Statistically profiling memory in OCaml

Jacques-Henri Jourdan

November 12th, 2019
Chameau sur le plateau, Gif-sur-Yvette

Memory profilers

Why does my program eat so much memory?

- Memory leaks
- Inefficient data structures
- ...

Solution 1: profiling allocations

- Use a *generic* profiler for runtime
- Focus on allocations

Solution 1: profiling allocations

- Use a *generic* profiler for runtime
- Focus on allocations

Released blocks should not be counted
$\Rightarrow$ Does not faithfully represent the heap.

Solution 2: attach meta-data to blocks

At each allocation: attach meta-data about the allocation point.

- When needed, analyze the meta-data in the heap.

Examples for OCaml:

- *Ocp-Memprof*: identifier of allocation site
- *Spacetime*: pointer to call graph (built on-the-fly)

Solution 2: attach meta-data to blocks

At each allocation: attach meta-data about the allocation point.

- When needed, analyze the meta-data in the heap.

Examples for OCaml:

- *Ocp-Memprof*: identifier of allocation site
- *Spacetime*: pointer to call graph (built on-the-fly)

Runtime/memory overhead
$\Rightarrow$ Limited amount of information

A **statistical** memory profiler

Track only a **small, representative fraction** of allocations.

Much **lower overhead**

- **Tunable sampling rate**
- Relevant information even for low sampling rates

$\Rightarrow$ Attach **much larger meta-data**

- Full stack traces, values of some variables...

Architecture

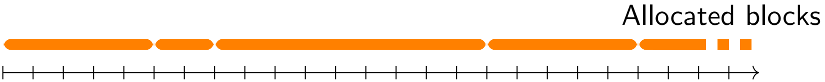
In the runtime system:
only the **sampling** and **tracking** mechanisms

An **arbitrary OCaml closure** is called when:

- a block is *sampled*,
- a sampled block is *promoted*, or
- a sampled block is *deallocated*.

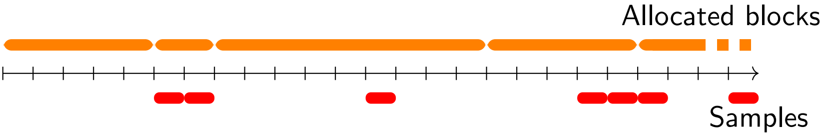
A **client library** chooses, collects and displays relevant information.

Sampling engine



- See allocations as a stream of blocks, seen one after the other
  - Sizes are taken into account

Sampling engine



- See allocations as a stream of blocks, seen one after the other
  - Sizes are taken into account
- Choose sampled **words** at random ("binomial process") at a **tunable rate**
  - Some blocks not sampled, some sampled several times
  - Easy to simulate
  - $\mathbb{E}$(Samples in a block) = Size of the block $\times$ Sampling rate

Interface of the sampler

```
type allocation = private { n_samples : int;
                            size : int;
                            tag : int;
                            unmarshalled : bool;
                            callstack : Printexc.raw_backtrace }
val start :
  sampling_rate:float →
  ?callstack_size:int →
  ?minor_alloc_callback:(allocation → 'minor option) →
  ?major_alloc_callback:(allocation → 'major option) →
  ?promote_callback:('minor → 'major option) →
  ?minor_dealloc_callback:('minor → unit) →
  ?major_dealloc_callback:('major → unit) →
  unit → unit

val stop : unit → unit
```

Sampling algorithm

- Major heap: direct simulation of binomial distribution
  - Large blocks $\Rightarrow$ Amortized cost
- Minor heap:

$$S_2 - S_1 \sim Geom(\lambda)$$

At each event:

1. Simulate position of next sample (geometric law)
2. Change lower limit of the minor allocation arena
   $\Rightarrow$ Control goes back to runtime system when sampling

Non-sampled allocations performed as usual
$\Rightarrow$ No performance regression when $\lambda \ll 1$

Lessons learnt from the prototype

ML workshop 2016

- **Every allocation can be sampled**: C stubs, deserialized objects...

