

Rapport

Étude de Caml-Gmp au niveau des problèmes de la gestion mémoire

Philippe Wang¹

27 novembre 2006

Revision : 1.6

¹philippe.wang@etu.upmc.fr
Université Pierre et Marie Curie, Paris, France
Master de Recherche en Informatique,
Spécialité Science et Technologie du Logiciel - Algorithmique et Programmation

Résumé

Ce rapport présente l'étude de l'interface Caml-GMP au niveau de la gestion mémoire.

Les performances de la gestion mémoire offerte par le ramasse-miettes d'Objective Caml ne répondent pas aux attentes des utilisateurs de l'interface Caml-GMP qui permet d'utiliser la bibliothèque de calculs sur les grands nombres GMP ^a.

En effet, les objets non utilisés sont libérés trop lentement dans certains cas de figure, comme lorsqu'on manipule peu d'objets prenant beaucoup de place mémoire hors des tas d'Objective Caml. Le GC d'Objective Caml ne connaissant que les pointeurs sur les objets alloués hors de ses tas, son algorithme est « leurré » et le GC ne se déclenche alors pas assez souvent.

Cette étude met en évidence le genre de cas où ce problème se manifeste, et fait le point sur ce qu'on arrive à faire actuellement, ce qu'il est possible ou impossible de faire pour améliorer la situation.

^a[GNU Multiple Precision Arithmetic](#)

Remarques

Pour une utilisation optimale, ce document doit rester en PDF, ou être imprimé en couleur dans la mesure du possible s'il faut une version papier. (il faut éviter de l'utiliser avec le format PS qui fait perdre des informations)

Table des matières

1	Introduction	4
1.1	GNU Multiple Precision Arithmetic Library	4
1.2	Objective Caml	4
1.3	Caml-GMP	5
1.4	Le GC d'Objective Caml	5
2	Environnement de tests	6
2.1	Outils de tests	6
2.1.1	Systèmes	6
2.2	Outils de mesure	7
3	Tests : <code>caml_(alloc free)_dependent_memory</code>	7
3.1	Qu'est-ce que c'est ?	7
3.2	Quelle est son influence ?	7
3.2.1	Pour des « petits » objets nombreux	7
3.2.2	Pour des « grands » objets peu nombreux	8
3.2.3	Des gros objets bien accompagnés	10
3.2.4	Comparaison entre « beaucoup d'objets gros et petits » et « peu d'objets gros »	10
4	Conclusion	14
5	Annexe	15
6	Bibliographie	17
7	Remerciements	17

Table des figures

1	Environnement 1	6
2	Environnement 2	7
3	<code>caml_alloc_dependent_memory</code> n'a pas d'effet sur le com- portement du GC pour des petits objets	9
4	L'occupation mémoire avec beaucoup d'objets gros et petits . .	11
5	Mise en évidence de l'influence de <code>caml_(alloc free)_dependent_- memory</code>	12
6	Comparaison de l'occupation mémoire entre beaucoup d'objets gros et petits, et peu d'objets gros	13

1 Introduction

1.1 GNU Multiple Precision Arithmetic Library

La bibliothèque de calculs multiprécision GMP permet d'effectuer des calculs avec des nombres à précision arbitraire.

What is GMP? [1]

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface.

The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.

GMP is carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is achieved by using fullwords as the basic arithmetic type, by using fast algorithms, with highly optimized assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed.

GMP is faster than any other bignum library. The advantage for GMP increases with the operand sizes for many operations, since GMP uses asymptotically faster algorithms.

The first GMP release was made in 1991. It is continually developed and maintained, with a new release about once a year.

GMP is distributed under the [GNU LGPL](#). This license makes the library free to use, share, and improve, and allows you to pass on the result. The license gives freedoms, but also sets firm restrictions on the use with non-free programs.

GMP is part of the GNU project. For more information about the GNU project, please see the official [GNU web site](#).

GMP is brought to you by a team [listed in the manual](#).

GMP est écrite en langage C pour le langage C, mais ayant bien sûr de nombreuses interfaces pour être utilisé par d'autres langages de programmation.

