Profiling the Memory Usage of OCaml Applications without Changing its Behavior

Çağdaş Bozman
OCamlPro & INRIA & ENSTA-ParisTech
cagdas.bozman@ocamlpro.com

Michel Mauny
ENSTA-ParisTech
michel.mauny@ensta.fr

Fabrice Le Fessant
INRIA & OCamlPro
fabrice.le_fessant@inria.fr

Thomas Gazagnaire
OCamlPro
thomas.gazagnaire@ocamlpro.com

Abstract

In this paper, we present the current state of our work on profiling the memory usage of OCaml programs. Our technique allows to observe track types, allocation points and reachability paths of blocks, with no runtime cost, except for saving the observations.

1 Introduction

OCaml is a strongly typed functional programming language with automatic memory management, using a generational and incremental garbage collector. The main benefit from automatic memory management is that it frees the programmer from manually dealing with memory allocation/deallocation and thus ensuring that a large class of bugs never happen, like dangling pointers, double frees, etc. However, automatic memory management has also some drawbacks: some values might never be reclaimed by the garbage collector (for instance when they are stored in a global hash-table) and the garbage collector itself consumes resources to decide which memory cells to free. As the memory pressure increases, automatic memory management can become a performance problem for the application as well.

Thus, there is a need for tools to help developers profile the memory usage of their OCaml programs, but almost nothing is currently available for OCaml. We implemented a tool, called `ocamlmemprof` [1], that can save information on the memory usage of an OCaml application, providing a way to gather type/allocation statistics and reachability information for every block in the heap of the application, with no runtime penalty except for saving the data. We present our technique and implementation in the remaining of this paper.

2 General Idea

Thanks to its strong type system, OCaml does not need much runtime type information in values to perform standard computations. Consequently, the memory representation of OCaml values is much more compact than values of a weakly typed language, resulting in better memory performance. However, when profiling memory performance of OCaml programs, the – almost complete – absence of runtime type information becomes a problem to recover information from the values currently using the heap space.

In previous versions of `ocamlmemprof` [1], we stored additional runtime type information in blocks, to be able to recover enough

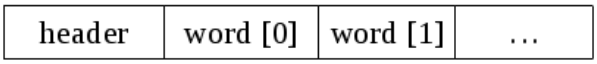

Figure 1: A block used for representing an OCaml value

Figure 2: Format of the 64-bit block header

interesting information. However, doing so changes the behavior of the application: as blocks are one-value longer (for a list, it means a 25% overhead), there is an increased pressure on the garbage collector, resulting in two drawbacks: as the program allocates more, and the overhead ratio depends on the size of the block, the profiled program does not perform exactly like the production program, leading to possibly inaccurate conclusions; also, since profiling impacts the general performance of the application, it might be impossible to profile an application in production, and thus to debug a problem occuring after a few days or weeks of execution.

In the new version of `ocamlmemprof`, we are experimenting a new technique to profile the memory behavior of an application, without impacting its performance nor adding memory overhead. Basically, the technique uses a useless part of the header of values on 64 bits platforms, to store a small identifier. This identifier can be used to recover the allocation point and the type of the value, although its small size might lead to collisions, that we think can be distinguished using simple heuristics.

Figure 1 shows a block in the OCaml heap, corresponding to an OCaml value. The block is prefixed by a 64-bit header (on 64 bits systems) whose format is displayed on Figure 2: The first 54-bits are used to encode the size of the block (in words, i.e. 8 bytes values), then 2 bits are used for garbage collection (mark-and-sweep colors[2]), and finally 8 bits are used to encode the minimal type information

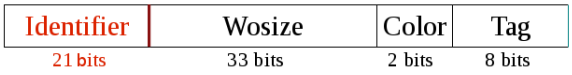

Figure 3: The *modified* header format

required for standard operations (discrimination between variants with arguments, polymorphic comparisons, etc.).

We modified this format to use 21 bits of the size to encode an identifier, corresponding to the location in the code where the block was allocated, and thus, the type of the block with high probability. The new format of the header is presented on Figure 3. Thanks to this modification, we can discriminate 2 million allocation sites, while still being able to allocate blocks of size 64 GB (33 bits times 8 bytes), fitting all usages in known OCaml applications.

3 Locations Identifiers

3.1 Generating Identifiers

First we need to patch the OCaml compiler to propagate the source location to the bytecode and/or the code generation (native version). For that purpose, we generate a random identifier directly in the compiler for each allocation site, which will allow us, later, to associate a location in the source code to each given memory block in the heap.

Identifiers are generated by a simple calculation: a hash of the location and the digest of the module name. The drawback with this computation is that there is a risk of collision (two locations with the same identifier), given the little identifier space.

