
# Rapport CA Projet : Garbage Collector Mark & Compact pour la Mini-ZAMG

Ewen GLASZIOU (21312544) Yacine KESSAL (21311739)

2 mars 2024

# Table des matières



## Informations

Nous avions commencé le projet chacun de notre côté avant de savoir que celui-ci pouvait être réalisé en binôme et nous avons discuté d'un blocage similaire au niveau de l'implémentation du Mark & Compact dans la phase de marquage. Cela explique que certaines parties du code contiennent en commentaires des versions différentes d'implémentation d'une même fonction. Le choix de se mettre en binôme a aussi été poussé par la difficulté à corriger et tester certaines parties de l'implémentation du GC.

# 1 Introduction

La gestion de la mémoire est un aspect crucial dans la conception et l'implémentation de machines virtuelles (VM) pour les langages de programmation, notamment pour garantir des performances optimales et une utilisation efficace des ressources systèmes. Dans le cadre du projet de Compilation Avancée de cette année, l'objectif était de s'attaquer à un défi spécifique lié à la gestion de la mémoire dans une Mini-ZAM via l'implémentation d'un Garbage Collector (GC) de type Mark & Compact.

La Mini-ZAM, inspirée par la ZINC Abstract Machine (ZAM) utilisée par OCaml, représente un modèle réduit mais fonctionnel, conçu pour illustrer les principes fondamentaux des VMs à pile, avec une stratégie d'évaluation par appel par valeur. Toutefois, à la différence de son homologue complet, la Mini-ZAM est dépourvue d'un système de gestion de mémoire automatisé, rendant nécessaire le développement d'un GC spécifique pour optimiser son fonctionnement.

Le choix d'implémenter un GC Mark & Compact s'est imposé en raison de sa capacité à minimiser la fragmentation de la mémoire, un problème courant affectant les performances des applications à long terme. Ce type de GC offre un compromis intéressant entre la complexité de mise en œuvre et l'efficacité en termes de récupération de l'espace mémoire et de maintien de la continuité des blocs mémoire alloués.

Ce rapport présente le contexte de ce projet, les défis rencontrés lors de l'implémentation du GC dans la Mini-ZAM, ainsi que les choix techniques et conceptuels effectués pour atteindre les objectifs fixés. Nous décrirons la structure générale de notre projet, les étapes clés de l'implémentation du GC Mark & Compact, et nous évaluerons les performances et l'impact de cette addition sur l'exécution de programmes OCaml dans la Mini-ZAM.

# 2 Contexte et objectifs

## 2.1 Contexte

La gestion automatisée de la mémoire est essentielle dans la conception des langages de programmation modernes et de leurs environnements d'exécution, tels que les machines virtuelles. Elle permet de libérer les développeurs de la charge complexe et sujet aux erreurs de la gestion manuelle de la mémoire, réduisant ainsi les risques de fuites de mémoire et d'autres bugs critiques. La ZAM, qui sert de machine virtuelle pour le langage OCaml, utilise un système sophistiqué de GC pour optimiser la gestion de la mémoire lors de l'exécution des programmes OCaml qu'il nous est compliqué de reproduire.

Dans le cadre de l'enseignement de Compilation Avancée, le projet vise à explorer les mécanismes internes d'une machine virtuelle par la réalisation d'une version simplifiée de la ZAM, nommée Mini-ZAM. La Mini-ZAM nous sert d'outil pédagogique pour comprendre les principes de base des machines virtuelles à pile et des stratégies d'évaluation telles que le "call-by-value", typiques des langages fonctionnels comme OCaml.

## 2.2 Objectifs

L'objectif principal de ce projet est de doter la Mini-ZAM d'un Garbage Collector de type Mark & Compact. Ce choix d'algorithme de GC n'est pas anodin : il répond à plusieurs besoins critiques en matière de gestion de la mémoire, notamment la réduction de la fragmentation mémoire, qui peut considérablement affecter les performances des applications sur le long terme. En effet, les algorithmes de GC tels que le Mark & Sweep, bien que plus simples à implémenter, laissent derrière eux des "fragments" de mémoire ce qui peut rendre difficile l'allocation de

nouveaux blocs de mémoire de taille importante et cela même quand la quantité de mémoire disponible est suffisante.

