Retrofitting a Concurrent GC onto OCaml

KC Sivaramakrishnan

University of Cambridge

OCaml Labs
OCaml

industrial-strength, pragmatic, functional programming language

- Functional core with imperative and object-oriented features
- Native (x86, ARM, ...), JavaScript, JVM

Hindley-Milner Type Inference
Powerful module system

Facebook: REASON, Infer, flow, Hack, PXRE

Microsoft: Project Everest, Jane Street

The Coq Proof Assistant, MIRAGE OS
OCaml

industrial-strength, pragmatic, functional programming language

• Functional core with imperative and object-oriented features
• Native (x86, ARM, ...), JavaScript, JVM

No multicore support!

Facebook: REASON Infer flow Hack
Microsoft: Project Everest Jane Street
The Coq Proof Assistant
Multicore OCaml

- Native support for concurrency and parallelism in OCaml
- Lead from OCaml Labs, University of Cambridge
  - Collaborators Stephen Dolan (OCaml Labs), Leo White (Jane Street)
- Expected to hit mainline in late 2019
- In this talk,
  - Overview of Multicore GC, with a few deep dives
Multicore OCaml GC: Desiderata

- **Code backwards compatibility**
  - Do not break existing code

- **Performance backwards compatibility**
  - Do not slow down existing programs

- **Minimise pause times**
  - Latency is more important than throughput

- **Performance predictability and stability**
  - Slow and stable better than fast but unpredictable

- **Minimize knobs**
  - 90% of programs should run at 90% peak performance by default
Outline

• Difficult to appreciate GC choices in isolation

• Begin with a GC for a sequential purely functional language
  ✦ Gradually add mutations, parallelism and concurrency
Sequential purely functional

- Stop-the-world mark and sweep
- Tri-color marking
  - States: White (Unmarked), Grey (Marking), Black (Marked)
  - White $\rightarrow$ Grey (mark stack) $\rightarrow$ Black
- Mark stack is empty $\Rightarrow$ done marking
  - Tri-color invariant: *No black object points to a white object*
- Sweeping: walk the heap and free white objects
Sequential purely functional

- **Pros**
  - Simple
  - Can perform the GC incrementally
    - ...|←mutator→|←mark→|←mutator→|←mark→|←mutator→|←sweep→...

- **Cons**
  - Need to maintain free-list of objects => allocations overheads + fragmentation
Generational GC

- Generational Hypothesis
  - Young objects are much more likely to die than old objects

- Minor heap collected by copying collection
  - Survivors promoted to major heap
  - Only touches live objects (typically, < 10% of total)

- Roots are registers and stack
  - purely functional => no pointers from major to minor
Mutations

• OCaml does not prohibit mutations
  ✦ Mutable references, Arrays…

• Encourages it with syntactic support!

```ocaml
type client_info =
  { addr: Unix.inet_addr;
    port: int;
    user: string;
    credentials: string;
    mutable last_heartbeat_time: Time.t;
    mutable last_heartbeat_status: string;
  }

let handle_heartbeat cinfo time status =
  cinfo.last_heartbeat_time <- time;
  cinfo.last_heartbeat_status <- status
```

• Mutations are pervasive in real-world code
Mutations

- Load immutable field
- Initialising store
- Load mutable field
- Assignment

Memory Access Distribution (%)

almbench (29.4)
ssetrip (106.2)
setstrip-smallbuf (119.6)
levinson-durbin (154.8)
cpdf_transform (37.46)
jsontrip-sample (145.49)
minlight (156.1)
cpdf-squeeze (69.38)
cpdf-reformat (77.59)
simple-access (39.39)
lu-decomposition (144.24)
naive-multilayer (146.33)
levit-g2vp (65.67)

Load immutable field
Initialising store
Load mutable field
Assignment

less functional

more functional
Mutations — Minor GC

- Old objects might point to young objects
- Must know those pointers for minor GC
  - (Naively) scan the major GC for such pointers
- Intercept mutations with write barrier
  ```
  (* Before r := x *)
  let write_barrier (r, x) =
    if is_major r && is_minor x then
      remembered_set.add r
  ```
- Remembered set
  - Set of major heap addresses that point to minor heap
  - Used as root for minor collection
  - Cleared after minor collection.
Mutations — Major GC

- Mutations are problematic if both conditions hold:
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

- Insertion/Dijkstra/Incremental barrier prevents 1

- Deletion/Yuasa/snapshot-at-beginning prevents 2

(* Before $r := x$ *)

```plaintext
let write_barrier (r, x) =
  if is_major r && is_minor x then
    remembered_set.add r
  else if is_major r && is_major x then
    mark(!r)
```

```plaintext

```
Parallelism — Minor GC

- Domain.spawn : (unit -> unit) -> unit

- Invariant: Minor heap objects are only accessed by owning domain

- Doligez-Leroy POPL’93
  - No pointers between minor heaps
  - No pointers from major to minor heaps

- Before r := x, if is_major(r) && is_minor(x), then promote(x).