1.2 Objective Caml

Objective Caml[2] est un langage de programmation fonctionnel, typé statiquement, polymorphe paramétrique, possédant une inférence de types, muni d'un mécanisme d'exceptions, possédant des traits impératifs, capable d'exécuter des processus légers, capable de communiquer sur le réseau Internet, dis-

posant de nombreuses bibliothèques, disposant d'un environnement de programmation, capable d'interagir avec le langage C, disposant de trois modes d'exécution [3]...

Et Objective Caml dispose d'un « ramasse-miettes », également appelé « glaneur de cellules » ou « garbage collector » qui permet de ne pas avoir à gérer la mémoire soi-même. On l'appellera simplement « GC » dans la suite de ce rapport.

1.3 Caml-GMP

Pour combiner les fonctionnalités proposées par GMP et par Objective Caml, on implante une interface entre GMP et Objective Caml qui permet d'utiliser GMP en programmant en Objective Caml.

On appellera cette interface « `mlgmp` » dans la suite de ce rapport.

1.4 Le GC d'Objective Caml

L'utilisation d'un GC offre la garantie de non libération d'un objet encore atteignable durant l'exécution, en plus de libérer le programmeur de la gestion de la mémoire...

Objective CAML's garbage collector combines the various techniques described above. It works on two generations, the old and the new. It mainly uses a Stop&Copy on the new generation (a minor garbage collection) and an incremental Mark&Sweep on the old generation (major garbage collection).

A young object that survives a minor garbage collection is relocated to the old generation. The Stop&Copy uses the old generation as the to-space. When it is finished, the entire from-space is completely freed.

[...]

[3]

Le problème que nous rencontrons et mettons en évidence est celui du rôle du GC dans les programmes utilisant des interfaces de communication entre deux langages de programmation.

Le GC se déclenche en fonction du nombre d'objets alloués et de leur taille connue par Objective Caml. En se restreignant à la librairie standard fournie avec Objective Caml, les objets sont quasi tous de petites taille. Les chaînes de caractères sont de tailles arbitraires, de même pour les tableaux, et peuvent donc être gros. Les autres objets potentiellement « gros » sont souvent composés de beaucoup de petits objets, ce qui incrémente assez rapidement le compteur du GC d'Objective Caml pour qu'il se déclenche assez fréquemment (mais pas trop souvent non plus).

L'interface `mlgmp` laisse GMP allouer ses objets, et indique seulement sa présence à Objective Caml qui n'a alors connaissance que d'un pointeur. Ainsi lorsque l'on manipule relativement peu d'objets mais qui sont gros, Objective Caml n'a pas connaissance de la taille de ces objets et ne déclenche pas le GC assez fréquemment, ce qui peut faire exploser la quantité de mémoire utilisée.

Objective Caml propose alors un mécanisme encore expérimental qui consiste à indiquer au niveau de l'interface les tailles des objets alloués ailleurs que dans un tas d'Objective Caml. Deux fonctions aux noms plus ou moins éloquents sont fournies : `caml_alloc_dependent_memory` (qui permet d'indiquer la taille d'un objet qui vient d'être alloué) et `caml_free_dependent_memory` (qui permet d'indiquer qu'un objet d'une certaine taille a été libéré).

Remarquons également que les objets alloués hors du tas `caml` ne sont pas libérés lors des GC mineurs : il faut que le GC majeur se déclenche pour appeler les fonctions de finalisation des objets extérieurs aux tas d'Objective Caml.

Nous verrons que ce mécanisme n'est pas encore au point...

Ainsi, dans un premier temps, nous présentons les environnements de tests, puis dans un second temps, les résultats des tests effectués. Enfin, nous concluons en donnant des perspectives potentielles.

2 Environnement de tests

2.1 Outils de tests

2.1.1 Systèmes

Les tests ont utilisé deux systèmes (figures 1 et 2).

- AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ @ 2.0GHz, 512KB cache/core
- 2 × 512 MB DDR2 800 (dual channel)
- Ubuntu Linux 6.10 (64-bit SMP)
- Objective Caml 3.09.3 (2006-09)
- GMP 4.2.1 (2006-05-04)
- MPFR 2.2.0 (2005-09-20) (pour satisfaire `mlgmp`...)
- `mlgmp` original et `mlgmp` avec `mlgmp_z.c` modifié

FIG. 1 – Environnement 1

Les premiers tests ont été effectués avec la machine 32-bit, puis sur les deux architectures pour observer les différences. Enfin les tests ont été faits uniquement avec la machine 64-bit. La seule différence est que l'une est grossièrement entre 2 et 4 fois plus rapide que l'autre. Il est à noter que le *dual core* n'est pas mis à contribution pour l'exécution des calculs (seul le système en profite, pas les programmes de tests, et la gestion de la mémoire par le GC donne les mêmes résultats).