Les objectifs spécifiques de ce projet sont les suivants :

— Implémenter un système de gestion de la mémoire efficace pour la Mini-ZAM, en utilisant l'algorithme Mark & Compact, afin de minimiser la fragmentation mémoire.

— Analyser et comparer les performances de la Mini-ZAM avec et sans le GC Mark & Compact, pour évaluer l'impact de ce dernier sur l'exécution des programmes.

— Explorer les possibilités d'améliorations futures et d'optimisations du GC, pour renforcer la robustesse et l'efficacité de la Mini-ZAM (voire bonus du sujet).

Ce projet s'inscrit dans une démarche d'approfondissement des connaissances en matière de compilation et de gestion de la mémoire, en mettant l'accent sur l'importance d'un GC performant dans les environnements d'exécution des langages de programmation. À travers ce travail, nous visons non seulement à améliorer la Mini-ZAM mais également à fournir une base solide pour la compréhension des mécanismes avancés de gestion de la mémoire dans les machines virtuelles modernes.

# 3 Choix d'implémentation

L'implémentation d'un Garbage Collector pour la Mini-ZAM nécessite une série de choix techniques déterminants pour la performance, la maintenabilité et l'efficacité du système. Ce projet s'articule autour de deux choix majeurs : le langage de programmation utilisé et l'adoption de l'algorithme Mark & Compact pour le GC.

## 3.1 Langage de programmation

Le choix du langage C pour l'implémentation de la Mini-ZAM et de son GC n'est pas du au hasard. Envisager un langage de bas niveau tel que le C répond à plusieurs contraintes et objectifs spécifiques :

— **Contrôle précis de la mémoire** : Le C offre un contrôle granulaire sur la gestion de la mémoire, permettant une manipulation directe et optimisée des blocs mémoire, crucial pour l'efficacité d'un GC.

— **Performance** : Étant un langage compilé proche de l'assembleur, le C garantit des performances élevées, essentielles pour minimiser l'overhead (surcoût en performance et utilisation de mémoire nécessaire pour automatiser la gestion de la mémoire dans un programme) en mémoire nécessaire introduit par le GC lors de l'exécution des programmes.

— **Compatibilité et portabilité** : Le C est largement supporté sur de multiples plateformes, assurant une bonne portabilité de la Mini-ZAM.

## 3.2 Choix de l'algorithme Mark & Compact

L'algorithme Mark & Compact a été choisi pour plusieurs raisons clés, notamment sa capacité à réduire la fragmentation de la mémoire et à maintenir les performances de la Mini-ZAM sur le long terme. Comparé à d'autres stratégies de GC, telles que le Mark & Sweep, le Mark & Compact offre des avantages significatifs :

— **Réduction de la fragmentation** : En déplaçant et en compactant les objets actifs dans une zone contiguë de la mémoire, cet algorithme minimise l'espace perdu et facilite l'allocation de nouveaux objets.

— **Préservation de la performance** : La compaction de la mémoire permet de maintenir une performance constante de l'allocation mémoire, même après de nombreuses opérations d'allocation et de libération.

— **Simplicité relative** : Bien que plus complexe que le Mark & Sweep, l'algorithme Mark & Compact reste relativement simple à implémenter dans un contexte pédagogique, tout en offrant une approche robuste et performante de la gestion de la mémoire.

Ces choix d'implémentation s'alignent avec les objectifs du projet : fournir une Mini-ZAM performante et efficace, capable de gérer dynamiquement la mémoire lors de l'exécution des programmes OCaml. L'adoption de l'algorithme Mark & Compact, en particulier, s'inscrit dans une démarche d'optimisation et de fiabilisation de la machine virtuelle, répondant ainsi aux défis posés par la gestion avancée de la mémoire dans les systèmes informatiques modernes.