- Too much promotion. Ex: work-stealing queue
Parallelism — Minor GC

- Weaker invariant
  - No pointers between minor heaps
  - Objects in foreign minor heap are not accessed directly

- Read barrier. If the value loaded is
  - integers, object in shared heap or own minor heap => continue
  - object in foreign minor heap => Read fault (Interrupt + promote)
Efficient read barrier check

- Given $x$, is $x$ an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)
- Careful VM mapping + bit-twiddling
- Example: 16-bit address space, $0xPQRS$

- Minor area: $0x4200$ – $0x42ff$
- Domain 0 : $0x4220$ – $0x422f$
- Domain 1 : $0x4250$ – $0x425f$
- Domain 2 : $0x42a0$ – $0x42af$
- Reserved : $0x4300$ – $0x43ff$

- Integer $\text{lsb}(S) = 0x1$, Minor $PQ = 0x42$, $R$ determines domain
- Compare with template $y$, where $y$ lies within minor heap
  - $allocation$ $pointer!$
  - On amd64, allocation pointer is in $r15$ register
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# ZF set => foreign minor

Integer
# lsb(%rax) = 1
xor %r15, %rax
# lsb(%rax) = 1
sub 0x0010, %rax
# lsb(%rax) = 1
test 0xff01, %rax
# ZF not set

Shared heap
# PQ(%r15) != PQ(%rax)
xor %r15, %rax
# PQ(%rax) > 1
sub 0x0010, %rax
# PQ(%rax) is non-zero
test 0xff01, %rax
# ZF not set
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# ZF set => foreign minor

Own minor heap
# PQR(%r15) = PQR(%rax)
xor %r15, %rax
# PQR(%rax) is zero
sub 0x0010, %rax
# PQ(%rax) is non-zero
test 0xff01, %rax
# ZF not set

Foreign minor heap
# PQ(%r15) = PQ(%rax)
# R(%r15) != R(%rax)
# lsb(%r15) = lsb(%rax) = 0
xor %r15, %rax
# R(%rax) is non-zero
# PQ(%rax) = lsb(%rax) = 0
sub 0x0010, %rax
# PQ(%rax) = lsb(%rax) = 0
test 0xff01, %rax
# ZF set

Read fault
Parallelism — Major GC

• OCaml’s GC is *incremental*

![Diagram showing mutator and GC in parallel for multiple domains]

• Multicore OCaml’s GC needs to be *concurrent (and incremental)*
  
  *Parallel collectors have high latency budget*
Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently

- In Multicore OCaml,
  - States
    - Unmarked
    - Marked
    - Garbage
    - Free
  - Domains alternate between mutator and gc thread
  - Marking: Unmarked → Marked
    - Sweeping: Garbage → Free
  - Marking is *racy* but *idempotent*

- Marking & Sweeping done ⇒ stop-the-world
Concurrency

- **Fibers**: vm-threads, *linear* delimited continuations
- Stack segments managed on the heap
- Every fiber has a *unique* reference from a continuation object
  - *Fibers freed when continuations are swept*
- No write barriers on fiber stack operations (*push & pop*)
Concurrence — Minor GC

• Fibers may point to minor heap objects
  ✦ which fibers to scan among 1000s? (no write barriers on fiber stacks)

• Fresh continuation object for every fiber suspension
  ✦ Continuation in minor heap => fiber suspended in current minor cycle

![Diagram showing minor heap, major heap, Cont, fiber, Linear fiber heap, and domain x connections]
Concurrencity — Minor GC

- Fibers may point to minor heap objects
  ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*

- Fresh continuation object for every fiber suspension
  ✦ *Continuation in minor heap => fiber suspended in current minor cycle*

```
minor heap (domain x)                  Linear fiber heap (domain x)
```

```
Fibers may point to minor heap objects

which fibers to scan among 1000s? (no write barriers on fiber stacks)

Fresh continuation object for every fiber suspension

Continuation in minor heap => fiber suspended in current minor cycle
```
Concurrent GC

- Fibers may point to minor heap objects
  - *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - *Continuation in minor heap => fiber suspended in current minor cycle*

![Diagram of concurrent GC]

- Minor heap (domain x)
- Linear fiber heap (domain x)
Concurrenc — Major GC

- (Multicore) OCaml uses deletion barrier
  - Fiber stack pop is a deletion \textit{(but no write barrier)}
- Before switching to unmarked fiber, \textit{complete} marking the fiber
- Marking is racy
  - \textit{For fibers, race between mutator (context switch) and gc (marking) unsafe}
Performance

• Serial performance
  ✦ Multicore benchmarking Cl: http://ocamllabs.io/multicore/

• Parallel Benchmarks
  ✦ Multicore http server, model-checker, mathematical kernels…
  ✦ Intel Core i9 (x86_64), 8 domains (parallel threads)

• Latency is our primary concern
  ✦ Minor GC pause times (trunk & multicore) = ~1-2 ms
  ✦ Avg. 50th percentile pause times = ~4 ms (1-2 ms on trunk)
  ✦ Avg. 95th percentile pause times = ~7 ms (3-4 ms on trunk)

• Throughput is easier => add more domains
Summary

• Multicore OCaml GC
  ✦ Optimise for latency first, throughput next
  ✦ Independent minor GCs + concurrent mark-and-sweep

• Various other research directions in Multicore OCaml project
  ✦ Concurrency through *Algebraic Effects and Handlers* [TFP’17]
  ✦ *OCaml Memory Model* [PLDI’18]
  ✦ *Reagents*: STM + channel communication + Hardware transactions (Intel TSX) [OCaml’16]
Questions?

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

http://kcsrk.info