- AMD Athlon(tm) XP 2600+ @ 1.9GHz, 512KB cache
- 2 × 512 MB DDR 400 (dual channel)
- Ubuntu Linux 6.06 (32-bit)
- Objective Caml 3.09.3 (2006-09)
- GMP 4.2.1 (2006-05-04)
- MPFR 2.2.0 (2005-09-20) (pour satisfaire mlgmp...)
- mlgmp original et mlgmp avec `mlgmp_z.c` modifié

FIG. 2 – Environnement 2

2.2 Outils de mesure

Un script (listing 8) écrit en `bash` utilisant la commande `ps` a été utilisé pour mesurer les consommations en mémoire vive. Les consommations en mémoire virtuelle ont été observées avec l’outil graphique `gnome-monitor`, mais cela n’est pas important même s’il faut noter que chaque fois qu’on observe un sommet d’utilisation de la mémoire avoisinant les 90%, c’est qu’une partie de a été transférée dans la mémoire virtuelle (puisque l’outil n’observe que le taux d’utilisation de la mémoire vive matérielle).

3 Tests : `caml_(alloc|free)_dependent_memory`

3.1 Qu’est-ce que c’est ?

Les fonctions `caml_alloc_dependent_memory` et `caml_free_dependent_memory` ont pour rôle d’indiquer au GC les tailles des objets alloués hors des tas d’Objective Caml afin de pouvoir ajuster sa fréquence de déclenchement. Ces fonctions sont au stade expérimental.

Pour expérimenter les impacts de ce couple de fonctions, nous avons choisi le fichier `mlgmp_z.c` qui contient la partie en langage C de l’interface `mlgmp` pour les entiers multiprécision. Il s’agit alors de placer à chaque endroit où un objet est alloué hors des tas d’Objective Caml une indication sur la taille de l’objet alloué avec `caml_alloc_dependent_memory`, et de placer `caml_free_dependent_memory` au niveau de la fonction de finalisation en lui donnant également l’information sur la taille de l’objet alors désalloué.

Les macros du listing 4 servent à simplifier la tâche.

3.2 Quelle est son influence ?

3.2.1 Pour des « petits » objets nombreux

```

1 open Gmp
2
3 let rec fact n =
4 (* factorielle naïve : les objets générés pendant son calcul
5   ne peuvent pas être récupérés pendant le calcul *)
6   if Z.equal n Z.zero

```

```

7   then Z.one
8   else Z.mul n (fact (Z.pred n))
9
10  let nn =
11    (* nombre arbitraire *)
12    Z.mul (fact (Z.from_int 9999)) (fact (Z.from_int 9999))
13
14  let ffact n =
15    (* calcul créant des objets de tailles variées *)
16    let rec ffact n r =
17      if Z.equal n Z.zero
18      then r
19      else
20        ffact
21          (Z.pred n)
22          (Z.modulo (* pour limiter la taille du nombre *)
23                   (Z.mul r n)
24                   (Z.mul (Z.from_int (Random.int max_int)) nn))
25    in ffact n Z.one
26  let nn = ffact (Z.from_int 999)
27  let _ = print_endline (Z.to_string (ffact (Z.from_int 99991)))

```

Listing 1 – Test du GC sur des petits objets nombreux

La figure 3 montre que le GC est déclenché exactement de la même manière avec et sans `caml_(alloc|free)_dependent_memory`.

3.2.2 Pour des « grands » objets peu nombreux

D’une manière générale, lorsqu’un programme génère peu d’objets, le GC est peu déclenché. C’est exactement ce qui se passe lorsqu’on manipule peu d’objets (de grande taille) avec `mlgmp`. Il en résulte que les objets de grande taille inutiles ne sont pas récupérés ou bien sont récupérés (trop) tard.