- **Good performances**:

  Sampling rate $\lambda = 10^{-5} \quad \Rightarrow \quad < 1\%$ runtime overhead
  $\lambda = 10^{-4} \quad \Rightarrow \quad < 10\%$

  Yet, **very representative**

- **Requires invasive changes to the runtime and compiler**:
  - Deals with the "Comballoc" optimization
  - Needs good support for asynchronous callbacks (+cleanup)
  - Interacts subtly with the allocators

Challenge #1: The "Comballoc" optimization

Native compiler:

- combines successive allocations
- example: `Some([0; 1; 2], 4, 4)` $\Rightarrow$ **one** allocation of size 16

What happens if a word in a "combined block" is sampled?

- frame tables : description of combined allocations
  - changes needed in `ocamlopt`
- StatMemprof determines which sub-block is sampled, and calls the callback(s) correspondingly

Challenge #2: Async callback safety

It is not safe to run arbitrary OCaml code anywhere

Challenge #2: Async callback safety

It is not safe to run arbitrary OCaml code anywhere

Allocations from C code:

- Example: allocating arrays, ...
- Guarantees: no OCaml callback (in major heap: no GC allowed!)
- StatMemprof **postpones** callbacks for these allocations

Challenge #2: Async callback safety

It is not safe to run arbitrary OCaml code anywhere

Allocations from C code:

- Example: allocating arrays, ...
- Guarantees: no OCaml callback (in major heap: no GC allowed!)
- StatMemprof **postpones** callbacks for these allocations

Handling postponed callbacks:

- Mechanism shared with signals and finalizers
- In C code (incl. bytecode interpreter):
  - `process_pending_actions` called regularly at safe points
- In native code:
  - Minor allocation arena closed ⇒ handled at next minor allocation

Challenge #3: Interaction with native allocator

The problem

Generated native (pseudo-)code for allocations (OCaml $\leq$ 4.10)

```
redo:
young_ptr -= whsize;
if (young_ptr < young_limit) goto gc;
Hd_hp(young_ptr) = header;
[Rest of the function]

gc:
young_ptr += whsize;
call_runtime_system();
goto redo
```

The variable `young_limit` is used:

- as the begining of the minor heap
- for interrupting native code (*e.g.*, signals)
- by StatMemprof, for sampling

Challenge #3: Interaction with native allocator

The problem

Generated native (pseudo-)code for allocations (OCaml $\leq$ 4.10)

```
redo:
young_ptr -= whsize;
if (young_ptr < young_limit) goto gc;
Hd_hp(young_ptr) = header;
[Rest of the function]

gc:
young_ptr += whsize;
call_runtime_system();
goto redo
```

If signal arrives just after sampling

- signal handler will set `young_limit := young_alloc_end`
- signal callback will perform its own allocations before ours
- StatMemprof data structures will point to garbage

Challenge #3: Interaction with native allocator

The solution

Generated native (pseudo-)code for allocations (OCaml trunk)

```
young_ptr -= whsize;
if (young_ptr < young_limit) goto gc;
gc_done:
Hd_hp(young_ptr) = header;
[Rest of the function]

gc:
call_runtime_system();
goto gc_done
```

- Same hot path, smaller code overall ⇒ performances OK
- Very close to the bytecode/C code allocator ⇒ share more code
- Runtime system now needs to know `whsize`
  - Read it from frame tables (StatMemprof needs it anyway)

Future work

Needed for the release (in OCaml 4.11):

- Merge in OCaml trunk sampling for native code
- Make StatMemprof reentrant
  - Thread preemption can occur during a callback

Optimizations (in OCaml, some day):

- Faster capture of callstack
- Faster generation of geometric random variables
  - Better PRNG, faster log approximation, vectorized computations

Client libraries:

- Combine with Spacetime/Ocp-Memprof?
- Dedicated library?

Conclusion

- Together with Spacetime and Ocp-Memprof, we will soon have efficient tools for understanding memory consumption in OCaml.
- StatMemprof in 4.11:
  - Most of the code is merged.
  - Many improvements compared to initial prototype
  - Many thanks to Stephen Dolan, Jane Street, the core OCaml team !
  - Still a few PRs are needed