When immutable data is faster: write barrier costs

28 May 2009
EIGENCLASS.ORG
Subscribe

This example shows that in some cases immutable data can be faster than mutable state. One reason is that language implementations with incremental, generational or concurrent GCs (a few combine all three properties at once, like HotSpot's; OCaml's is only generational and incremental) typically use write barriers which add overhead to each heap pointer store (read barriers are normally more expensive). In particular, generational GCs use write barriers to maintain the remembered set: a list of references from an older generation to a younger one.

The synthetic benchmark operates on "simple records intended to emulate typical business entities":

```
module P0 = struct
  type person = { name: string; age: int; balance: float }
end

module P1 = struct
  type person = { name: string; mutable age: int; mutable balance: float }
end
```

The `balance` field is updated repeatedly, by creating a new record in the immutable case, and by changing the field directly in the mutable one. `floats` are generally boxed in OCaml (the main exceptions are float arrays and records holding only floats), so a `person` record takes 6 words on x86-64 (7 on x86, since the double value takes 8 bytes). In the first case, each iteration allocates 48(28) bytes; in the second one, only 16(12) are allocated, but the field update incurs into the write barrier overhead: a call to the `caml_modify` routine, that checks whether a pointer to a value in the younger generation is being stored in a block residing in the old (major) heap. As it turns out, the extra allocation is faster than the write barrier:

```
Immutable record: 138644594.442718 updates/s
Mutable record:   79740984.161326 updates/s
```

The example can be modified to illustrate clearly the effect of the write barrier, and when the latter is needed at all.

If the `balance` field is changed to an `int`, the stored value is known not to reside in the minor heap (integers, bools, chars and constant constructors are immediate), so the write barrier is not needed, and no call to `caml_modify` will be generated. Moreover, there will be no allocation at all in the mutable case, since there are no boxed floats involved anymore, so we can compare the cost of allocating a 4-word record to that of updating a single field:

```
Immutable record: 277762346.536304 updates/s
Mutable record:   499975001.249938 updates/s
```

Similarly, the `balance` field can be "boxed manually", so to speak, to avoid allocations in the mutable case, by making it of type

```
type ufloat = { mutable value : float }
```

This takes exactly as much space as a regular boxed float would, but allows to update the value without any allocation in the mutable case:

```
Immutable record: 147050173.519205 updates/s
Mutable record:   277762346.536304 updates/s
```

Finally, if all other fields are removed, we are left with

```
module P0 = struct
  type person = { balance: float }
end

module P1 = struct
  type person = { mutable balance: float }
end
```

Since the records only hold values of type `float`, they will be unboxed, so the benchmark now measures the relative costs of allocating 16(12) bytes and updating a double value in memory:

```
Immutable record: 224603010.554612 updates/s
Mutable record:   229258325.999858 updates/s
```

Allocation is fast enough to make the difference insignificant compared to the cost of the float operations (`balance +. 1.0`) and the loop overhead. This is the code generated on x86-64 for the mutable case:

```
.L110:
        movlpd  .L112(%rip), %xmm0
        addsd   (%rax), %xmm0
        movlpd  %xmm0, (%rax)
        movq    %rdi, %rsi
        addq    $2, %rdi
        cmpq    %rbx, %rsi
        jne .L110
```

corresponding to

```
for i = 1 to iters do
  p.balance <- p.balance +. 1.
done
```

It could clearly benefit from loop unrolling plus code hoisting and avoid memory traffic by keeping the intermediate result in the `%xmm0` register, but at least doesn't involve any allocation and doesn't incur into the `caml_modify` overhead.

---

28 May 2009 at 10:49

Comments

1. Excellent analysis, Mauricio. What I find surprising is that such simple code "violates" so easily the expected usage pattern of the generational GC: an old object shouldn't point to the young generation.

   Would a float ref substitute for the single mutable float record?

   —**Matías Giovannini**, 29 May 2009 at 03:33 #

2. How are you timing this? It seems kinda sensitive, and I have no idea how you'd instrument this.

   —**tyler**, 29 May 2009 at 05:22 #

3. Matías: the float in `float ref` is not unboxed; iow. you're left with the boxed float instead the ref's own box.

   tyler: the original code just evaluates several million times a function that does

```
for i = 1 to iters do
  p.balance <- p.balance +. 1.
done
```

so the loop, the float allocation and the actual operation add overhead. It's possible to measure the cost of `caml_modify` more directly by performing many updates per iteration and avoiding any computations/allocation. I show this in the followup to this post.

—**mfp**, 29 May 2009 at 16:10 #