# 4 Implémentation détaillée

L'ajout d'un Garbage Collector de type Mark & Compact à la Mini-ZAM a nécessité une série d'étapes concrètes pour intégrer efficacement ce mécanisme au sein de la machine virtuelle. Cette section décrit les différentes phases de l'implémentation, mettant en avant les défis rencontrés et les solutions apportées.

## 4.1 Initialisation de la mémoire

La première étape dans l'implémentation du GC a été la préparation de l'environnement de gestion de la mémoire. La Mini-ZAM nécessite un espace mémoire initial, qui a été alloué en utilisant `malloc`. Une taille initiale de 32Ko (32KBytes) a été choisie comme point de départ pour cette zone mémoire :

```
#define Heap_size (32 * KB)
```

Cette mémoire sert de tas dans lequel le GC opérera. Le pointeur **next_alloc** est utilisé dans la gestion du tas pour suivre l'endroit actuel où le prochain objet sera alloué. C'est un pointeur global qui pointe vers la première adresse de mémoire libre dans le tas.

## 4.2 Marquage des objets accessibles

La phase de marquage est cruciale pour identifier les objets toujours accessibles dans la mémoire, qui ne doivent donc pas être collectés par le GC. Cette étape consiste à parcourir, à partir des objets racines (variables globales (environnement), pointeurs de pile, etc.), et à marquer tous les objets accessibles. Le défi dans cette phase est de garantir l'efficacité et l'exhaustivité du marquage pour éviter de collecter par erreur des objets vivants. Pour réaliser cette étapes, nous avons réaliser la fonction `mark`.

La logique de marquage fonctionne comme suit :

— Pour chaque objet vérifié, si celui-ci est un bloc et sa couleur actuelle est BLANC (indiquant qu'il n'a pas encore été visité), sa couleur est d'abord changée en GRIS, marquant le début de son traitement.

— Ensuite, la fonction parcourt récursivement tous les champs de l'objet, appliquant la même logique de marquage à chaque objet référencé.

— Une fois tous les champs visités, la couleur de l'objet est changée en NOIR, indiquant que l'objet et tous ses enfants sont accessibles et donc marqués comme vivants.

**Pseudocode pour le marquage**

**Algorithme 1 :** Marquage

```
Entrée : Un mlvalue v
1 Fonction Marquer(v) :
2     si Is_block(v) et Color(v) = BLANC alors
3         header ← (header_t*)(Ptr_val(v) − 1);
4         *header ← Make_header(Size(v), GRIS, Tag(v));
5         pour i ← 0 à Size(v) faire
6             Marquer(Field(v, i));
7         fin
8         *header ← Make_header(Size(v), NOIR, Tag(v));
9     fin
```

## 4.3 Compactage de la mémoire

Après la phase de marquage, l'étape suivante est le compactage de la mémoire. Cette étape vise à relocaliser tous les objets marqués (c'est-à-dire, accessibles) dans un bloc contigu de mémoire, réduisant ainsi la fragmentation et optimisant l'utilisation de la mémoire. Le processus nécessite une manipulation soigneuse des pointeurs pour s'assurer qu'ils pointent vers les nouvelles emplacements des objets après le compactage. Pour cette étapes, nous avons réaliser la fonction `compact` et ses fonctions auxiliaires.

Le processus de compactage se déroule en plusieurs étapes :

— Initialisation d'un mappaging d'adresses pour conserver la correspondance entre les anciennes et nouvelles adresses des objets déplacés.

— Un premier parcours de la mémoire est effectué pour calculer les nouvelles adresses de tous les objets marqués (NOIR). Durant ce parcours, chaque objet marqué est enregistré dans le mappaging d'adresses avec sa nouvelle localisation prévue.

— Un deuxième parcours met à jour tous les pointeurs dans la mémoire pour refléter les nouvelles adresses des objets déplacés, utilisant le mappaging d'adresses établi précédemment.

