Statistically profiling memory in OCaml

Jacques-Henri Jourdan

MPI-SWS, Inria Paris-Rocquencourt

1 Introduction

Even on today's powerful computers, memory is a limited resource. Therefore, it is important to understand the memory usage of programs. A common technique used is to use *memory profilers* [3, 1]. Such a tool monitors the heap during the execution of the program and is able to give information about the objects it contains. A memory profiler does not simply tracks which part of the code allocates a lot of memory, but rather it gives information about the memory that stays allocated in the heap.

Typical memory profilers attach an additional information to every allocated block. They are limited in the amount of information they can collect, because of the performance overhead this may cause. In this work, we investigate the design of a statistical memory profiler: we track only a representative part of the heap and deduce global statistical properties.

The idea of our approach for memory profiling is to track randomly chosen memory blocks in the heap without changing their reachability properties. Because we track only a small fraction of the heap, we can reasonably attach much more information to each tracked block, such as call stacks or information about the program state at allocation. In fact, we give the opportunity to call an arbitrary function when sampling a memory block.

Two mechanisms are needed: first, a data structure able to associate meta-data to each tracked object transparently without modifying its lifetime and second, a mechanism to randomly sample allocations. For better flexibility, in our proposal, these mechanisms are independent and manageable by the program itself.

2 Managing the set of tracked blocks with ephemerons

The data structure used for tracking sampled blocks has to meet several constraints. First, it needs to not interfere with the lifetime of the block: indeed, in order to avoid any bias in the profiling results, we need to make sure that the sampled blocks behave like the not sampled ones, up to a statistical error. Second, in order to avoid any memory leak in the memory profiler itself, we need to make sure the meta-data associated to tracked memory blocks is freed as soon as the tracked block is. Hence, if the sampling rate is sufficiently low, the memory overhead of the profiler is amortized by the memory used by the program.

Natural candidates meeting these two constraints are *ephemerons* [4], recently implemented in OCaml as part of the 4.03 release [2]. They are a special kind of blocks handled by the garbage collector. Slightly simplifying for our purpose, ephemerons are pairs of a *data* field and a *key* field. The key is a weak reference: it does not prevent the block it points to from being collected. On the other hand, the data is a strong reference as long as the key is alive, but it is cleared automatically by the GC when the key is reclaimed.

In our memory profiler, the set of tracked memory locations are kept in an extendable array of ephemerons, where keys point to sampled blocks and data contain the associated meta-data. We periodically traverse the array for removing the emptied ephemerons and shrink it if its filling ratio is bellow a certain threshold.

An interesting feature of this mechanism is that, apart from ephemerons, which are already implemented in the OCaml runtime, it needs no runtime support and can entirely by implemented and customized as part of a user library or a part of the program itself.

3 Sampling allocations

The other essential component needed for our memory profiler is a mechanism for sampling allocations. The sampler works by calling a user specified callback each time an allocation is sampled. The callback can be provided by the program itself or by a library it is linked to. Typically, it will consists in populating a data structure such as the one described in §2. It is implemented in a patched version of the OCaml runtime [5].

The sampling distribution itself follows a Poisson process: intuitively, the sampler behaves as if a random, memoryless stream of tokens were produced while memory is allocated. An allocated memory blocks may contain zero of these tokens, in which case it is not sampled, or one or more tokens, in which case it is sampled. The expected number of tokens an allocated block has is proportional to the size of the block (including its header). This random distribution has several advantages: first, its statistical properties make then very easy to estimate actual statistical characteristics of the heap. Indeed, we certainly want large allocated blocks to weight more than small ones. Second, Poisson process are relatively easy to simulate with reasonable performance.

The API provided by our sampler supports changing the sampling rate (defined as the average number of tokens per allocated word), changing the size of the call stack captured at every sampled block and setting the callback. A callback is given by the sampling engine a parameter containing some meta data about the allocation, represented by the `sample_info` type:

```
type sample_info = {
    n_samples: int;             (** The number of samples in this block. *)
    kind: alloc_kind;           (** The kind of the allocation (i.e., major,
                                    minor or serialized). *)
    tag: int;                   (** The tag of the allocated block. *)
    size: int;                  (** The size of the allocated block. *)
    callstack:
      Printexc.raw_backtrace;   (** The callstack for the allocation. *)
}
```

It is worth noting that this structure does not contain the block being allocated itself. Indeed, because it has not already been allocated, it contains potentially invalid data and cannot already be manipulated by the program. Instead, in order to implement a tracker such as the one described in §2, the callback can optionally return an ephemeron the key of which will be assigned to the fresh block by the sampling engine. Hence, the type for callbacks is:

```
type 'a callback = sample_info -> (Obj.t, 'a) Ephemeron.K1.t option
```

Internally, the sampling engine uses two different techniques to sample allocations, depending on whether they take place in the minor or in the major heap. For allocations in the major heap, it simply draws a random variable within a Poisson distribution and launch the callback whenever it is not zero. For allocations in the minor heap, it uses a mechanism similar to the one already used for handling signals: it modifies the limit of the current allocation arena in order to interrupt the program when a sampling takes place. Both systems have low performance overhead: in practice, the total overhead is mainly due to the execution of the callback.

4 Conclusions

We used this profiling system to debug several of our programs. When using 10^{-4} as a sampling rate, the performance overhead is usually under 10%, but we still get much relevant information about the state of the heap of the program. Using lower sampling rates, such as 10^{-5}, we can make the performance overhead barely noticeable while still being able to understand major memory problems.

Perhaps the most important piece of work to improve this tools would be to design tools efficiently exploit the information gathered. Additionally, we wonder whether our sampling mechanism for allocations could be used for other purposes.

References

[1] The Spacetime memory profiler. https://github.com/ocaml/ocaml/pull/585/.

[2] F. Bobot. Ephemerons meet OCaml GC. OCaml Workshop, 2014.

[3] Ç. Bozman. *Profilage mémoire d'applications OCaml*. PhD thesis, ENSTA ParisTech, 2014.

[4] B. Hayes. Ephemerons: a new finalization mechanism. In *OOPSLA*, pages 176–183. ACM, 1997.

[5] J.-H. Jourdan. OCaml 4.03 patched for allocation sampling. https://github.com/jhjourdan/ocaml/tree/memprof/.