Cela est dû à la non-connaissance par le GC Objective Caml des tailles des objets alloués. Et les fonctions offertes pour informer le GC n’y font rien.

Pour ce test, le programme (listing 2) montre à l’exécution qu’il n’y a aucune influence visible, et que le rôle du GC n’est pas bien tenu.

Pour obtenir les mêmes performances en C (listing 6) et en Objective Caml pour ce programme, il suffit de déclencher explicitement le GC majeur² après chaque itération de la première boucle `for` du programme en Objective Caml : comme il n’y a que très peu d’objets, le parcours des tas d’Objective Caml a un coût négligeable. Cette solution n’est pas acceptable pour un programme avec beaucoup d’objets, soit la plupart des programmes !

```

1  open Gmp
2
3  let calc () =
4    let a = Z.from_int 999 in
5    let x = ref a in
6    for i = 1 to 26 do
7      x := (Z.mul !x !x);
8    done
9
10 let _ =
11   for i = 0 to 10 do
12     calc();

```

²avec `Gc.full_major()` ;

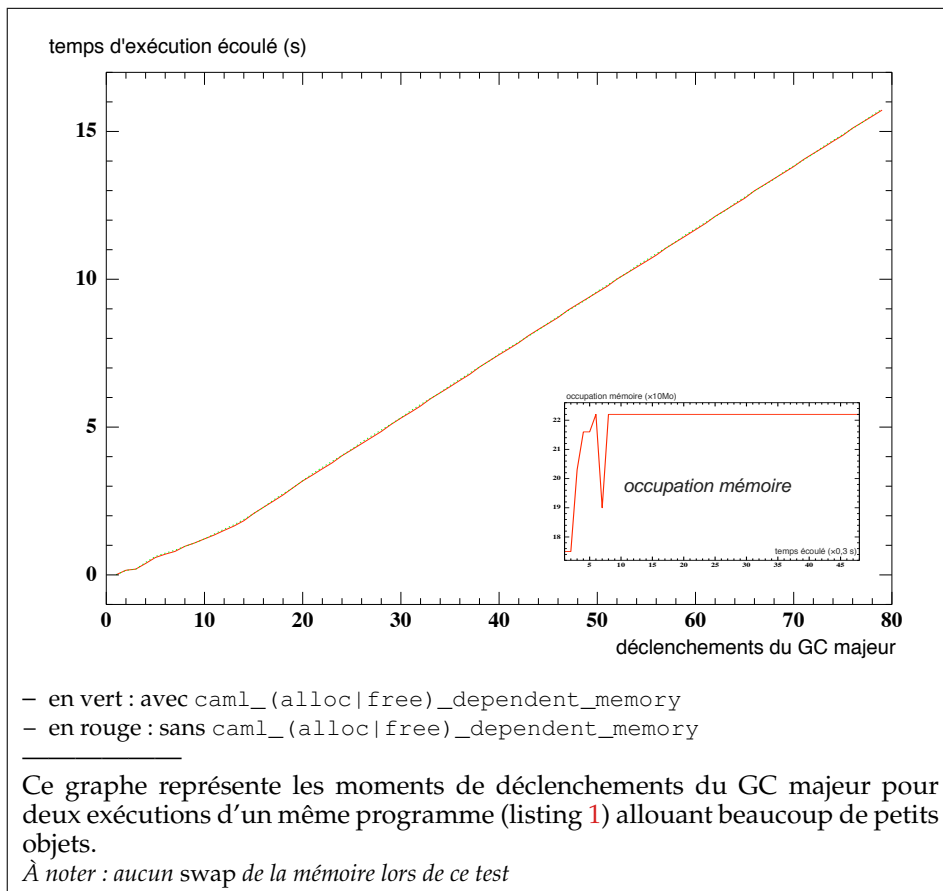


FIG. 3 – `caml_alloc_dependent_memory` n'a pas d'effet sur le comportement du GC pour des petits objets

