# Multicore OCaml

Stephen Dolan Leo White Anil Madhavapeddy

May 24, 2014

*Currently, threading is supported in OCaml only by means of a global lock, allowing at most thread to run OCaml code at any time. We present ongoing work to design and implement an OCaml runtime capable of shared-memory parallelism.*

## 1 Introduction

Adding shared-memory parallelism to an existing language presents an interesting set of challenges. As well as the difficulties of memory management in a parallel setting, we must maintain as much backwards compatibility as practicable. This includes not just compatibility of the language semantics, but also of the performance profile, memory usage and C bindings. In the case of OCaml, users have come to rely on certain operations being cheap, and OCaml's C API exposes quite a lot of internals.

The biggest challenge is implementing the garbage collector. GC in OCaml is interesting because of pervasive immutability. Many objects are immutable, which simplifies some aspects of a parallel GC but requires the GC to sustain a very high allocation rate.

Operations on immutable objects are very fast in OCaml: allocation is by bumping a pointer, initialising writes (the only ones) are done with no barriers, and reads require no barriers. Our design is focussed on keeping these operations as fast as they are at the moment, with some compromises for mutable objects.

A previous design by Doligez et al. [1] for Caml Light was based on many thread-private heaps and a single shared heap. It maintains the invariant that there are no pointers from the shared to the private heaps. Thus, storing a pointer to a private object into the shared heap causes the private object and all objects reachable from it to be promoted to the shared heap en masse. Unfortunately this eagerly promotes many objects that were never really shared: just because an object is pointed to by a shared object does not mean another thread is actually going to attempt to access it.

Our design is similar but lazier, along the lines of the multicore Haskell work [2], where objects are promoted to the shared heap whenever another thread actually tries to access them. This has a slower sharing operation, since it requires synchronisation of two different threads, but it is performed less often.

## 2 Garbage collector overview

In our approach, the virtual machine contains a number of *domains*, each running as a separate system thread in the same address space. Each domain has a relatively small *local heap*, and a large *shared heap* is shared between all domains. The local heaps are collected with OCaml's existing minor collector (modified to use thread-local instead of global state), and long-lived objects are promoted to the shared heap.

We maintain the invariants that no domain ever follows a pointer to another domain's local heap, and that immutable fields of objects on the shared heap only point to other objects on the shared heap. Mutable fields of objects on the shared heap may point to objects on a domain's local heap, and we describe how reads of such fields are handled in Section 4.

Since local heaps are only ever accessed by a single domain, no synchronisation between threads is required when a domain collects its local heap. This allows us to sustain the high rate of allocation of short-lived objects that many OCaml programs exhibit.

# 3 Collecting the shared heap

Our GC for the shared heap is a mostly-concurrent mark-and-sweep collector. Each thread maintains its own grey stack and periodically does some marking work. Marking is by changing the GC bits in the object header, which is done without any synchronisation between threads. Multiple domains may attempt to mark the same object, but this is safe since marking is idempotent.

Each domain has a multiple-writer, single-reader message queue which it frequently polls for new messages (using the mechanism currently used to handle Unix signals). When a domain finishes marking, it triggers a stop-the-world phase, and notifies the other domains through their message queues.

During the stop-the-world phase, each thread scans its roots and marks any unmarked objects it finds. Generally, this does not involve much marking: the stop-the-world phase exists so that all threads can verify that marking has completed successfully. After the stop-the-world phase completes, any unmarked objects are garbage and available to be swept.

Our mark-and-sweep algorithm is based on the VCGC [3], a design which avoids having explicit phase transitions between "marking" and "sweeping" which are a traditional source of bugs. Our shared heap allocator is based on the Streamflow [4] design, which uses size-segmented thread-local pages. This has been shown to have good multicore behaviour and fragmentation performance.

# 4 Reading and writing

The invariants maintined by our system allow reads of immutable fields to proceed without additional overhead. However, both writes and reads to mutable fields – which can be statically distinguised in OCaml – require barriers. These are required to prevent a thread from attempting to access another thread's local heap.

Currently, OCaml has a write barrier but no read barrier for mutable fields, so the addition of a read barrier could affect the language's performance profile. However, our system has a efficient three-instruction read barrier, whose fast path consists of a single branch and no additional memory accesses.

The slow path of this barrier occurs when a domain attempts to access an object in another domain's local heap, triggering a *read fault*. In this case, the thread performing the access sends a message to the thread that owns the local heap in question using the same messaging system that triggers GC, asking it to perform the read instead. Upon receipt of such a request, a thread performs the read, copies the resulting object to the shared heap, and replies with the new shared copy.

During long-running C calls domains are unable to handle messages. In such cases domains release a lock allowing other threads to access their local heaps.

# 5 Mutable objects

Read faults cause objects to be copied from local heaps to the shared heap. If a mutable object is copied our system must ensure that all future mutable reads and writes to that object are performed on the new shared version, rather than from a stale local copy.

We have a number of options for dealing with this issue. The differences between the options are slight changes to allocation behaviour and barrier code, so we intend to implement several alternatives and compare them for performance, including:

- Allocate all mutable objects directly in the shared heap.
- Detect reads from stale local copies during the read barrier.
- Rewrite any pointers to the stale local copy during the read fault.

# References


[1] Damien Doligez and Xavier Leroy. *A concurrent, generational garbage collector for a multithreaded implementation of ML*. POPL 1993.

[2] Simon Marlow, Simon Peyton Jones, and Satnam Singh. *Runtime support for multicore Haskell*. ICFP 2009.

[3] Lorenz Huelsbergen and Phil Winterbottom. *Very concurrent mark-&-sweep garbage collection without fine-grain synchronization*. ISMM 1998.

[4] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. *Scalable, locality-conscious multithreaded memory allocation*. ISMM 2006.