— Finalement, un troisième parcours effectue physiquement le déplacement des objets vers leurs nouvelles adresses, et la mémoire est ainsi compactée.

**Pseudocode pour le compactage**

**Algorithme 2 :** Calcule d'une nouvelle adresse d'un block

```
Entrée : mlvalue représentant une ancienne adresse old_address
Sortie  : La nouvelle adresse ou MLVALUE_NULL
1 Fonction Nouvelle_adresse_de(old_address) :
2     pour i ← 0 address_map_size faire
3         si address_map[i].old_address = old_address alors
4             retourner *address_map[i].new_address;
5         fin
6     fin
7 retourner MLVALUE_NULL;
```

**Algorithme 3 :** Mise à jour des pointeurs dans le tas

**Entrée :** Aucune
**Sortie :** Aucune

```
1  Fonction Mettre_à_jour_tous_les_pointeurs() :
2      current ← Caml_state− > heap;
3      tant que current < next_alloc faire
4          header ← ∗current;
5          si Is_block(header) et Color_hd(header) = NOIR alors
6              size ← Size_hd(header);
7              pour i ← 0 à size faire
8                  field_address ← &Field(current, i);
9                  si Is_block(∗field_address) alors
10                     field ← ∗field_address;
11                     field_ptr ← Ptr_val(field);
12                     si field_ptr ≥ Caml_state− > heap et field_ptr < next_alloc
                        alors
13                         new_address ← Nouvelle_adresse_de(field_ptr);
14                         si new_address ≠ (mlvalue)NULL alors
15                             ∗field_address ← new_address;
16                         fin
17                     fin
18                 fin
19             fin
20             current ← current + size + 1;
21         fin
22         sinon
23             current ← current + 1;
24         fin
25     fin
```

**Algorithme 4 :** compactage des blocks

**Entrée :** Aucune
**Sortie :** La mémoire compactée

```
1  Fonction compacter() :
2      from ← Caml_state− > heap;
3      to ← Caml_state− > heap;
4      map_capacity ← Caml_state− > heap_size;
5      address_map ← allouer(map_capacity× sizeof(address_mapping) ×10);
6      map_index ← 0;
7      tant que from < next_alloc faire
8          header ← ∗((header_t∗)from);
9          si Color_hd(header) = NOIR alors
10             size ← Size_hd(header);
11             si map_index ≥ map_capacity alors
12                 map_capacity ← map_capacity × 2;
13                 address_map ← redimensionner(address_map, map_capacity×
                    sizeof(address_mapping));
14             fin
15             address_map[map_index].old_address ← from;
16             address_map[map_index].new_address ← to;
17             map_index ← map_index + 1;
18             from ← from + size + 1;
19             to ← to + size + 1;
20         fin
21         sinon
22             from ← from+ Size_hd(header) + 1;
23         fin
24     fin
25     Mettre_à_jour_tous_les_pointeurs();
26     from ← Caml_state− > heap;
27     to ← Caml_state− > heap;
28     tant que from < next_alloc faire
29         header ← ∗((header_t∗)from);
30         si Color_hd(header) = NOIR alors
31             size ← Size_hd(header);
32             si from ≠ to alors
33                 copier(to, from, (size + 1)× sizeof(mlvalue));
34             fin
35             to ← to + size + 1;
36         fin
37         from ← from+ Size_hd(header) + 1;
38     fin
39     next_alloc ← to;
40     libérer(address_map);
41     address_map ← NULL;
```

## 4.4 Gestion des racines et accès au GC

La gestion adéquate des objets racines est essentielle pour que la phase de marquage du GC identifie correctement tous les objets vivants. Implémenter une manière efficace pour que le GC accède à ces racines est nécessaire pour le processus de marquage. Pour cela, nous avons dans ajouté dans `Caml_state` des champs correspondant au racine de notre parcours pour le marquage obtenant ainsi la structure complète suivante :

```
typedef struct _caml_domain_state {
    /* Stack */
    mlvalue* stack;
    /* Heap */
    mlvalue* heap;
    size_t heap_size; //taille macimum du tas

    /* Pointers */
    size_t sp;      // Pour stocker le pointeur de pile
    mlvalue env;    // Pour stocker l'environnement courant
    mlvalue accu;   // Pour stocker l'accumulateur

} caml_domain_state;
```