13 | `done`

Listing 2 – Test du GC avec des gros objets solitaires

3.2.3 Des gros objets bien accompagnés

On se pose alors la question de ce qui se passe lorsqu'on entrelace des calculs manipulant peu de gros objets avec des calculs manipulant beaucoup de plus petits objets (listing 3, figure 4).

```
1 open Gmp
2
3 let rec fact n =
4   if Z.equal n Z.zero
5   then Z.one
6   else Z.mul (fact (Z.pred n)) n
7
8 let calc () =
9   let a = Z.from_int 999 in
10  let x = ref a in
11  for i = 1 to 26 do
12    x := (Z.mul !x !x);
13    ignore(fact (Z.from_int 2600))
14  done
15
16 let _ =
17   for i = 0 to 10 do
18     calc();
19     ignore(fact (Z.from_int 26000))
20  done
```

Listing 3 – Test du GC des gros objets accompagnés

On prend le programme précédent (listing 2), et on ajoute des calculs rapides générant beaucoup d'objets relativement petits : le calcul (naïf) de factorielle 2600 (resp. 26000) rend un nombre à environ 7800 (resp. 103500) chiffres décimaux, et génère obligatoirement plus de 2600 (resp. 26000) objets³. Cela force le GC majeur à se déclencher plus souvent (c'est-à-dire de temps en temps au lieu de jamais...)

3.2.4 Comparaison entre « beaucoup d'objets gros et petits » et « peu d'objets gros »

La figure 6 met en évidence la perte engendrée par la surconsommation de la mémoire vive physique, puisqu'en augmentant le nombre de calculs effectués, on arrive d'une part à une meilleure utilisation de la mémoire et d'autre part à une exécution plus rapide (puisque'on évite la recopie de la mémoire physique vers la mémoire virtuelle).

³On peut choisir un programme plus ou moins rapide à écrire et qui s'exécute beaucoup plus vite... mais celui-ci fait bien ce qu'on veut