When compiling an application, we generate a `.prof` file, containing all (identifier,location) pairs of the program: for each unit, we store them in `.cmo/.cmi` files, and combine them at link time. Later, this file will allow us to find quickly the location corresponding to identifiers.

Once we have those locations, it is quite

easy to extract the type information that we need from `.cmt` files.[1]

3.2 Storing Identifiers

Now that we can generate identifiers and we have the association between those identifiers and the locations they denote, we need to store this information in the heap.

For this, we modified both the compiler backends, to store the identifiers when allocating blocks (headers are statically computed at compile time, so adding identifiers has no performance impact). We also modified all C primitives that would allocate in the heap, and predefined static location identifiers for all of them, to be able to discriminate between values allocated in the runtime.

4 Profiling with Snapshots

There are several ways to use the identifiers stored in the blocks. In this section, we present a first technique, based on saving snapshots of the program memory.

A snapshot is a file containing the graph of blocks in the heap, after a full garbage collection. Each snapshot is called `heap.dump.PID.COUNT`, where PID is the identifier of the running process, and COUNT the number of the snapshot.

The file is generated by scanning all the chunks composing the heap (a chunk is a huge block of memory, where the garbage collector can allocate OCaml values). In each chunk, a block is saved into the snapshot if it is not in the free-list (blocks of the freelist have a header with blue color). When saving a block, we store the header of the block (tag, size, identifier) and the content, if it contains OCaml values (i.e. not a string, abstract block, floats or custom). Finally, we also save in the snapshot the content of globals values, i.e. toplevel modules, so that it is possible to recover from the snapshot, the path by which any block is reachable from a global root.

All this information can be used to provide statistics on the blocks in the heap (types and allocators, using identifiers, but also sizes or tags). It can also be used to navigate inside the graph of values, and to recover exact types when collisions happen between our identifiers, by comparing the shape of the content with the possible types.

For the user, there are mostly two ways to trigger the generation of snapshots:

- The first method is to use `OCAMLRUNPARAM=m`, that will force a program to generate a snapshot after every garbage collection. This behavior can be especially useful to follow the evolution of the heap content during the lifetime of a program. However, given the size of snapshots, such a technique can only be used by short-live programs, that run too fast to be monitored by an online tool.

- The second method is to ask the program to generate a snapshot at a specific time. Such a method can be used with a server running for days (typically *ML-Donkey* [3]), where an increase of memory usage has been noticed, to understand the reasons for such an increase. The current implementation can trigger the generation of a snapshot when receiving the `HUP` signal, or when the function `Gc.dump_heap ()` is called from the user interface.

5 Continous Profiling

We saw in the previous section how to profile an application using memory snapshots, but sometimes we want to profile some applications during their execution.

We want to track blocks in the minor generation heap for example. For that purpose,

[1] Files containing type-annotated parse trees generated by `-bin-annot`.

we added some arrays, indexed by the identifiers generated by the compiler, which will allow us to know at any time during the application's execution the number of element of a value alive in the minor heap.

In the same way, we can extend these arrays to the major generation heap and then know the number of value directly allocated in the major heap. Having an array of collected object is then trivial.

Finally, we can easily have different type of information like the promotion from the minor generation heap to the major generation heap. We can also have information about a sort of life span of memory blocks: this information can be collected by counting how many GC a bloc survived.

A profiler tool can then just be connected to these arrays to show continuous way the evolution of the memory.

6 Conclusion and Perspective

Using our tool, it is now possible to have a better idea of the memory behavior of OCaml programs. Once the profiling is done, we need to understand results and analyze the usage of memory for each type to observe suspicious amount of memory usage.

To summarize, the new `ocamlmemprof` consists in a set of modifications to the compiler and runtime from the OCaml distribution to dump heap images into a journal.

We hope that this tool will help developers spot memory problems in their programs. Advanced users can use this information to improve their programs by reducing allocations in their programs' critical paths. For less advanced users, we plan to develop other tools, more intuitive, heuristically providing advice on how to reduce the memory cost of a particular function or type.

Since we have some data associated to heap objects, we plan to track other interesting information like which values go from minor generation heap to major generation heap, or which values stay alive after a number of major collections, and (why not) tracking how long a value survives, in collections. In the next versions, we will also try to manage collision in identifiers looking for example the shape of blocks to try to distinguish one type from another.

To conclude, we have now a lot of information that we have to exploit and our work is to highlight the useful part to show memory issues.

References

[1] C. Bozman, M. Mauny, F. Le Fessant, and T. Gazagnaire. Study of ocaml programs' memory behavior. OCaml Users and Developers, 2012.

[2] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. 20th symp. Principles of Programming Languages*, pages 113–123. ACM press, 1993.

[3] F. Le Fessant, S. Patarin, et al. Mldonkey, a multi-network peer-to-peer filesharing program. 2003.