De plus, le système doit déclencher périodiquement le processus de GC basé sur certains critères, tels que des seuils d'allocation de mémoire, pour maintenir une utilisation optimale de la mémoire sans pénalités de performance du à l'éxécution du GC. Pour réaliser cela nous avons réaliser dans "src/mlvalues.c" la fonction suivante :

```
void check_and_run_gc(size_t size) {
    if ((char*)next_alloc + ((size) + 1) * sizeof(mlvalue) >
    (char*)Caml_state->heap + Caml_state->heap_size) {
    run_gc();
    if ((char*)next_alloc + ((size) + 1) * sizeof(mlvalue) >
    (char*)Caml_state->heap + Caml_state->heap_size) {
    /* si après le GC on a toujours pas assez de place on retourne une
    erreur. Plus tard on pourrais faire un realloc */
      fprintf(stderr, "GC bien éxécuté, mais pas d'espace dans le Tas ! \n");
      exit(EXIT_FAILURE);
    }
  }
}
```

Cette fonction est appelée par la macro définis dans "src/mlvalues.h" responsable de la créations des blocs et a pour objectif de faire un appel le GC uniquement si l'ajout du nouveau block provoquerai un dépassement de la taille alloué au tas. Puis si malgré le GC la place n'est pas suffisante, provoque une erreur d'exécution pour du au surchargement du tas.

Finalement, la fonction `gc()` suites les étapes décrites dans le sujet [document 2] pour l'éxecution d'un Mark & Compact.

**Pseudocode pour le garbage collector**

**Algorithme 5 :** Exécution du Garbage Collector

```
1  Fonction run_gc() :
       // Accédez aux racines
2      accu ← Caml_state− > accu
3      stack ← Caml_state− > stack
4      sp ← Caml_state− > sp
5      env ← Caml_state− > env
       // Étape 1 : marquage
6      pour i ← 0 à sp − 1 faire
7          Marquer(stack[i])
8      fin
9      Marquer(env)
10     Marquer(accu)
       // Étape 2 : Compacter
11     Compacter()
12     printf"gc fini\n"
13     reset_colors()
       // Optionnel : Afficher l'état du tas
       // print_heap_state()
```

# 5 Tests et résultats

La validation fonctionnelle et la mesure des performances du Garbage Collector intégré à Mini-ZAM ont été réalisées à travers une série de tests. Ces tests, situés dans le dossier "tests/", ont été conçus pour évaluer l'efficacité du GC dans différents scénarios d'utilisation de la mémoire. Un aspect crucial de ces tests a été l'observation de l'impact de la taille du tas, définie dans `config.h`, sur la capacité du GC à exécuter avec succès les scripts de test.

- Avec une taille de tas initiale de **32 KB**, certains scripts de test, notamment `list_1.txt, list_2.txt,list_3.txt et list_4.txt`, ne pouvaient pas s'exécuter correctement, indiquant que la taille du tas était insuffisante pour supporter les allocations mémoire requises par ces tests.
- En augmentant la taille du tas à **256 KB**, le test `list_1.txt` s'exécute sans problème, ce qui montre une amélioration directe liée à l'augmentation de la taille du tas.
- Une taille de tas de **4.5 GB** permet l'exécution réussie de tous les quatre scripts dans "tests/bench/", démontrant la capacité du GC à gérer efficacement des charges de travail importantes lorsque suffisamment de mémoire est disponible.
- Cependant, pour une taille de tas de 4 GB, l'exécution de l'ensemble des tests provoque une erreur. Mais après plusieurs tests avec la version sans GC, ces erreurs semblent provenir de l'implémentation de la Mini-ZAM.