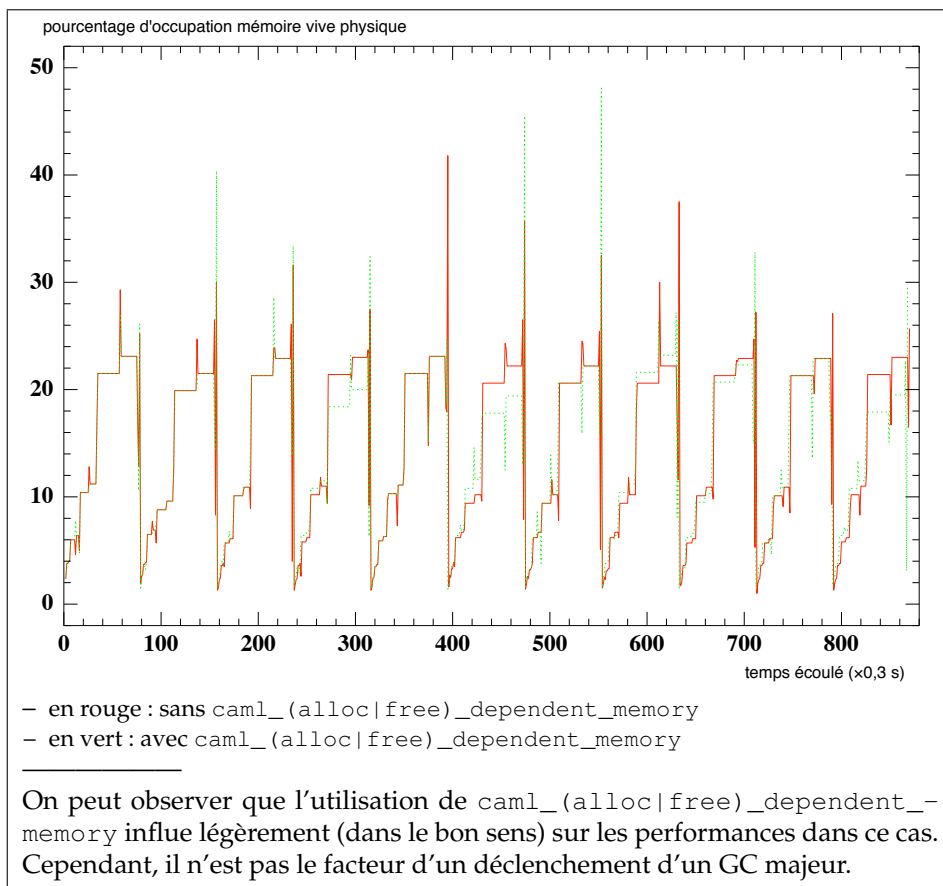


FIG. 4 – L'occupation mémoire avec beaucoup d'objets gros et petits

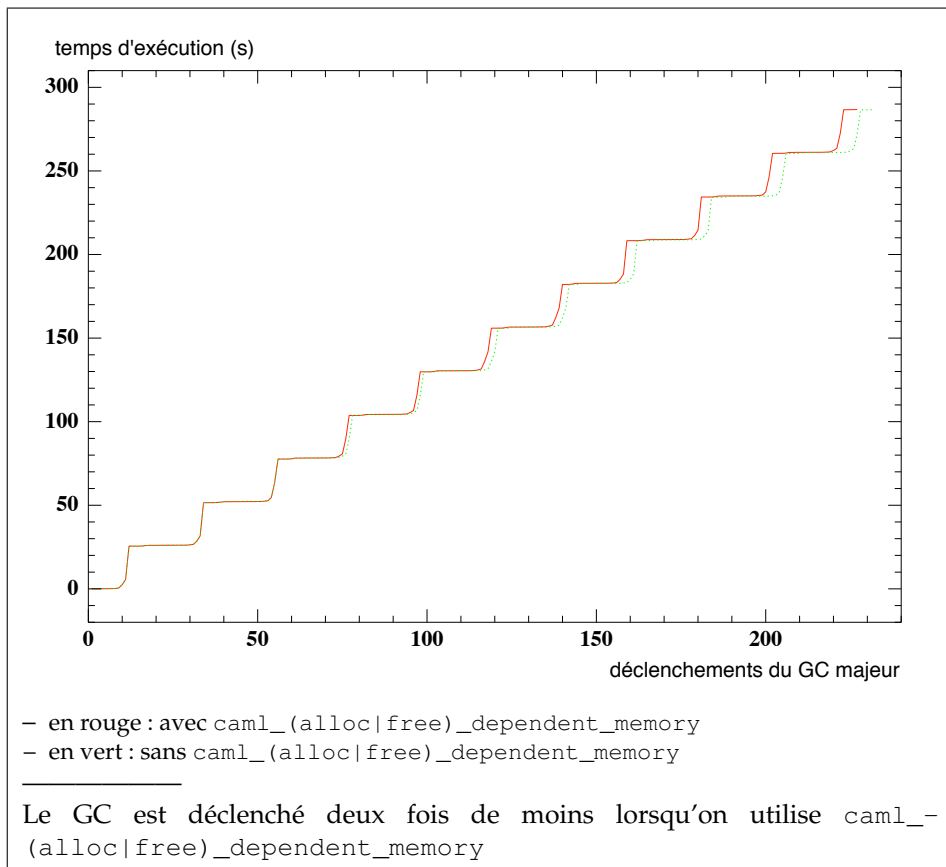


FIG. 5 – Mise en évidence de l'influence de `caml_(alloc|free)_dependent_memory`

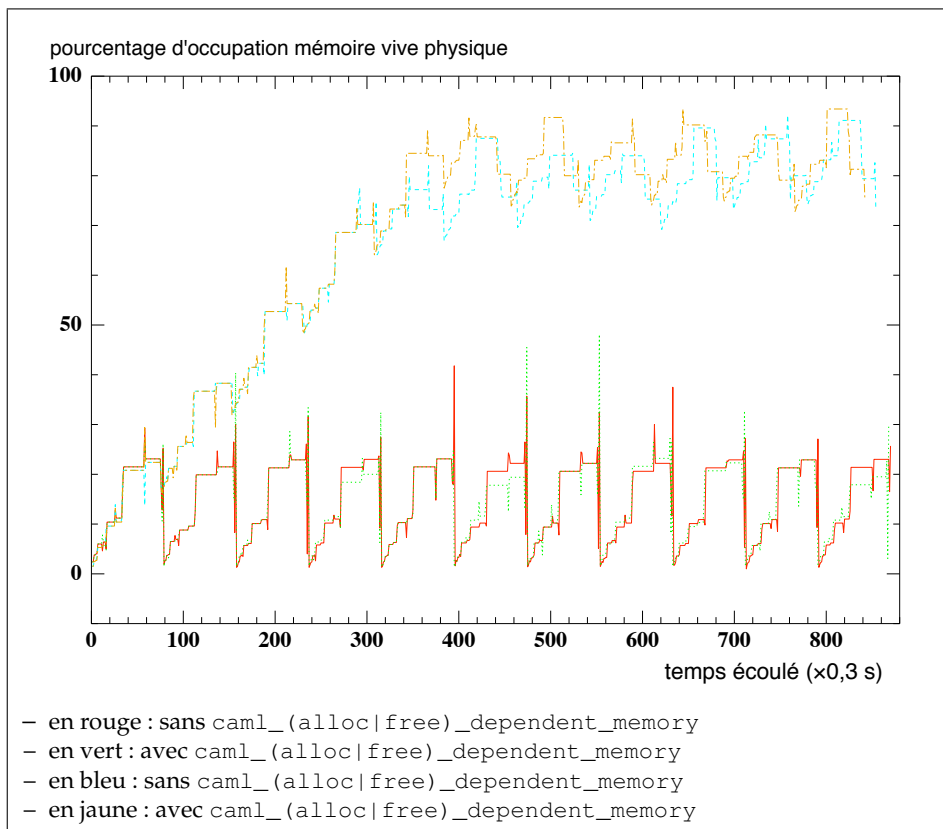


FIG. 6 – Comparaison de l'occupation mémoire entre beaucoup d'objets gros et petits, et peu d'objets gros

4 Conclusion

La gestion de la mémoire quand on veut utiliser de gros objets alloués hors des tas d'Objective Caml, comme le fait `mlgmp`, est à améliorer. Elle possède des heuristiques bien différentes de celles rencontrées lors de l'utilisation de bibliothèques plus classiques qui ne travaillent pas sur des si gros objets « cachés ». Il faut bien sûr remarquer que les bibliothèques, qui travaillent sur des objets potentiellement énormes (plusieurs centaines de Mo, voire plus) cachés derrière des pointeurs (de l'ordre de la dizaine d'octets...), telles que GMP sont plutôt « rares », ce qui ne facilite pas la tâche.

La solution `caml_(alloc|free)_dependent_memory` n'est pas au point. Elle apporte un semblant de plus actuellement. La difficulté est qu'en apportant des modifications sur l'algorithme du GC en vue de l'améliorer sur un point ne doit pas pénaliser ceux qui n'en profitent pas, alors on ne peut pas modifier le GC facilement parce qu'il est actuellement particulièrement efficace dans les autres applications.

