "Finalized
              !\n%!")
          lock_a;
        perform E
      )

let g () =
  try f () with
  | effect E, _ -> ()

let () =
  g ();
  Gc.full_major ()
```

A fiber allocates a resource and installs a GC finalizer, then suspends indefinitely by performing an unhandled effect. The surrounding computation catches the effect and resumes execution, allowing the switch to complete. When a full major GC is triggered, the finalizer executes, demonstrating that the fiber-local allocation became unreachable.

References

[1] K. C. Sivaramakrishnan et al., "Retrofitting Effect Handlers onto OCaml," PLDI, 2021.

[2] K. C. Sivaramakrishnan, "Effective Concurrency with Algebraic Effects in OCaml 5.0," Lambda Days, 2021.

[3] Gabriele Saioc et al., "Dynamic Partial Deadlock Detection and Recovery via Garbage Collection," ASPLOS, 2025.