Ces observations soulignent l'importance d'ajuster la taille du tas en fonction des besoins spécifiques des programmes exécutés sur Mini-ZAM. Elles révèlent également l'impact significatif que la gestion de la mémoire a sur la performance et la fiabilité de l'exécution des programmes. En conséquence, une partie cruciale du processus de développement et de test implique la recherche d'un compromis optimal entre les ressources allouées et les exigences des applications, afin de garantir à la fois l'efficacité et la robustesse du système.

# 6 Améliorations potentielles

L'implémentation actuelle du Garbage Collector (GC) pour Mini-ZAM ouvre la voie à plusieurs améliorations potentielles visant à optimiser ses performances. Parmi ces améliorations, nous envisageons :

— **Redimensionnement dynamique du tas** : Une gestion plus dynamique de la mémoire permettrait d'adapter la taille du tas en fonction des besoins réels des applications exécutées. Ceci pourrait être réalisé en surveillant l'utilisation de la mémoire et en ajustant la taille du tas via un mécanisme de `realloc`, comme indiqué dans la proposition de bonus du projet. Ici nous pensons à doubler la taille du tas dans le cas où après l'exécution du GC, l'espace disponible n'est toujours pas suffisant.

— **Optimisation des algorithmes de marquage et de compactage** : Des recherches sur des algorithmes plus efficaces pour le marquage et le compactage pourraient réduire le temps de pause GC, améliorant ainsi la réactivité des applications (Correction et optimisation de notre code actuelle).

— **Gestion spécifique des charges de travail** : Le développement de stratégies de GC adaptées à certaines classes d'applications pourrait optimiser l'utilisation de la mémoire pour des scénarios d'utilisation spécifiques (Bien trop complexe actuellement).

Le redimensionnement dynamique du tas est particulièrement intérressant pour améliorer la gestion de la mémoire. Voici un pseudocode illustrant comment nous pensions intégrer cette fonctionnalité :

**Algorithme 6 :** Redimensionnement dynamique du tas

**Entrée :** La taille requise *size*

**Fonction** `realloc_tas(`*size*`)` **:**
  *nouvelleTaille* ← *Caml_state*− > *heap_size* × 2;
  *nouveauTas* ← **realloc**(*Caml_state*− > *heap*, *nouvelleTaille*);
  **si** *nouveauTas* ***n'est pas*** *NULL* **alors**
    *Caml_state*− > *heap* ← *nouveauTas*;
    *Caml_state*− > *heap_size* ← *nouvelleTaille*;
    `// Mettre à jour next_alloc selon le nouveau tas`
    *next_alloc* ←
     (char∗)*nouveauTas* + ((char∗)*next_alloc* − (char∗)*Caml_state*− > *heap*);
    `// Mettre à jour les pointeurs des blocs pour qu'il pointes`
       `correctement vers les nouvelles adresses`
  **sinon**
    **exit**(EXIT_FAILURE);
  **fin**

Cette algorithme serai appelé dans "src/mlvalues.c/void check_and_run_gc(size_t size)" dans le cas où l'exécution du GC n'as pas libérer suffisamment de place dans le tas.

# 7 Conclusion

Le projet de Garbage Collector Mark & Compact pour Mini-ZAM a été une occasion précieuse d'approfondir notre compréhension des mécanismes de gestion de la mémoire dans les machines virtuelles. En dépit des défis rencontrés, les résultats obtenus soulignent l'importance d'une gestion efficace de la mémoire pour la performance des applications. Les améliorations proposées, en particulier le redimensionnement dynamique du tas, représentent des pistes prometteuses pour des recherches et développements futurs. Ce projet jette ainsi les bases pour

l'exploration continue des stratégies de gestion de la mémoire, contribuant au domaine de la compilation et de l'exécution des langages de programmation.

# 8 Références

1. Projet de Compilation Avancée Mini-ZAM : Interprète de bytecode fonctionnel
2. Projet de Compilation Avancée Garbage Collector pour la Mini-ZAM v0.3