On ne peut pas non plus allouer dans un tas Objective Caml dans la mesure où ces objets risquent d'être déplacés et qu'il faudrait adapter GMP, ce qui n'est en fait pas du tout envisageable.

Un travail pourrait être la mise en place d'une sorte de « *brut force* » cherchant une formule optimale pour la prise en compte des valeurs passées à `caml_(alloc|free)_dependent_memory`. Ce genre de fonctionnalités étant mises au point selon des heuristiques, il n'est facile d'arriver à une bonne solution que lorsque l'on a de la chance... car sinon il faut chercher plus longtemps...

D'autre part, si on accepte de perdre un peu en rapidité, on peut ajouter un GC « spécialisable », « adaptable » voire « auto-adaptable », dans l'interface même de `mlgmp`. Cela consisterait soit à indiquer optionnellement à ce second GC les tailles des objets qu'on pense manipuler, soit à instrumenter les objets créés pour s'autoadapter lors de l'exécution. Pour ce faire, on peut penser à utiliser le module `GC` offert dans la bibliothèque standard d'Objective Caml, et traiter les informations sur les tailles des objets à la place du GC d'Objective Caml. Il faut alors bien être conscient qu'il s'agit d'une solution « potentiellement » moins efficace qu'une implantation directe dans le GC d'Objective Caml.

Une autre idée serait d'appeler les fonctions de finalisation lors des GC mineurs et ainsi ne pas attendre les GC majeurs pour libérer les objets (hors des tas d'Objective Caml) devenus inatteignables.

