Making mutable state faster: optimizing the caml_modify write barrier

29 MAY 2009
EIGENCLASS.ORG
SUBSCRIBE

As I explained yesterday, the `caml_modify` write barrier needed by OCaml's incremental + generational GC is the main reason why immutable data can often be faster than mutable state. I've played a little with it and achieved a >50% speed boost in synthetic benchmarks, which could map to up to two-digit percentage gains in actual programs (in particular, it can make `Array.map`, `init`, `make` and the likes over 20% faster). The mutable state part from the synthetic business entity update benchmark becomes 32% faster.

How the write barrier works

`caml_modify` needs to record the references from values in the major heap to the minor heap. The code is straightforward:

```
#define Modify(fp, val) do{                                                \
  value _old_ = *(fp);                                                     \
  *(fp) = (val);                                                           \
  if (Is_in_heap (fp)){                                                    \
    if (caml_gc_phase == Phase_mark) caml_darken (_old_, NULL);            \
    if (Is_block (val) && Is_young (val)                                   \
        && ! (Is_block (_old_) && Is_young (_old_))){                      \
      if (caml_ref_table.ptr >= caml_ref_table.limit){                     \
        CAMLassert (caml_ref_table.ptr == caml_ref_table.limit);           \
        caml_realloc_ref_table (&caml_ref_table);                          \
      }                                                                    \
      *caml_ref_table.ptr++ = (fp);                                        \
    }                                                                      \
  }                                                                        \
}while(0)
```

`fp` is the block slot where the (suspected) pointer is to be stored. `caml_modify` first checks if the destination is in the major heap with the `Is_in_heap` macro I'll come back to in a minute. `Is_block(v)` is true when `v` is not an immediate value. There's an interesting piece of logic that tries to avoid recording the same reference twice: if the previous reference `_old_` held in `fp` was to a block in the minor heap, we know it's been already added to the ref table and there's no need to do it again. In the worst case, when updating `N` times the same slot with values in the minor and the major heap alternatingly, the reference will be recorded `N/2` times --- it seems very unlikely this would happen in practice, though, as the ref table is cleared on each minor GC run.

The main problem with the above routine is that `Is_in_heap` is fairly slow. On 64-bit architectures it involves a lookup in a sparse hash table to see if the corresponding page is being managed by OCaml (on 32 bit, a two-level array is used). The reason why this check cannot just be implemented as a negated "is it in the minor heap" test (two pointer comparisons at most) is that it is possible to have blocks which belong neither to the minor heap nor to the major one --- iow., memory areas not managed by OCaml at all, and which are not to be traversed by the GC.

Speeding it up

When we're not in the `Phase_mark` GC phase, the order of the checks can be altered so the expensive `Is_in_heap` test is only performed when the value to be stored resides in the minor heap *and* the old one didn't before. This way, when the same slot is being updated several times, `Is_in_heap` will only be used once (with the exception explained above).

The resulting code, albeit quite ugly because of the code duplication, is considerably faster. I measure the gain with two trivial programs:

```
let () =
  let a = Array.make 1_000 0 in
    for i = 1 to 100000 do
      ignore (Array.map (fun i -> i) a)
    done
```

and

```
open Printf

let time f x =
  let t0 = Unix.gettimeofday () in
  let y = f x in
    Printf.printf "Needed %8.5fs\n" (Unix.gettimeofday () -. t0);
    y

let ntimes n f x = for i = 1 to n do f x done

type silly = A | B of int

let () =
  let f a =
    for i = 1 to Array.length a - 1 do
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
      a.(i) <- B 0;
    done
  in
    time (ntimes 100 f) (Array.make 1_000_000 A)
```

The latter is just meant to illustrate the isolate the effect of the write barrier, while the former measures the impact in higher-order functions with some overhead. Note that the array updates in the second one do not involve any allocation, as `B 0` is evaluated once when the module is initialized. A block (i.e. non-immediate) value is needed here so that the compiler generates a `caml_modify` call --- it would know there's no need to if we were using `A`.

The times are 8.3s (for loop) and 1.66s (Array.map) for an unpatched OCaml 3.11.0, and 5.3s and 1.4s with a modified runtime using this routine:

```
#define Modify(fp, val) do{                                             \
  value _old_ = *(fp);                                                  \
  *(fp) = (val);                                                        \
  if (caml_gc_phase == Phase_mark) {                                    \
    if (Is_in_heap (fp)) {                                              \
      caml_darken (_old_, NULL);                                        \
      if (Is_block (val) && Is_young (val)                              \
```

```
          && ! (Is_block (_old_) && Is_young (_old_))){                    \
        if (caml_ref_table.ptr >= caml_ref_table.limit){                   \
          CAMLassert (caml_ref_table.ptr == caml_ref_table.limit);         \
          caml_realloc_ref_table (&caml_ref_table);                        \
        }                                                                  \
        *caml_ref_table.ptr++ = (fp);                                      \
      }                                                                    \
    }                                                                      \
  } else {                                                                 \
      if (Is_block (val) && Is_young (val)                                 \
          && ! (Is_block (_old_) && Is_young (_old_)) &&                   \
          Is_in_heap (fp) ){                                               \
        if (caml_ref_table.ptr >= caml_ref_table.limit){                   \
          CAMLassert (caml_ref_table.ptr == caml_ref_table.limit);         \
          caml_realloc_ref_table (&caml_ref_table);                        \
        }                                                                  \
        *caml_ref_table.ptr++ = (fp);                                      \
      }                                                                    \
  }                                                                        \
}while(0)
```

I tried to avoid the code duplication as follows, but this proved to be slightly slower (5.6s for the for loop, no difference for Array.map):

```
#define Modify(fp, val) do{                                                \
  value in_heap = 0;                                                       \
  value _old_ = *(fp);                                                     \
  *(fp) = (val);                                                           \
  if (caml_gc_phase == Phase_mark) {                                       \
    if (Is_in_heap (fp)) {                                                 \
      caml_darken (_old_, NULL);                                           \
      in_heap = 1;                                                         \
      goto caml_modify_maybe_add_to_ref_table;                             \
    }                                                                      \
  } else {                                                                 \
      caml_modify_maybe_add_to_ref_table:                                  \
      if (Is_block (val) && Is_young (val)                                 \
          && ! (Is_block (_old_) && Is_young (_old_)) &&                   \
          (in_heap || Is_in_heap (fp)) ){                                  \
        if (caml_ref_table.ptr >= caml_ref_table.limit){                   \
          CAMLassert (caml_ref_table.ptr == caml_ref_table.limit);         \
          caml_realloc_ref_table (&caml_ref_table);                        \
        }                                                                  \
        *caml_ref_table.ptr++ = (fp);                                      \
      }                                                                    \
  }                                                                        \
}while(0)
```

---

29 May 2009 at 15:16