5 Annexe

```
1 #define CAML_ALLOC_DEPENDENT_MEMORY(x)      \
2     do {                                    \
3         int s ;                             \
4         s = (*mpz_val(x))->_mp_alloc;      \
5         caml_alloc_dependent_memory(s);    \
6     } while(0)
7
8 #define CAML_FREE_DEPENDENT_MEMORY(x)      \
9     do {                                    \
10        int s ;                             \
11        s = (*mpz_val(x))->_mp_alloc;      \
12        caml_free_dependent_memory(s);     \
13    } while(0)
```

Listing 4 – Macros C pour utiliser `caml_(alloc|free)_dependent_memory`

```
1 /* Dependent memory is all memory blocks allocated out of the heap
2    that depend on the GC (and finalizers) for deallocation.
3    For the GC to take dependent memory into account when computing
4    its automatic speed setting,
5    you must call [caml_alloc_dependent_memory] when you allocate some
6    dependent memory, and [caml_free_dependent_memory] when you
7    free it. In both cases, you pass as argument the size (in bytes)
8    of the block being allocated or freed.
9 */
10 CAMLexport void caml_alloc_dependent_memory (mssize_t nbytes)
11 {
12     caml_dependent_size += nbytes / sizeof (value);
13     caml_dependent_allocated += nbytes / sizeof (value);
14 }
15
16 CAMLexport void caml_free_dependent_memory (mssize_t nbytes)
17 {
18     if (caml_dependent_size < nbytes / sizeof (value)){
19         caml_dependent_size = 0;
20     }else{
21         caml_dependent_size -= nbytes / sizeof (value);
22     }
23 }
```

Listing 5 – `caml_(alloc|free)_dependent_memory` dans `ocaml-3.09.3/byterun/memory.c`

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <gmp.h>
4 int main (void)
5 {
6     int i ;
7     mpz_t a;
8     mpz_init (a);
9     mpz_set_ui (a, 999);
10    for (i=1;i<=26;i++) {
11        mpz_mul (a,a,a);
12    }
13    mpz_clear (a);
14    return (0);
15 }
```

Listing 6 – Version C du test avec les gros objets solitaires

```

1  (* ----- diverses initialisations ----- *)
2  let start = Unix.gettimeofday()
3  let tmp = string_of_float start ^ ".log"
4  let gc_out = open_out tmp
5  open Sys
6  let _ = signal sigint (Signal_handle (fun _ -> Gc.full_major()))
7  let _ = at_exit (fun () -> close_out gc_out)
8  let a =
9    Gc.create_alarm
10   (fun () ->
11     Printf.fprintf gc_out "%f\n" (Unix.gettimeofday() -. start);
12     flush stdout)

```

Listing 7 – Entête des programmes de tests écrits en Objective Caml

```

1  #!/bin/bash
2  # Bash Memory Monitor
3
4  function usage () {
5    echo "Usage: $0 program+"
6    echo "   executes every given program and trace the"
7    echo "   memory usage (a tick every 0.3 seconde)."

```

Listing 8 – Moniteur d'usage de la mémoire

6 Bibliographie

Références

- [1] *GNU Multiple Precision Arithmetic Library*
<http://www.swox.com/gmp/>
- [2] **Xavier Leroy** (with **Damien Doligez**, Jacques Garrigue, Didier Rémy and Jérôme Vouillon) : *The Objective Caml system 3.09. Documentation and user's manual*. Inria, France (1985-2006).
<http://caml.inria.fr/pub/docs/manual-ocaml/>
- [3] **Emmanuel Chailloux**, Bruno Pagano, **Pascal Manoury** : *Développement d'applications avec Objective Caml*, O'REILLY, 2000.
- [4] **Emmanuel Chailloux** : *Typage et Polymorphisme*, cours de master, UPMC, 2006-2007.
- [5] David Monniaux : *Interface GMP avec Objective Caml(mlgmp)*
<http://www.di.ens.fr/~monniaux/programmes.html>
- [6] **LORIA** (Laboratoire Lorrain de Recherche en Informatique et ses Applications) et **INRIA Lorraine** : *The MPFR Library*
<http://www.mpfr.org/>

7 Remerciements

Que les personnes qui ont contribué directement ou indirectement à cette petite étude en soient remerciées, notamment **Emmanuel Chailloux**, **Mohab Safey El Din**, **Valérie Menissier-Morain**, **Damien Doligez**, **Renaud Rioboo**, et **Jérémy Antonucci** avec qui j'avais démarré cette étude...